OpenECHO for Processing

Tutorial

Produced by the Research Project on Connection and Control Technology Research Program for Energy Management System Standardization

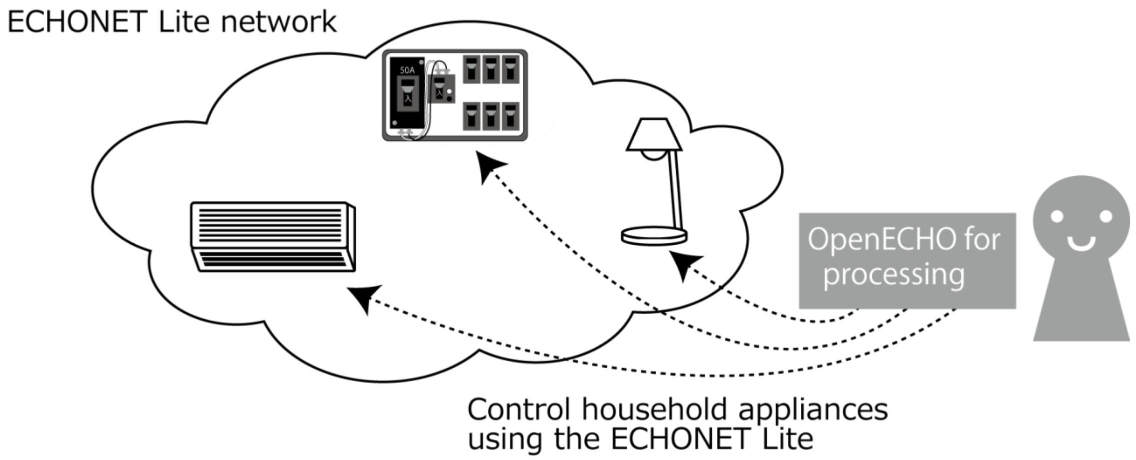
Smart House Research Center, Kanagawa Institute of Technology
Sony Computer Science Laboratories, Inc.

Table of Contents

Introduction

This tutorial will describe how to use the **Processing** library "**OpenECHO for Processing**", based on the open-source project **OpenECHO**. After reading this tutorial, you will have learned how to complete the following tasks.

- Control household appliances that support the **ECHONET Lite** protocol
- Join devices without built-in **ECHONET Lite** support, including homemade devices that can be controlled via serial communications, standard infrared terminals, or other techniques, to the **ECHONET Lite** network.
- Combine your own hand-crafted services with externally purchased services in a shared-control framework.



To follow this tutorial, you will need basic Java programming skills and some knowledge of **Processing**. You will not need any prior knowledge of **ECHONET Lite**.

Although the **ECHONET Lite** protocol does not specify any particular physical layer, **OpenECHO** is implemented atop IPv4. For this reason, **OpenECHO for Processing** may only be used on **ECHONET Lite** networks that use IPv4 as the physical layer.

Background

OpenECHO for Processing is a library designed to make it easy to build programs in the **Processing** environment for use with **ECHONET Lite**, a protocol released in December 2011 that allows networked devices such as appliances and sensors to exchange information. The **OpenECHO for Processing** library, the sample programs, and this tutorial were all developed by Sony Computer Science Laboratories, Inc. Budget resources were provided by the Smart House Research Center at Kanagawa Institute of Technology, through its Research Project on Connectivity and Control Technologies for Energy Management System Standardization.

HEMS (ECHONET Lite) Certification Center at the Kanagawa Institute of Technology:

http://smarthouse-center.org/

What is ECHONET Lite?

ECHONET Lite is a communications protocol approved by the ECHONET Consortium that has become an international standard via ISO/IEC. Its features include a simple control system, a wide variety of device object definitions, and a physical-layer-independent design, which ensures that the protocol may be used atop existing physical layers. The protocol is expected to achieve widespread adoption in the coming years. In particular, **ECHONET Lite** has been recognized by Japan's Ministry of Economy, Trade, and Industry (METI) as a communications protocol for home networks, particularly for home energy management systems or HEMS. Its adoption has been determined as the protocol for future smart meters. These provisions are designed to make the protocol attractive to manufacturers to promote its rapid adoption. For more information on the background and detailed specifications of the **ECHONET Lite** protocol, visit the home page of the ECHONET Consortium.

ECHONET Consortium website**:** http://www.echonet.gr.jp/english/index.htm

What is OpenECHO?

Because the **ECHONET Lite** specification is fully open to the public, anybody can implement the protocol. Sony Computer Science Laboratories, Inc. (Sony CSL) has developed a class library that implements the protocol in Java; this class library is distributed as open-source software. Because **ECHONET Lite** does not specify a physical layer, it may operate over IPv4, IPv6, ZigBee, or a variety of other protocols, but specific policies are available for the case of IP implementations. **OpenECHO** is an IPv4 implementation of **ECHONET Lite** that is based on these policies. As of January 2014, ECHONET Lite includes detailed guidelines (that is, predetermined specifications) for 90 devices, and **OpenECHO** supports all of these. (In addition, **OpenECHO** supports a device known as a **Controller**, which does not have detailed specifications included in the **ECHONET Lite** specification.)

OpenECHO distribution site: https://github.com/SonyCSL/OpenECHO/

What is Processing?

Processing is an open-source programming environment developed at the Massachusetts Institute of Technology (MIT) to realize the notion of a "software sketchpad." The environment was originally developed for applications in programming education. However, the simplicity of the environment's development tools, and the plentiful libraries and sample programs bundled with it, have earned it broad popularity. Today, **Processing** is used for a wide range of applications not limited to educational purposes, including media art, simple hardware control, prototyping of homemade systems, and Android development. In practice the environment does not require the use of any specialized languages, and actual programming is done in Java. Visit the **Processing** website for more information on the environment.

Processing website: http://processing.org/

What is OpenECHO for Processing?

OpenECHO for Processing is a version of the **OpenECHO** software compiled as a library to be used with **Processing**. The **Processing** environment can import standard Java libraries, which means the library can be built with almost no modifications to the **OpenECHO** source code. All that changes is the compilation procedure. However, in order to minimize the size of the compiled code, classes providing default implementations of node profiles and device objects have been added. In addition, sample programs have been developed together with the library itself. The discussions of this tutorial are based on these sample programs.

The sample code is written for the **Processing** environment. However, the portions of the code that access the library may be used, without modification, to access **OpenECHO** directly from standalone Java programs. Thus this tutorial may also be used as an introduction to **OpenECHO**. **Processing** is an outstanding development environment with an extremely low barrier to entry for novices. However, it does not incorporate the useful features offered by more sophisticated source-code editors such as **Eclipse**. These features include code-assistance functionality to display lists of class members and automatic generation of required methods when implementing abstract classes. Once you have become familiar with **OpenECHO for Processing** development, you will find that the most troublesome part of the development process is the need to work with numerous methods for each distinct device. For this reason, once you have familiarized yourself with the basic usage of the library, we encourage you to use a source-code editor with code-assistance features. This will dramatically reduce programming hassles and speed your development process.

Preparations for working through this tutorial

Before launching into the tutorial, let's make sure all preparations are ready for executing the sample programs it includes. These preparations include installing the **OpenECHO for Processing** and **ControlP5** libraries within the **Processing** environment. To do this, create a folder named `libraries` within the folder you use to store **Processing** sketches and unpack the two folders that come bundled with the **OpenECHO for Processing** library, namely:

- the `controlP5` folder, and
- the `OpenECHO` folder.

Now restart **Processing**. You will see that two new options (**ControlP5** and **OpenECHO**) have been added to the **Sketch->Import Library** menu. If you do not see these options in the menu, use **File->Preferences** to check the location of the currently active sketch folder, modify it as necessary, and restart the **Processing** environment.

In the unpacked folders you will find the libraries, the sample programs, and an HTML-format library reference that is generated from the **OpenECHO** source code. This reference is useful for looking up functionality and names of properties incorporated in the various classes. (This library reference also includes a simple description copied from the English version of the **ECHONET Lite** public documents, discussed below. However, this description is somewhat incomplete.)

ControlP5 is a GUI library for use with **Processing** that makes it easy to use widgets such as buttons and text boxes. In this tutorial, we will use it to create remote-control buttons.

ControlP5 website: http://www.sojamo.de/libraries/controlP5/

How to access the open specifications for ECHONET Lite

Technical data related to **ECHONET Lite** may be found at this website:

ECHONET Specifications (Available to the general public):
http://www.echonet.gr.jp/english/spec/index.htm

Note that there is no need to read through the full **ECHONET** documentation to use **OpenECHO for Processing.** For this tutorial, the following two pieces of documentation are most relevant:

- The sections *Overview of Device Object Super Class Specifications* and *Node Profile Class: Detailed Specifications* in the document ***ECHONET Lite Specification, Part II: ECHONET Lite Communication Middleware Specification.***
  These documents contain information on the properties contained within the `NodeProfile` object, discussed below. As of January 2014, the most recent version of the **ECHONET Lite** specification is Version 1.10, and the above information may be found in Section 6.10.1 (P. 6-4) and Section 6.11.1 (P. 6-6) of the Japanese version of the document. The most recent version of the document (in Japanese) as of January 2014 may be downloaded from this location:

  http://www.echonet.gr.jp/spec/pdf_110_lite/ECHONET-Lite_Ver.1.10_02.pdf

As of January 2013, Version 1.01 of the **ECHONET Lite** specification has not yet been translated into English. The most recent version of the specification for which an English translation exists is Version 1.00. However, the portions of the specification pertaining to `NodeProfile` objects may be found in the same sections (and even the same page numbers) in both versions of the document: Section 6.10.1 (Page 6-4) and Section 6.11.1 (Page 6-6). The English translation of the Version 1.00 specification may be downloaded from this location:

http://www.echonet.gr.jp/english/spec/pdf_v100_lite_e/SpecLiteVer.1.0_e_02.pdf

- The document ***APPENDIX: Detailed Requirements for ECHONET Device Objects.***

This document contains information on the properties contained within the various device objects. Because this is a reference document, you will only need to consult the sections pertaining to the particular devices or sensors you will be using. As of January 2014, the most recent version of this document is **Release D** (Japanese Version). However, **OpenECHO** is based on **Release C** (English Version). Property names are also based on the English version of the document. The differences between Releases C and D are summarized in the *Revision History* at the beginning of Release D. No device object type has been changed, but some device names and detailed regulations of properties have been changed. The most recent version of this document as of January 2014 may be downloaded from this location:

http://www.echonet.gr.jp/spec/pdf_spec_app_d/SpecAppendixD.pdf (Jpn)
http://www.echonet.gr.jp/english/spec/pdf_spec_app_c_e/SpecAppendixC_e.pdf (Eng)

Chapter 1 Creating a first node

In this chapter, we will learn how to write a program to join the **ECHONET Lite** network and access information on other devices.

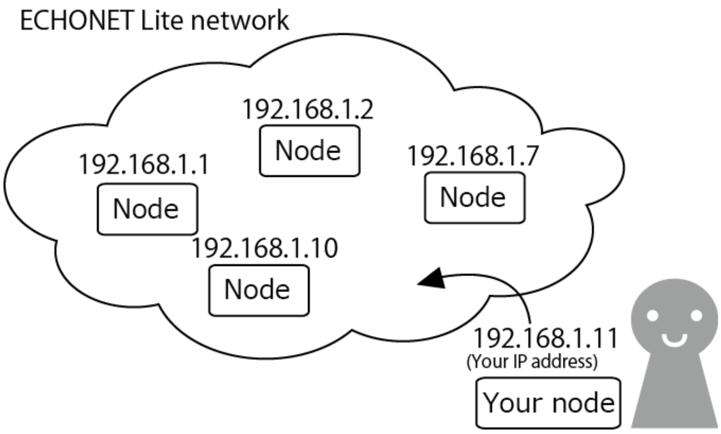
Sample program: `Tutorial1_HowToMakeANode`

> In this tutorial, each chapter begins with the name of a sample program. These sample programs may be accessed by launching **OpenECHO for Processing** and opening the **File -> Examples -> Contributed Libraries -> OpenECHO** menu. The programs may be executed by loading them as **Processing** sketches.

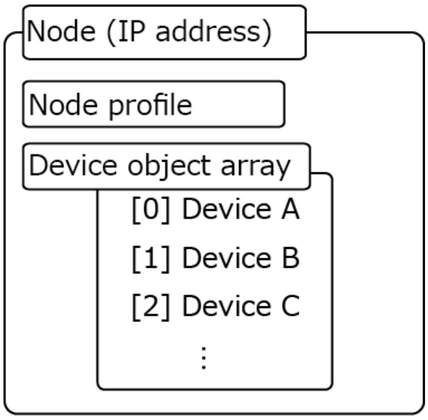
What is a node?

In **ECHONET Lite**, communication between devices (such as appliances or sensors) is carried out on a node-by-node basis. A **node** is an entity on an **ECHONET Lite** network; there are no entities other than nodes on **ECHONET Lite** networks. For this reason, before joining a network we must first create a node corresponding to our own machine.

Each node has a unique ID. In **OpenECHO**, which is implemented atop IPv4, the IP address of the node is used as the node's ID. Because all nodes must have distinct IDs, there can be no more than one node assigned to a single IP address. It is important to keep in mind that, when operating in a DHCP environment, IP addresses (and thus node IDs) may vary.



A node consists of a *node profile* and a *device object array* containing the names of one or more device objects. Device objects may be sensors or objects corresponding to individual appliances. The node profile stores information about the node and the current status of the node. Some of the information stored in the node profile may be modified from outside the node. Note that a single node can represent multiple devices, so it is *not* necessarily the case that each node corresponds to a single appliance.



Creating a first node

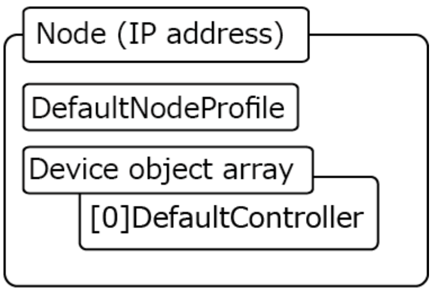
Let's create our first node. The IP address allocated to the machine on which the program runs will automatically be taken as the node's ID, so there is no particular need to specify this address. The items that we must create ourselves are the following:

- a node profile (an instance of the `NodeProfile` class)
- an array of device objects (an array of `DeviceObject` classes).

To specify these items and create the node, we invoke the `Echo.start` method.

```
try {
        Echo.start( new DefaultNodeProfile()
                        ,new DeviceObject[]{new DefaultController()} );
} catch( IOException e){
        e.printStackTrace();
}
```

Here we have used `DefaultNodeProfile` as our node profile, and our array of device objects contains just the single entry `DefaultController`. These parameters are passed to the `Echo.start` method to create the node. The device array may not be empty, so we create one instance of a particular device known as a controller.



For simplicity in this example we have used default classes for both the node profile and the controller. However, this means that default implementations of all class methods will be used, whereas we should modify these methods as appropriate for the implementation of our device. Depending on how the device is used within the network, this may lead to incorrect behavior. We will discuss proper implementation methods that avoid this difficulty in Chapter 5.

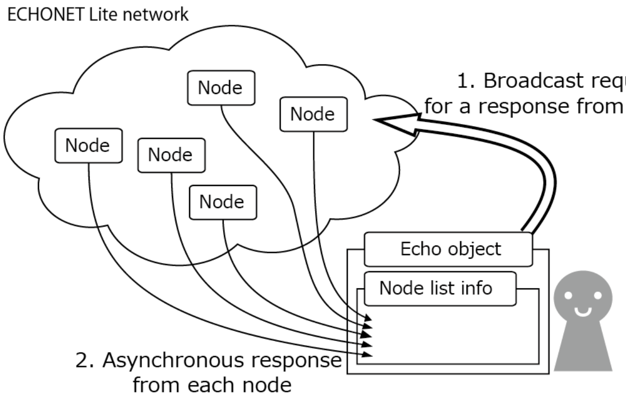
Searching for other appliances

Thus far we have created a node and joined the network. Next let's gather information on the other nodes present on the network. As noted above, information on each node is stored in the node profile maintained by that node. We can query a node profile for a list of all devices included in that node. Thus, in order to obtain information on all devices connected to a network, we must first ask all nodes on the network to send us lists of the device objects stored within their node profiles.

NodeProfile.informG().reqInformInstanceListNotification().send();

This code snippet issues a request, to all `NodeProfiles` present on the network, for arrays of device objects.

We will discuss the various types and formats of requests in Chapter 2 and thereafter. For now, just think of the code snippet above as a fixed incantation.



Within **OpenECHO**, each time a node on the network responds to the above request, the incoming node data are stored within the `Echo` object. The point that you need to keep in mind is that, within **OpenECHO**, the transmission and reception of requests like this are handled asynchronously. This means that, after you send a request, the system does not automatically wait to receive responses. Instead, you must choose to await the reception of responses from other nodes before launching into any subsequent processing. The sample programs carry out updates every 10 seconds.

Next, assuming we have waited enough time for responses to have been received from all nodes on the network, let's display the information we have gathered. Because the information is stored within the `Echo` object, we will begin by calling class methods to retrieve it.

```
EchoNode[] nodes = Echo.getNodes();
```

Note that the data returned by `getNodes` contains data on your own node.
If you wish to gather information on your node only, you can instead use `getNode`.

```
EchoNode localNode = Echo.getSelfNode();
```

Note: This method was called getNode() in the editions issued on Sep 9, 2013 or earlier.
Having executed both of the above function calls, let's display the node IDs (the IP addresses) received from all the nodes on our network. The `EchoNode.getAddress` routine returns an `InetAddress` (an object representing an IP address), so we will use `getHostAddress` to convert it to a character string.

```
for(EchoNode en : nodes){
   if(en == localNode){
      println("Node id = " + en.getAddress().getHostAddress() + "(local)");
   }else{
      println("Node id = " + en.getAddress().getHostAddress());
   }
}
```

Next, we will access the node profiles for each node to read out the information they contain.

```
for(EchoNode en : nodes){
   println("NodeProfile=" + en.getNodeProfile()) ;
}
```

This produces output looking something like this:

NodeProfile=groupCode:03,classCode:f0,instanceCode:01,address:192.168.0.3

Note: The IP address at the end of this line (192.168.0.3) will differ depending on your environment.

In **ECHONET Lite**, `NodeProfiles` and the associated device objects have `groupCode` and `classCode` attributes depending on the type of device. Values for these fields are tabulated in the **ECHONET Lite** specification. The `instanceCode` attribute identifies instances of devices within a given node; the first device of a given type has `instanceCode=1`, the second device of that type has `instanceCode=2`, etc. Taken together, these three attributes (`groupCode`, `classCode`, and `instanceCode`) uniquely specify a single device object within a node. In ECHONET Lite, these attributes are called ECHONET objects (EOJ) and are labeled as [03.f0][01]. ***Note that all such values in this tutorial are expressed in hexadecimal with the `0x` prefix omitted.***

Next let's list the node's device objects. To fetch a node's device objects, we call `EchoNode.getDevices`. Within the body of the loop that we wrote above to query all nodes, we will add an inner loop to access each device.

```
for(EchoNode en : nodes){
   println(" Devices:");
   DeviceObject[] dos = en.getDevices();
   for(DeviceObject d : dos){
      println("    " + d);
   }
}
```

What we obtain here is output similar to that generated by `NodeProfile` above. The `groupCode` and `classCode` fields allow us to determine the type of the device.

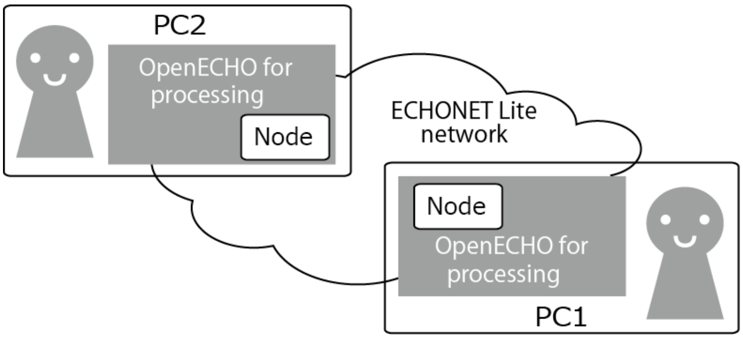
The code listing below aggregates all the code snippets we have discussed thus far.

```
import java.io.IOException;
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.node.EchoNode;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.DeviceObject;
```

```
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

void setup(){
  try {
    Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
  } catch( IOException e){ e.printStackTrace(); }

  while(true) {
    try {
      // Query existing node profiles (amounts to finding existing nodes)
      NodeProfile.informG().reqInformInstanceListNotification().send();

      EchoNode[] nodes = Echo.getNodes() ;
      for( int i=0;i<nodes.length;++i ){
          EchoNode en = nodes[i] ;
          println( "node id = "+en.getAddress().getHostAddress()) ;
          println( "node profile = "+en.getNodeProfile()) ;

          DeviceObject[] dos = en.getDevices() ;
          println( "There are "+dos.length+" devices in this node");

          for( int j=0;j<dos.length;++j ){
            DeviceObject d = dos[j] ;
            println("device type = "+d.getClass().getSuperclass().getSimpleName()) ;
          }
          println("----");
       }
    } catch( IOException e){ e.printStackTrace(); }

    // Wait 10 seconds.
    try { Thread.sleep(10000); } catch (InterruptedException e) { e.printStackTrace(); }
  }
}
```

Executing this program produces a listing of all nodes that are present on the network, together with information on the device objects they contain.

If there are no **ECHONET Lite** appliances in the house, we can run the same program from other machines connected to the same local area network. We should then be able to discover those nodes as other new nodes.



Chapter 2 Reading information on other nodes

In the previous chapter, we created our first node, learned how to obtain a list of the nodes on the network, and studied how to display lists of devices contained within those nodes.

In this chapter we will proceed a step further, learning how to read individual node data from nodes of particular types. We will learn how to fetch data on electric power usage from a "power distribution board" object, which will generally be defined for households equipped with an **ECHONET Lite**-compliant HEMS device. (This device performs tasks such as fetching electric power usage data from sensors installed in the home's central power distribution board and storing the output of solar panels in batteries).

Sample programs: `Tutorial2a_PowerDistributionBoard`
`Tutorial2c_TemperatureHumiditySensor`

Using an EventListener to perform device-specific operations

To read information from nodes, we will set up something known as an `EventListener`. This is a general technique used to receive information on various **OpenECHO** status changes. Here we will set up a handler routine that will be called whenever a new device is discovered on the network.

In the previous chapter, we waited for responses to be received from all nodes, then used `Echo.getNodes()` to fetch a list of nodes and processed the device information they contained. In contrast, by using an `EventListener`, we can arrange to take certain actions whenever a device in which we are interested is discovered on the network. Because we only need to create an `EventListener` once, we will use **Processing**'s `setup` method. The `EventListener` class includes overrideable handlers for each device defined by **ECHONET Lite**. Users may override the handler corresponding to the node they wish to access, replacing it with a handler that performs the desired operations. The handlers defined within the `EventListener` class are all empty (they don't do anything), but when overriding a handler it is nonetheless good practice to call the existing handler in the parent class. Among other things, this eliminates the possibility of confusion over function names.

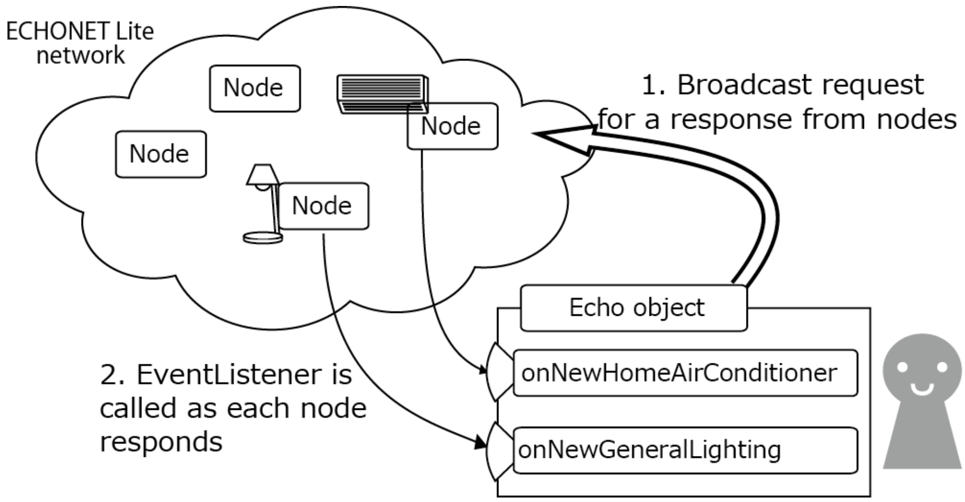
To register an `EventListener`, you call `Echo.addEventListener`. To try this out, let's register an empty `EventListener`. (Because `EventListener` does nothing by default, the following code has no effect.)

```
Echo.addEventListener(new Echo.EventListener() {});
```

Next let's add some code to be executed whenever a power distribution board is discovered. To this end, we will override the following `Echo.EventListener` handler:

```
void onNewPowerDistributionBoardMetering(PowerDistributionBoardMetering)
```

Within the body of our new handler, we will insert code to call the handler (which has the same name) in the parent class:




```
Echo.addEventListener(new Echo.EventListener() {
  public void onNewPowerDistributionBoardMetering(
                                        PowerDistributionBoardMetering device){
      super.onNewPowerDistributionBoardMetering(device);
  }
});
```

Note: There are as many overrideable handlers of this type as there are **ECHONET Lite** device objects. To see a list of them all, consult the definition of `com.sonycsl.echo.Echo.EventListener` within the **OpenECHO** reference.

Using a Receiver to receive device information

In this case we want to query the power distribution board for the current electric power usage. To this end, we will add code to our overridden version of *onNewPowerDistributionBoardMetering* to perform the following tasks:

- Register an object known as a `Receiver` for the power distribution board object.
- Send a request to the power distribution board.

In general, when working with networked appliances it takes some time after sending a request before a response is received. `Receiver` is a class used to process these sorts of asynchronous responses. In order to obtain power usage information from the power distribution board, before sending the data request we will first override the `onNewPowerDistributionBoardMetering` handler within the `Receiver` class. This will ensure that we are able to receive the response when it arrives. In the example below, we will also set up a handler known as `onGetOperationStatus`, which is used to determine whether or not the power is turned on.

```
Echo.addEventListener(new Echo.EventListener() {
  // called whenever a new power distribution board is discovered on the network
  public void onNewPowerDistributionBoardMetering(PowerDistributionBoardMetering device){
    // make sure we call the method in the parent class
    super.onNewPowerDistributionBoardMetering(device);

    //before calling Get... methods below, we set up a Receiver to receive the responses
    device.setReceiver(new PowerDistributionBoardMetering.Receiver() {
      // handler to obtain power usage information
      protected void onGetMeasuredInstantaneousCurrents(EchoObject eoj, short tid, byte esv,
                                                        EchoProperty property, boolean success) {
        super.onGetMeasuredInstantaneousCurrents(eoj, tid, esv, property, success) ;
        System.out.println("GetMeasuredInstantaneousCurrents : "+toHexStr(property.edt));
      }
      // handler to obtain information on whether or not the power is turned on
      protected void onGetOperationStatus(EchoObject eoj, short tid, byte esv,
                                          EchoProperty property, boolean success) {
       super.onGetOperationStatus(eoj, tid, esv, property, success);
       System.out.println("PowerDistributionBoardMetering power : "+toHexStr(property.edt));
      }
    });
    // done setting up receivers

    // send requests to the power distribution board (see text below)
    try{
      device.get().reqGetMeasuredInstantaneousCurrents().reqGetOperationStatus().send() ;
```

```
      } catch(IOException e){
        e.printStackTrace();
      }
      // done with processing performed upon discovery of a power distribution board
    }
});

// done setting up the EventListener
// OK, all preparations are complete; now ask for a list of all nodes on the network
try {
   Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
   NodeProfile.informG().reqInformInstanceListNotification().send();
} catch( IOException e){ e.printStackTrace(); }
```

Note: `toHexStr()` is a function that converts a byte array to a hexadecimal character string, defined in the sample programs.

When this program is executed, it queries all power distribution boards present on the network for their current power consumption and whether or not they are turned on. For the latter query, the values `0x30` and `0x31` correspond respectively to power on and power off. The program then produces a listing the power on/off status and the power consumption (if it was successfully obtained) for all devices.

All handlers within the `Receiver` class take the same arguments, regardless of the type of handler. When reading device information, the most important argument is `byte[] property.edt`. For example, when reading power usage, the returned power usage value is stored in this argument. When the data returned are too large to fit in a single `byte`, they will be divided up into multiple `byte`s. Consult the **ECHONET Lite** specification to see what types of results are returned by various requests.

The significance of the other arguments is as follows. `eoj` is the device that received the request. `tid` is the frame number. `esv` is the type of request. `property` is the received data. `success` indicates whether or not the operation succeeded. `property` contains the members `epc`, `pdc`, and `edt`. `epc` is the property ID (think of this as the ID of a variable indicating the state of the device). `edt`, as noted above, is the received byte string. `pdc` is the length of `edt`.

When we have finished registering the `Receiver`, we issue the actual requests for devices to send us information. This is done by the following function call:

device.get().reqGetMeasuredInstantaneousCurrents().send();

Here `device` is an object representing the device itself; it is specified by **Echo** as an argument to the `EventListener` (in this case, `onNewPowerDistributionBoardMetering`). We use `device.get()` to fetch a "getter" for this device. A getter represents the values that may be requested from a device in the form of methods; calling the member functions of the getter sends data requests to the device. More specifically, we call the method within the getter corresponding to the particular request we wish to issue; this returns a class whose `.send()` method we call to send the request to the device object. In the actual code, we must use a `try-catch` construction to catch any `IOException`. Note that getters are device-specific; the actual requests that may be sent to a device depend on the implementation of the device. There may also be vendor-specific requests, and requests that do not result in the return of any data.

It is also possible to send multiple simultaneous requests. For this purpose we chain methods together, as in the following example. (Here we first call `reqGetMeasuredInstantaneousCurrents()`, then call `reqGetOperationStatus()` on the value returned by the first call.)

```
try{
  device.get().reqGetMeasuredInstantaneousCurrents().reqGetOperationStatus().send() ;
} catch(IOException e){
     e.printStackTrace();
}
```

In this case we used `get`, but `set` and `inform` are also available. For these methods as well, similar requests can be chained together as we did above, with `send()` called at the end to send all requests simultaneously. The three types of requests have the following basic significant.

- `get`*:* A request that asks a particular device object for information. The responses will be sent only back to us.

- `set`*:* A request that sends (writes) information to a particular device object. By default, it is also possible to receive information on whether or not the operation succeeded.

- `inform`: A request that asks a particular device object for information. The difference between `inform` and `get` is that `inform` causes responses to be sent to all nodes on the network, including us. However, in so-called "non-responsive" cases, in which the target objects are unable to generate responses, this fact will be returned as an error only to the caller.

Within **ECHONET Lite**, each device object contains something known as a set of "properties," which are akin to member variables. These properties may be accessed or modified `get`, `set`, and `inform`. In this case, we only needed to read the power usage information, so we used `get`; however, if (for example) we had wanted to turn a light on or off, this would be a write operation, so instead we would have issued `set` requests to control the device.

Also, whereas these methods issue requests to only a single device object on a network, it is also possible to send requests to all device objects of the same type on a network. The methods used in this case are `getG`, `setG`, and `informG` (the `G` suffix stands for "Group"). These are static methods in which no specific target object is specified; instead, the methods are invoked directly from the class name. We will discuss these methods in more detail in Chapter 3.

To see all the types of request that may be sent to the power distribution board in this case, we would consult the list of getter/setter methods for `PowerDistributionBoardMetering`. Similarly, the library reference contains lists of getter and setter methods for all the various types of device objects.

Our sample program above ends with a code snippet that requests a list of nodes. This ensures that our `EventListener` will be called whenever a new node is discovered on the network. In contrast to the example discussed in Chapter 1, in this case there is no need to issue multiple repeated requests for lists of nodes.

send via TCP

In the previous examples, the `send()` method was used to send requests. This method is used to send a request via UDP. **ECHONET Lite** specification Ver. 1.10 added a new protocol for sending requests via TCP. **OpenECHO** provides the `sendTCP()` method, which sends a request via TCP. You can simply replace `send()` with `sendTCP()`.

However, note that `sendTCP()` cannot be used for objects obtained from `setG`, `getG` and `informG`, because TCP requests cannot perform multicasting.

Another example: Reading information from a sensor

Here's a second example program in which we read information from a temperature sensor and a humidity sensor. Although this sample program is not substantively different from the power distribution board program, it offers one more example to reinforce the concepts introduced above.

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.sensor.TemperatureSensor ;
import com.sonycsl.echo.eoj.device.sensor.HumiditySensor ;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

int bti(byte[] b){
  int ret = 0 ;
  for( int bi=0;bi<b.length;++bi ) ret = (ret<<8)|(int)(b[bi]&0xff) ;
  return ret ;
}

void setup(){
  // ensure that a log is written to System.out
  //Echo.addEventListener( new Echo.Logger(System.out) ) ;

  Echo.addEventListener(new Echo.EventListener() {
    public void onNewTemperatureSensor (TemperatureSensor device){
      println( "Temperature sensor found.") ;
      device.setReceiver( new TemperatureSensor.Receiver(){
        protected void onGetMeasuredTemperatureValue(EchoObject eoj, short tid, byte esv,
                                                     EchoProperty property, boolean success) {
          super.onGetMeasuredTemperatureValue(eoj, tid, esv, property, success);
          int ti = bti(property.edt) ;
          // In ECHONET Lite, returned temperature values are 10x the actual temperature, so
we divide by 10.
          float tmpe = ti*0.1;
          println("Temperature : "+ tmpe + " degree");
        }
      }) ;
      try {
        device.get().reqGetMeasuredTemperatureValue().send() ;
      } catch( IOException e){ e.printStackTrace(); }
    }

    public void onNewHumiditySensor (HumiditySensor device){
      println( "Humidity sensor found.") ;
      device.setReceiver( new HumiditySensor.Receiver(){
        protected void onGetMeasuredValueOfRelativeHumidity(EchoObject eoj, short tid
```

```
                                , byte esv, EchoProperty property, boolean success) {
          super.onGetMeasuredValueOfRelativeHumidity(eoj, tid, esv, property, success);
          println("Humidity : "+property.edt[0]+"%") ;
        }
      }) ;
      try {
        device.get().reqGetMeasuredValueOfRelativeHumidity().send() ;
      } catch( IOException e){ e.printStackTrace(); }
    }
  }) ;
  try {
    Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
    NodeProfile.informG().reqInformInstanceListNotification().send();
  } catch( IOException e){ e.printStackTrace(); }
  println("Started") ;
}
```

When this program is executed, it will print "`Temperature sensor found`" if it finds a temperature sensor, and "`Humidity sensor found`" if it finds a humidity sensor. In each case the program will also display measured values obtained from the sensors.

To conclude this chapter, we list the EOJs of the device objects we accessed in this chapter, together with the property IDs (EPCs) of the properties we accessed within those device objects. This information does not need to be explicitly specified for most typical uses of **OpenECHO for Processing**, but it is handy information to have for future reference - for example, when debugging **ECHONET Lite** systems via packet monitoring. The names of the **OpenECHO** methods used here were generated automatically by referring to the English version of the ECHONET Lite specification. Of course, as far as the protocol is concerned, only codes such as the EOJ and EPC are actually transmitted.

EOJ for `PowerDistributionBoardMetering`: `[02.87]` (instance code omitted)
  EPC for `OperationStatus`: `80`
  EPC for `MeasuredInstantaneousCurrents`: `C7`

EOJ for temperature sensor: `[00.11]`
  EPC for `MeasuredTemperatureValue`: `E0`

EOJ for humidity sensor: `[00.12]`
  EPC for `MeasuredValueOfRelativeHumidity`: `E0`

Chapter 3 Controlling Other Nodes

In Chapter 2, we issued `get` requests to read information from a power distribution board object. In Chapter 3, we will learn how to use `set` requests to modify the state of other devices. The devices we will consider in this chapter are lights (`GeneralLighting`) and air conditioners (`HomeAirConditioner`).

Sample Programs: Tutorial3a_AllLightsAirconOff
Tutorial3b_AllLightsAirconOff_Individual

Controlling multiple devices at once

First we'll write a program to turn off all lights and air conditioners that are present on the network. For example, it would be convenient to be able to turn off all the lights in our house with a single button as we get ready to leave for the day. Although the **ECHONET Lite** specification describes several different types of device objects for lights, in this case we will use a class named `GeneralLighting`.

As in the previous examples, we begin by using the `Echo.start` method to create a node:

```
Echo.start( new DefaultNodeProfile()
            ,new DeviceObject[]{new DefaultController()});
```

Next we say

```
GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
```

This sends a `SetOperationStatus` request with data value `0x31` to all instances of `GeneralLighting` that are present on the network. The value `0x31` corresponds to "power off."

The calling convention here is similar to that used in the following code snippet, which we have used previously to obtain a list of nodes:

```
NodeProfile.informG().reqInformInstanceListNotification().send();
```

This code for obtaining a list of nodes actually works by sending a request for a list of device objects to all instances of `NodeProfile`. Both this call and the above call to `GeneralLighting.setG()` have the common feature that they issue requests to all entities on the network. More generally, commands of the form

(Device Class).(Request Type)G.req(Request)((Data to send)).req...send();

issue requests to all instances of the device class in question. Replacing "Device Class" with `NodeProfile` yields entirely analogous behavior. Here "Request Type" can be `get, set, or inform`, while "Request" should correspondingly be `get*, set*, or inform*`, where the * denotes a property name such as `OperationStatus`. The "Data to send" field is not needed for `get` or `inform`.

By invoking `NodeProfile.informG()`, we can send an `inform` request to all `NodeProfiles`. If we only wish to receive the response ourselves, we could alternatively use `getG`.

`set` involves a write operation, and thus takes an argument representing the data to be written. In the case of an air conditioner, this might look like the following:

HomeAirConditioner.setG().reqSetOperationStatus(new byte[]{0x31}).send();

Combining all of these techniques, we obtain a program to turn off all devices. The full program looks like this:

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting ;
import com.sonycsl.echo.eoj.device.airconditioner.HomeAirConditioner;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

void setup(){
  try {
     Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
      // turn off all lights
     GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
      // turn off all air conditioners
     HomeAirConditioner.setG().reqSetOperationStatus(new byte[]{0x31}).send();
```

```
    } catch(IOException e){
      e.printStackTrace();
    }
    println("Started") ;
  }
```

Controlling one device at a time

If, instead of turning off all devices, we only wish to turn off one particular device, we can proceed as we did to obtain information from the power distribution board: we create an `EventListener` to execute certain operations whenever a particular node is discovered, then issue a request for a list of nodes on the network.

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting ;
import com.sonycsl.echo.eoj.device.airconditioner.HomeAirConditioner;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

void setup(){
  Echo.addEventListener(new Echo.EventListener() {
    public void onNewGeneralLighting (GeneralLighting device){
     super. onNewGeneralLighting(device);
      println( "General Lighting found.") ;
     // here we add code to address only the particular light we care about
      try {
        device.set().reqSetOperationStatus( new byte[]{0x31}).send() ;
      } catch(IOException e){
        e.printStackTrace();
      }
    }
    public void onNewHomeAirConditioner (HomeAirConditioner device){
      super. onNewHomeAirConditioner (device);
      println( "HomeAirConditioner found.") ;
     // here we add code to address only the particular air conditioner we care about
      try {
        device.set().reqSetOperationStatus( new byte[]{0x31}).send() ;
      } catch(IOException e){
        e.printStackTrace();
      }
    }
  }) ;

  try {
```

```
        Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
        NodeProfile.informG().reqInformInstanceListNotification().send();
    } catch(IOException e){
        e.printStackTrace();
    }
}
```

Note that we do not need to issue multiple repeated `set` requests; instead, responses are received as in the example with `get` above. As with `get`, we need to set up a `Receiver` to receive these responses. In the case of `get`, we overrode the `device.Receiver.onGet`... handler; for `set`, we instead override `device.Receiver.onSet`...

For example, all device objects include the following handler in their `Receiver`:

protected void onSetOperationStatus(EchoObject eoj, short tid, byte esv, EchoProperty property, boolean success)

This is a handler that will be called upon receipt of a response to a previously-issued `reqSetOperationStatus` request. On success, the `success` flag will be set to `true`, and `property` will be empty. On failure, `success` will be `false`, and `property` will contain the unmodified data passed to the set request.

Note: In OpenECHO, responses to `inform` requests are also received by `onGet...` and may not be received by separate `Receiver` handlers.

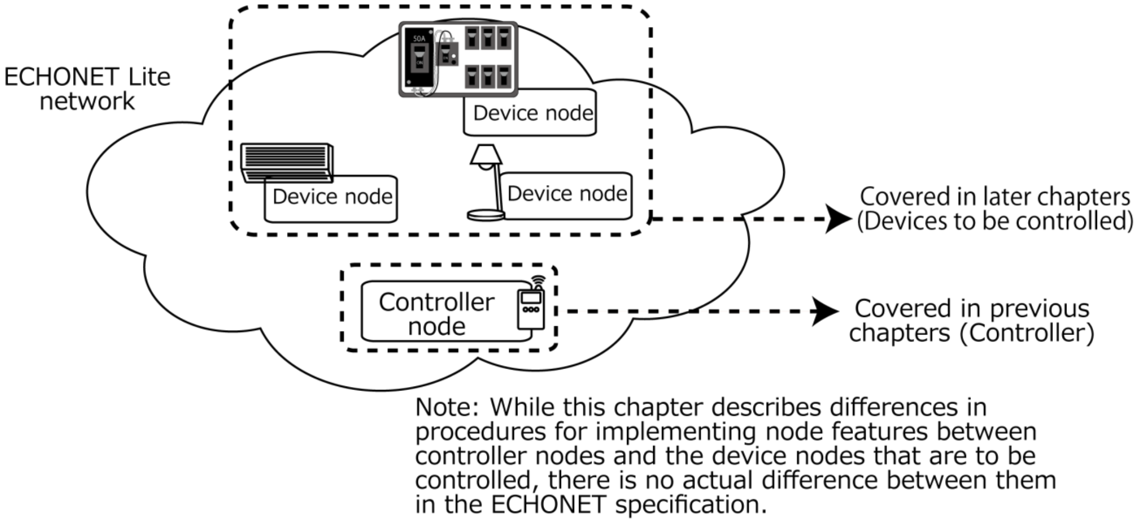
Once again, let's list the EOJs of all device objects we accessed in this chapter, together with the property IDs (EPCs) of all properties we referenced.

EOJ for `GeneralLighting`: `[02.90]`
EPC for `OperationStatus`: `80`

EOJ for `HomeAirConditioner`: `[01.30]`
EPC for `OperationStatus`: `80`

Chapter 4 Implementing a Device Object

The nodes we have created so far have not had the functionality of actual household appliances, but rather the functionality of *controllers* that manage those appliances. In this chapter, we will create a node for a device that might be controlled by one of the controllers discussed previously. We will learn how to allow our new node to be controlled from other nodes on the network.



Once we have learned how to do this, it will be possible to add homemade devices without built-in **ECHONET Lite** support to the network as if they were **ECHONET Lite** compliant. These devices can then be controlled by other nodes. For example, we might imagine building a small fan using a store-bought motor.

Sample programs: Tutorial4_LightEmulator

Device Emulator

First let's create an emulator that will mimic an actual device. To do this, we will create our own new class derived from the device class we will be using. In this case, we will create a class that emulates the `GeneralLighting` class.

For the case of `GeneralLighting`, the methods we will need to override are the following:

- *setOperationStatus*
- *getOperationStatus*
- *setInstallationLocation*
- *getInstallationLocation*
- *getFaultStatus*
- *getManufacturerCode*

These methods are declared `abstract` in the parent class, which means that if we don't override them we will get compiler errors. These are the methods defined as required by **ECHONET Lite**. It might seem that there are a lot of methods to override, but in fact the procedure is rather straightforward. Let's consider a sample implementation.

```
public class LightEmulator extends GeneralLighting {
   byte[] mStatus = {0x31}; // whether the power is on or off. Assumed OFF by default.
   byte[] mLocation = {0x00}; // where the device is located
   byte[] mFaultStatus = {0x42};    // the error code in the event of any problems with the device
   byte[] mManufacturerCode = {0, 0, 0};    // a vendor-specific code

   protected boolean setOperationStatus(byte[] edt) {
      mStatus[0] = edt[0] ;
      // notify other nodes that our power status has changed
      try {
         inform().reqInformOperationStatus().send();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
      return true;
   }

   protected byte[] getOperationStatus() {
      return mStatus;
   }

   protected boolean setInstallationLocation(byte[] edt) {
      mLocation[0] = edt[0];
      try {
         inform().reqInformInstallationLocation().send();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
      return true;
   }
   protected byte[] getInstallationLocation() {
```

```
        return mLocation;
    }

    protected byte[] getFaultStatus() {
        return mFaultStatus;
    }
    protected byte[] getManufacturerCode() {
        return mManufacturerCode;
    }

}
```

Let's briefly discuss the various methods we overrode in this example.

- *setOperationStatus*
- *getOperationStatus*

These are the methods we used previously in Chapters 1 and 2 to write and read the operational status of a device. Methods named `set`* are called when requests are received from other nodes to set values. Methods named `get`* are called when requests are received from other nodes to obtain values.

- *setInstallationLocation*
- *getInstallationLocation*

These methods write and read the location of a device. Because these methods are declared `abstract` in the `DeviceObject` class that serves as the parent class for all devices, they must be overridden by all devices.

- *getFaultStatus*

This function indicates whether or not any fault condition is present. The data values returned by this function indicate the type of fault that is present. This method is also declared `abstract` in the `DeviceObject` parent class.

- *getManufacturerCode*

This function returns the manufacturer code, a unique value assigned to each member of the **ECHONET Lite** consortium. This method is declared `abstract` in the `DeviceObject` parent class.

For details on the return values of these functions, consult the **ECHONET Lite** specification or the **OpenECHO** reference.

There is one point here that requires caution. In this example, we created a device within our own node. However, **we may not directly call routines like `setOperationStatus` and `getOperationStatus` ourselves to get or set data for our own device.** Instead, we must use `set().reqSet*` and `get().reqGet*` to query or control the device, even though it exists within our own node. The reason is that, when methods such as `setOperationStatus` or `getOperationStatus` are used, the device status is modified via methods that differ from the methods expected by **OpenECHO**. This means that the modifications will not be communicated to the **OpenECHO** library, and erroneous behavior may result.

Note that the `set` methods above contain lines like the following:

inform().reqInformOperationStatus().send();

This line notifies other nodes that the operational status of the device object has changed. We use `inform` here because we want this notification to be communicated to the entire network from our device object. The question of whether notifications of status changes should be transmitted to the entire network is addressed in the **ECHONET Lite** specification. The table of properties for each device object in the specification includes a column titled "**Announcement at status change**" For properties that have a circle in this column, status changes should be broadcast using `inform()`.

In this case, we use `inform()` to notify the network of changes in the `OperationStatus` property. If another node created by **OpenECHO** were to receive this notification, the following handler within its `EventListener` would be called:

public void onSetProperty(EchoObject eoj, short tid, byte esv, EchoProperty property, boolean success)

In this chapter, we have only implemented the required methods. For non-required methods, it is not enough merely to override the method. We must additionally register information about our new implementation of the method in the *property map*. The implementation of non-required methods and the procedure for updating the property map is discussed in Chapter 6.

The full program including the class we defined above is listed below. This program changes the window background depending on the operational status of the device (i.e. whether the power is on or off).

```
import java.io.IOException;
import processing.net.*;
import controlP5.*;

import com.sonycsl.echo.Echo;
import com.sonycsl.echo.node.EchoNode;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.DeviceObject;

import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting;

color backgroundLightOnColor = color(255, 204, 0);
color backgroundLightOffColor = color(0, 0, 0);
color backgroundNow = backgroundLightOffColor;

// implementation of a device class for a light
public class LightEmulator extends GeneralLighting {
  byte[] mStatus = {0x31}; // whether the power is on or off. Assumed OFF by default.
  byte[] mLocation = {0x00}; // where the device is located
  byte[] mFaultStatus = {0x42};    // the error code in the event of any problems with the device
  byte[] mManufacturerCode = {0, 0, 0};    // a vendor-specific code

  protected boolean setOperationStatus(byte[] edt) {
    mStatus[0] = edt[0] ;
    // change the background color
    if(mStatus[0] == 0x30){
      backgroundNow = backgroundLightOnColor;
    }else{
      backgroundNow = backgroundLightOffColor;
    }
    // notify other nodes that our power status has changed
    try {
      inform().reqInformOperationStatus().send();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
    return true;
  }

  protected byte[] getOperationStatus() {
    return mStatus;
  }
  protected boolean setInstallationLocation(byte[] edt) {
    mLocation[0] = edt[0];
    try {
```

```
            inform().reqInformInstallationLocation().send();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }
    protected byte[] getInstallationLocation() {
        return mLocation;
    }
    protected byte[] getFaultStatus() {
        return mFaultStatus;
    }
    protected byte[] getManufacturerCode() {
        return mManufacturerCode;
    }

    public String toString() {
        if (mStatus[0] == 0x31) {
            return "Light Emulator(Off)";
        }
        else {
            return "Light Emulator(On)";
        }
    }
}

ControlP5 cp5 ;
LightEmulator light ;
String[] btnStrs = {
    "SWITCH_ON", "SWITCH_OFF"
};

void setup() {
    size(210, (btnStrs.length)*30);
    frameRate(30);

    // next create a user interface for learning and playback
    cp5 = new ControlP5(this) ;
    // display the "Send" button on the left and the "Learn" button on the right
    for ( int bi=0;bi<btnStrs.length;++bi ) {
        cp5.addButton(btnStrs[bi], 0, 0, (bi)*30, 100, 25) ;
    }

    // write a log to System.out
    //Echo.addEventListener( new Echo.Logger(System.out) ) ;

    // we will become a node that contains a LightEmulator
    try {
        light = new LightEmulator() ;
        Echo.start( new DefaultNodeProfile(), new DeviceObject[] {
            light
        }
```

```
        );
    }
    catch( IOException e) {
        e.printStackTrace();
    }
}

void draw() {
    background(backgroundNow);
}

// code to handle button presses
// Note: For ControlP5, the button label is used as the function name
public void SWITCH_ON(int theValue) {
    try {
        light.set().reqSetOperationStatus(new byte[]{0x30}).send();
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}
public void SWITCH_OFF(int theValue) {
    try {
        light.set().reqSetOperationStatus(new byte[]{0x31}).send();
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}
```

Once this code has been executed, programs running on other machines that gather data on other nodes (such as the program we wrote in Chapter 1) should be able to see the `GeneralLighting` device we created here.

Here's a list of the device object EOJs and property IDs (EPCs) that we used in this chapter.

EOJ for `GeneralLighting`: `[02.90]`
- EPC for `OperationStatus`: `80`
- EPC for `InstallationLocation`: `81`
- EPC for `FaultStatus`: `88`

EPC for `ManufacturerCode`: `8A`

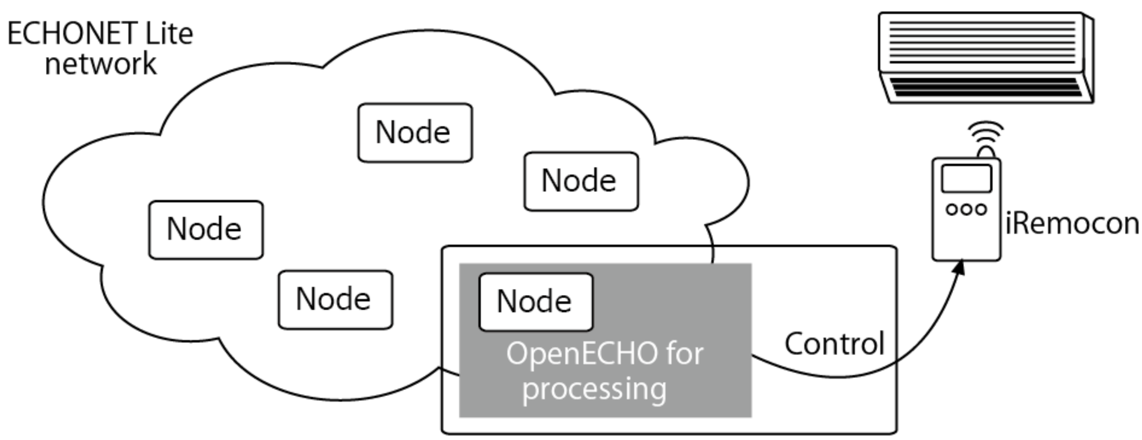
Chapter 5 Creating an actual device object for an infrared remote control

Next let's try controlling an actual device, not an emulation. (Of course, methods for controlling actual hardware devices from **Processing** vary significantly depending on the device hardware implementation. As an interesting challenge, we suggest using **OpenECHO for Processing** to convert a device without built-in **ECHONET Lite** support that you might own - such as an air conditioner - into an **ECHONET Lite** - compliant device.)

Sample programs: Tutorial5a_iRemoconLight
Tutorial5b_iRemoconAircon

The iRemocon : A learning remote control with network-control functionality

As a means of controlling our device, we will use the **iRemocon**, a network-connected infrared learning remote control from Glamo, Inc. However, the methods we describe in this chapter may be used, with slight modifications, to connect a wide variety of existing network appliances to **ECHONET Lite** networks. All that's needed is to replace the **iRemocon** code snippets in the example programs below with code for other communication protocols such as serial communications or ZigBee.



Note: Kanagawa Institute of Technology and Sony Computer Science Laboratories, Inc. have no relationship with **iRemocon** or with Glamo, Inc. Only publicly-available information has been used to create this sample program. Use this program at your own risk; please do not contact us or Glamo with reports of incorrect behavior.

The **iRemocon** infrared learning remote control can be taught to learn the signals emitted by infrared remote controls that come with other appliances. Then, the **iRemocon** can emit those signals itself, thus replacing multiple existing remote controls. This helps to counteract the proliferation of household remote control units.

Most learning remote controls require users to program signal codes by pushing buttons. In contrast, the **iRemocon** offers the significant advantage of being able to receive instructions over the Internet. Moreover, the **iRemocon** can itself be controlled remotely from devices such as smart phones. These features have made the device highly popular.

iRemocon web site: *http://i-remocon.com/ (Japanese only)*

We will use socket communication to control the **iRemocon**, sending commands and receiving the results. Information on controlling the **iRemocon** may be found at the following site:

http://i-remocon.com/development/ (Japanese only)

Infrared remote controls offer *unidirectional* control - that is, data is sent but not received. Thus we can issue commands to turn a device on or off, but we cannot query the current power state of the device. However, **ECHONET Lite** requires that devices receive and respond to status requests. For this reason, here we will remember the last command we sent and use this data to respond to queries. This should not lead to any difficulties when controlling the device from this program alone. However, when this program is used simultaneously with another infrared remote control (for example, the remote control that originally came with the household appliance in question), the device status stored by the program may differ from the status of the actual device.

Let's now describe how the **iRemocon** is controlled from within **Processing**.

We begin by looking up the IP address of the **iRemocon**. For this purpose, you will use publicly available Android or iPhone apps provided by Glamo, Inc.

We will use the following two **iRemocon** commands:

- *ic (Begin learning)*
- *is (Send infrared signal)*

These commands are sent by opening a socket to the **iRemocon** and writing the following byte strings. The port number is 51013.

- **ic;XX;\r\n*
- **is;XX;\r\n*

Here XX is an integer between 1 and 1500 specifying the slot within the **iRemocon** in which the infrared pattern is stored. For example, to command the **iRemocon** to learn an infrared pattern for slot 319, we would write

**ic;319;¥r¥n*

to the socket. Then, to instruct the **iRemocon** to transmit this pattern, we would write

**is;319;¥r¥n*

to the socket.

To perform socket communication in **Processing**, we use the **Client** class within the **Processing** standard library. The above commands then take the form of the following function calls. Here the variable `iRemoconIP` should be set to the IP address as identified using the official **iRemocon** app, which is publicly available from Google Play or the App Store.

```
final String iRemoconIP = "192.168.126.101" ;
final int iRemoconPort = 51013 ;
void iRemoconLearn(int id){
  Client c = new Client(this,iRemoconIP,iRemoconPort) ;
  c.write("*ic;"+id+"\r\n");
```

```
    c.stop() ;
}

void iRemoconSend(int id){
    Client c = new Client(this,iRemoconIP,iRemoconPort) ;
    c.write("*is;"+id+"\r\n");
    c.stop() ;
}
```

Note: the above code will fail if when run on a network with no **iRemocon** devices.

Using these code snippets, we will now create a device class using the **iRemocon**.

First, we will use constants to define the IDs within the **iRemocon** of the infrared patterns we will use. These can be any unused IDs, but for simplicity we will here choose them to be a consecutive block of IDs.

```
final int iBase = 100 ;
final int SWITCH_ON = 0 + iBase ;
final int SWITCH_OFF = 1 + iBase ;
```

As we did in the case of `LightEmulator` above, we will create a class called `iRemoconLight` that derives from `GeneralLighting`:

```
// implementation of a device class for a light
public class iRemoconLight extends GeneralLighting {
    byte[] mStatus = {0x31};// the initial power state is assumed to be OFF
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};

    protected boolean setOperationStatus(byte[] edt) {
        iRemoconSend( edt[0] == 0x30 ? SWITCH_ON : SWITCH_OFF ) ;
        mStatus[0] = edt[0] ;
        // notify other nodes that our power status has changed
        try {
            inform().reqInformOperationStatus().send();
        } catch (IOException e) { e.printStackTrace();}
        return true;
    }
    // when queried for our current power state, we return the last command we transmitted
    protected byte[] getOperationStatus() { return mStatus; }
    protected boolean setInstallationLocation(byte[] edt) {
        mLocation[0] = edt[0];
        try {
            inform().reqInformInstallationLocation().send();
```

```
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        return true;
    }
    protected byte[] getInstallationLocation() {return mLocation;}
    protected byte[] getFaultStatus() {     return mFaultStatus;}
    protected byte[] getManufacturerCode() {return mManufacturerCode;}
}
```

This code snippet is almost identical to the code we wrote above for `LightEmulator`. Among the few differences is that, in the `setOperationStatus` routine, we actually instruct the **iRemocon** to transmit infrared signals.

The full program including this class is listed below. Although the whole program is somewhat long, we list it in full for completeness.

```
import java.io.IOException;
import processing.net.*;
import controlP5.*;

import com.sonycsl.echo.Echo;
import com.sonycsl.echo.node.EchoNode;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.DeviceObject;

import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting;

// change iRemoconIP to the IP address found using the iRemocon app
final String iRemoconIP = "192.168.126.101" ;
final int iRemoconPort = 51013 ;

// define simple constants for the IDs of infrared patterns we use in this program
// the value chosen here for iBase is arbitrary; in practice, you will choose IDs
// that you know are not already used.
final int iBase = 100 ;
final int SWITCH_ON = 0 + iBase ;
final int SWITCH_OFF = 1 + iBase ;
// character strings used to create buttons
String[] btnStrs = {"SWITCH_ON","SWITCH_OFF"} ;


void iRemoconLearn(int id){
    println( "iRemoconLearn : "+btnStrs[id-SWITCH_ON] ) ;
    Client c = new Client(this,iRemoconIP,iRemoconPort) ;
    c.write("*ic;"+id+"\r\n");
    c.stop() ;
}
```

```
void iRemoconSend(int id){
   println( "iRemoconSend : "+btnStrs[id-SWITCH_ON] ) ;
   Client c = new Client(this,iRemoconIP,iRemoconPort) ;
   c.write("*is;"+id+"\r\n");
   c.stop() ;
}

// implementation of a device class for a light
public class iRemoconLight extends GeneralLighting {
   byte[] mStatus = {0x31};// the initial power state is assumed to be OFF
   byte[] mLocation = {0x00};
   byte[] mFaultStatus = {0x42};
   byte[] mManufacturerCode = {0,0,0};

   protected boolean setOperationStatus(byte[] edt) {
      iRemoconSend( edt[0] == 0x30 ? SWITCH_ON : SWITCH_OFF ) ;
      mStatus[0] = edt[0] ;
      //notify other nodes that our power status has changed
      try { inform().reqInformOperationStatus().send(); } catch (IOException e)
         { e.printStackTrace();}
      return true;
   }

   // when queried for our current power state, we return the last command we transmitted
   protected byte[] getOperationStatus() { return mStatus; }
   protected boolean setInstallationLocation(byte[] edt) {
      mLocation[0] = edt[0];
      try {
         inform().reqInformInstallationLocation().send();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
      return true;
   }
   protected byte[] getInstallationLocation() {return mLocation;}
   protected byte[] getFaultStatus() {    return mFaultStatus;}
   protected byte[] getManufacturerCode() {return mManufacturerCode;}
}

ControlP5 cp5 ;
iRemoconLight light ;

void setup(){
   size(210,(btnStrs.length)*30);
   frameRate(30);

   // next create a user interface for learning and playback
   cp5 = new ControlP5(this) ;
   // display the "Send" button on the left and the "Learn" button on the right
   for( int bi=0;bi<btnStrs.length;++bi ){
      cp5.addButton(btnStrs[bi],0,0,(bi)*30,100,25) ;
```

```
        cp5.addButton("LEARN_"+btnStrs[bi],0,110,(bi)*30,100,25) ;
    }

    // write a log to System.out
    // Echo.addEventListener( new Echo.Logger(System.out) ) ;

    try {
        light = new iRemoconLight() ;
        Echo.start( new DefaultNodeProfile(),new DeviceObject[]{light});
    } catch( IOException e){ e.printStackTrace(); }
}

// used only to draw buttons
void draw(){}

// code to handle button presses
// definitions of functions for sending and for learning alternate below
// Note: For ControlP5, the button label is used as the function name.
// light.setOperationStatus(new byte[]{0x30}); cannot do this!!!
public void SWITCH_ON(int theValue){
    try{
        light.set().reqSetOperationStatus(new byte[]{0x30}).send();
    }catch(IOException e){e.printStackTrace();}
}
public void LEARN_SWITCH_ON(int theValue){
    iRemoconLearn(SWITCH_ON) ;
}
public void SWITCH_OFF(int theValue){
    try{
        light.set().reqSetOperationStatus(new byte[]{0x31}).send();
    }catch(IOException e){e.printStackTrace();}
}
public void LEARN_SWITCH_OFF(int theValue){
    iRemoconLearn(SWITCH_OFF) ;
}
```

This program also creates a button that tells the device to learn an infrared pattern. Clicking this button, which will be displayed in your **Processing** session, will cause a light to begin flashing on the **iRemocon**. Point your remote control at the **iRemocon** and press your remote control's buttons to instruct the **iRemocon** to learn your remote control's patterns. Once the **iRemocon** has learned a pattern, it can emit that pattern in response to button presses or in response to requests received from other nodes.

The device class we implemented in this chapter was `GeneralLighting`. For other devices, such as air conditioners, there are more methods that must be implemented, but otherwise the basic procedure remains essentially the same. For an example, see the sample program `Tutorial4b_iRemoconAircon`.

Here's a list of the device object EOJs and property IDs (EPCs) that we used in this chapter.

EOJ for `GeneralLighting`: `[02.90]`

- EPC for `OperationStatus`: `80`
- EPC for `InstallationLocation`: `81`
- EPC for `FaultStatus`: `88`
- EPC for `ManufacturerCode`: `8A`

Chapter 6 Creating a realistic node implementation

In the previous chapters we used the default versions of the node profile and device object classes with no modifications. In this chapter, we will discuss the detailed procedures for implementing these classes. The default versions of these classes are little more than preliminary implementations whose primary purpose is to make this tutorial easy to follow. In particular, the default class implementations cannot guarantee that various requests from other nodes will be correctly answered. We encourage you to master the content of this chapter thoroughly before implementing your **OpenECHO** programs for public release.

Sample programs: Tutorial6a_ImplementRealNode
Tutorial6b_ElectricLock

Node Profiles

As discussed in previous chapters, your node profile is responsible for communicating information about your node to the rest of the network. For this reason, you must implement your own class derived from `NodeProfile` to manage the data needed to describe the node you create.

The following methods are required in any implementation of `NodeProfile`.

- *getManufacturerCode*
  This function returns the *manufacturer code* assigned by the ECHONET Consortium. (The manufacturer code is also known as the maker code.) This is a 3-byte code.

- *getOperatingStatus*
  Returns `0x30` if the power is on. Returns `0x31` if the power is off.

- *getIdentificationNumber*
  Returns the *identification number*, a 17-byte code that uniquely identifies the object within the domain. See the **ECHONET Lite** specification for details on the format of this code.

- *setUniqueIdentifierData*

This function sets the *unique identifier data*, a 2-byte code. See the **ECHONET Lite** specification for details on the format of this code.

- *getUniqueIdentifierData*
  This method responds to incoming requests for the unique identifier data.

A sample implementation is listed below. In general, each method need do nothing more than return the appropriate value.

```
public class MyNodeProfile extends NodeProfile {
   byte[] mManufactureCode = {0,0,0};    // Given by ECHONET Consortium
   byte[] mStatus = {0x30}; // 0x30:ON    0x31:OFF
   byte[] mIdNumber = {(byte)0xFE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   byte[] mUniqueId = {0,0};

   protected byte[] getManufacturerCode() {return mManufactureCode;}
   protected byte[] getOperatingStatus() {    return mStatus;    }
   protected byte[] getIdentificationNumber() {return mIdNumber;}
   protected boolean setUniqueIdentifierData(byte[] edt) {
      if((edt[0] & 0x40) != 0x40)      return false;
      mUniqueId[0] = (byte)((edt[0] & (byte)0x7F) | (mUniqueId[0] & 0x80));
      mUniqueId[1] = edt[1];
      return true;
   }
   protected byte[] getUniqueIdentifierData() {return mUniqueId;}
}
```

Overriding non-required methods

In Chapter 4, we used the example of a `GeneralLighting` class to define our own device object. In that case, we only overrode the required class methods. In this section we will discuss the procedure for implementing optional (non-required) methods. The procedure is the same for both node profiles and device objects.

Properties

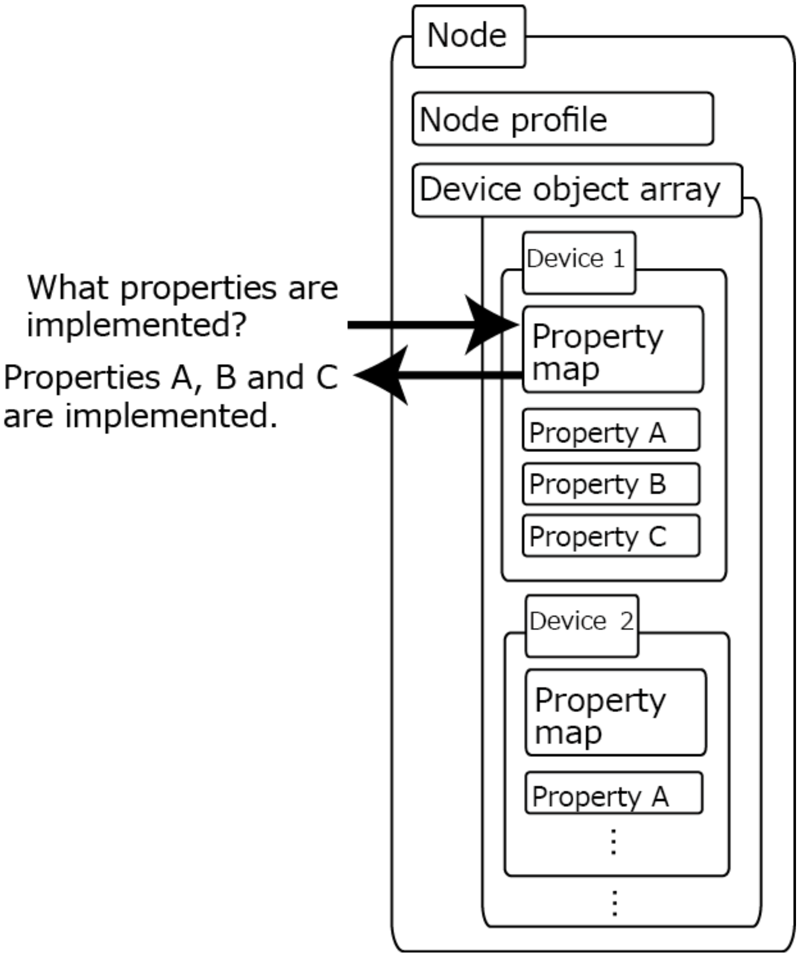
Properties are similar to Java class member variables. They describe the functionality that the device supports. An example of a property is `OperationStatus`. In **OpenECHO**, properties like this are not represented as variables. Instead, all access to these properties - both reads and writes - must use `get` or `set` methods.

Among these `get` and `set` methods, some are *required*, which means that they are declared `abstract` in the parent class. Others are not required, in which case it is entirely up the developer whether or not to implement the method. (This distinction between required and optional methods was discussed briefly in Chapter 4.) To find out whether a method is required or optional, consult the **OpenECHO** reference or the **ECHONET Lite** specification

The Property Map

The *property map* is a table that indicates which of the possible device object properties are implemented by a particular device. Properties that are required must always be implemented, so it is not particularly critical to indicate their presence in the property map. However, for optional properties it is important to indicate the presence of a definition in the property map so that this information can be communicated to other nodes.

In this section we will use an `ElectricLock` class to illustrate the use of the property map. The `ElectricLock` class contains a property named `OccupantNonOccupantStatus` that indicates whether or not any people are inside a locked room. The implementation of this property is not required. If it is implemented, only a `get` method exists.



In the example below, we will assume we have a device that is equipped with some method of detecting whether or not people are present in a room. This may be a homemade device that involves a manual button-press, or an emulator like the one we used in the previous chapter, or an actual device equipped with actual sensors.

Now let's proceed as before to create a derived subclass of `ElectricLock`. Our derived subclass will override the `getOccupantNonOccupantStatus` method. Of course we will also implement all the required methods.

```
public class MyElectricLock extends ElectricLock {
    byte[] mStatus = {0x30};
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};

     // whether or not the lock is currently locked
    byte[] mLockStatus = {0x30};
    // whether or not any people are in the room
    byte[] mOccupantStatus = {0x42};
        //    in practice, instead of storing the above two quantities as variables in this code
        //    it would be better to read and write them as part of the device status, but for
        //    convenience in this sample program we will do it this way.

     // returns whether or not the power is on
    protected byte[] getOperationStatus() {    return mStatus;    }

    // sets the location at which the device is installed
    protected boolean setInstallationLocation(byte[] edt) {
        if(mLocation[0] == edt[0]) return true;
        mLocation[0] = edt[0];
        try {
            inform().reqInformInstallationLocation().send();
        } catch (IOException e) { e.printStackTrace(); }
        return true;
    }
    // returns the location at which the device is installed
    protected byte[] getInstallationLocation() {return mLocation;}
    protected byte[] getFaultStatus() {    return mFaultStatus;}
    protected byte[] getManufacturerCode() {return mManufacturerCode;}
    // setLockSetting is set to 1 here because this device object
    // is capable of controlling multiple locks

    protected boolean setLockSetting1(byte[] edt) {
        mLockStatus = edt;
        return true;
    }
    protected byte[] getLockSetting1() {
        return mLockStatus;
    }
    // override the non-required method for querying whether any people are
    // in the room.
```

```
    // in practice, the body of this function would involve accessing a sensor
    // to determine and return the actual room occupancy status.
    protected byte[] getOccupantNonOccupantStatus() {return mOccupantStatus;}
}
```

Because we implemented a non-required method, our class implementation must also override the `setupPropertyMaps` method and notify **OpenECHO** of the method we overrode. This amounts to configuring the property map. The code to do this reads as follows.

```
protected void setupPropertyMaps() {
    super.setupPropertyMaps();
    addGetProperty(EPC_OCCUPANT_NON_OCCUPANT_STATUS);
}
```

Note that we must first call the method in the parent class. This call creates data for all required properties. Next, we call `addGetProperty` to specify the optional method we overrode. The argument to this function is the ID (known as the EPC) assigned to the property in question. Values for these IDs are defined in each device class as `final` constants with names starting with `EPC_`. Whenever you implement a `get` method, you must call `addGetProperty` with the ID of the property you implemented as the argument. When you implement a `set` method, you must similarly call `addSetProperty`.

The full program including the complete class implementation is listed below.

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.ElectricLock ;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;

public class MyElectricLock extends ElectricLock {
    byte[] mStatus = {0x30};
    byte[] mLocation = {0x00};
    byte[] mFaultStatus = {0x42};
    byte[] mManufacturerCode = {0,0,0};
    byte[] mLockStatus = {0x30};
    byte[] mOccupantStatus = {0x42};

    // we call the add(Get|Set)Property method within the setupPropertyMaps method
    // to ensure that non-required properties we implement are properly registered.
    // (Required properties are registered by super.setupPropertyMaps().)
```

```
    protected void setupPropertyMaps() {
        super.setupPropertyMaps();
        // To register a status change announcement property in the property map, we use
        // addStatusChangeAnnouncementProperty.
        // To register a settable property in the property map, we use addSetProperty.
        // To register a gettable property in the property map, we use addGetProperty.
        addGetProperty(EPC_OCCUPANT_NON_OCCUPANT_STATUS);
    }

protected byte[] getOperationStatus() {    return mStatus;    }
    protected boolean setInstallationLocation(byte[] edt) {
        changeInstallationLocation(edt[0]);
        return true;
    }
    protected byte[] getInstallationLocation() {return mLocation;}
    public void changeInstallationLocation(byte location) {
        if(mLocation[0] == location) return ;
        mLocation[0] = location;
        try {
            inform().reqInformInstallationLocation().send();
        } catch (IOException e) { e.printStackTrace(); }
    }
    protected byte[] getFaultStatus() {    return mFaultStatus;}
    protected byte[] getManufacturerCode() {return mManufacturerCode;}
    protected boolean setLockSetting1(byte[] edt) {
        mLockStatus = edt;
        return true;
    }
    protected byte[] getLockSetting1() {
        return mLockStatus;
    }

    // override the method that returns room occupancy status.
    // In the actual device implementation, this method should actually check whether or
   // not the room is occupied and return an appropriate code.
    protected byte[] getOccupantNonOccupantStatus() {return mOccupantStatus;}

}

void setup() {
    // write a log to System.out
    Echo.addEventListener( new Echo.Logger(System.out) ) ;
    try {
        Echo.start( new DefaultNodeProfile()
                    , new DeviceObject[] { new MyElectricLock() } );
        NodeProfile.informG().reqInformInstanceListNotification().send();
    }
    catch( IOException e) {
        e.printStackTrace();
    }
    println("Started") ;
}
```

Executing this program will not produce anything immediately obvious. However, the device will be now recognized by the controller node as an electric lock, and when queried for the room occupancy status it will return an appropriate response instead of remaining unresponsive.

Here's a list of the device object EOJs and property IDs (EPCs) that we used in this chapter.

EOJ for `ElectricLock: [02.6F]`

- EPC for `OperationStatus`: `80`
- EPC for `InstallationLocation`: `81`
- EPC for `FaultStatus`: `88`
- EPC for `ManufacturerCode`: `8A`
- EPC for `LockSetting1`: `E0`
- EPC for `OccupantNonOccupantStatus`: `E4`

Chapter 7 Putting it all together

In this chapter we will write a program that assembles all the pieces we have learned in this tutorial. This program will execute the following task: When an electric lock on a room is locked, the program will determine whether or not the room is occupied. If the room is not occupied, the program will turn off all room lights.

Sample program: `Tutorial7_AllLightsOff`

An overview of what the program needs to do

The various tasks that our program needs to complete are summarized below.

1. First, there are various settings we will want to configure whenever an electric lock is detected. To this end, we will register an `EventListener` and override its `onNewElectricLock` method. The procedure for doing this was discussed in Chapter 2.

2. When an electric lock is detected, we will establish a `Receiver` for it. Receivers were also discussed in Chapter 2. Setting up this Receiver will allow us to specify various tasks that will be carried out automatically whenever the status of the electric lock changes. More specifically, we will add a handler (`onGetLockSetting1`) to handle events in which the lock is locked or unlocked and to respond to queries regarding the status of the lock. This handler will do the actual work of checking whether or not the room is occupied.

   Responses to room occupancy status queries will also be received by a `Receiver` handler named `onGetOccupantNonOccupantStatus`. If the room is not occupied, this handler will use `GeneralLighting.setG().reqSetOperationStatus` to turn off the lights. This type of function call was discussed in Chapter 3.

   After we have set up these `Receiver`s, we will query the initial locked-or-unlocked status of the electric lock. This type of operation was also discussed in Chapter 3. According to the **ECHONET Lite** specification, implementations of `ElectricLock` are required to send a *status change announcement* whenever the status of the lock changes. Once we have completed our configurations, the appropriate `Receiver` will automatically be called whenever the lock status changes. Status change announcements were discussed in Chapter 4.

3. Finally, we will request a list of devices from all nodes on the network. Think of this as your reward for diligently making your way from Chapter 1 through the end of this tutorial.

   NodeProfile.informG().reqInformInstanceListNotification().send()

Assembling all of the above ingredients, we obtain a program that checks the room occupancy status whenever a new lock status is discovered - in particular, whenever someone locks the lock. If the room is found to be unoccupied, the program turns off all lights.

Source code

The source code for our program is listed below. This program is based on the assumption that somewhere on the network there is a node containing an electric lock. If you don't have an **ECHONET Lite**-compliant electric lock, we suggest you implement the electric-lock program we wrote in Chapter 6, which implements the `OccupantNonOccupantStatus` property, and execute this program on a separate machine.

```
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.EchoProperty;
import com.sonycsl.echo.eoj.EchoObject;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.housingfacilities.ElectricLock ;
import com.sonycsl.echo.eoj.device.housingfacilities.GeneralLighting ;
import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

void setup(){
  // write a log to System.out
```

```
    Echo.addEventListener( new Echo.Logger(System.out) ) ;

    Echo.addEventListener(new Echo.EventListener() {
        public void onNewElectricLock (ElectricLock device){
            println( "ElectricLock sensor found.") ;
            device.setReceiver( new ElectricLock.Receiver(){
                protected void onGetLockSetting1 (EchoObject eoj, short tid, byte esv
                                                , EchoProperty property, boolean success) {
                    super.onGetLockSetting1(eoj, tid, esv, property, success) ;
                    if( !success ){ println( "error in call reqGetLockSetting1" ) ; return ; }
                    if(property.edt[0]==0x42) { println("unlock") ; return ; }

                    // send a message to obtain the room occupancy status when the primary
                    // lock is locked.
                    // note that, because the room occupancy status property is not specified
                    // as a required property by the ECHONET Lite specification, there is no
                    // guarantee that we will be able to obtain this information.
                    try {
                        ((ElectricLock)eoj).get().reqGetOccupantNonOccupantStatus().send();
                    } catch(IOException e){e.printStackTrace();} ;
                }
                protected void onGetOccupantNonOccupantStatus (EchoObject eoj, short tid
                                                , byte esv, EchoProperty property, boolean success) {
                    super.onGetOccupantNonOccupantStatus(eoj, tid, esv, property, success) ;
                    if( !success ){ println( "error in call reqGetOccupantNonOccupantStatus" ) ; return ; }
                    if(property.edt[0]==0x41) { println("occupant") ; return ; }
                    // if the room is unoccupied, send a multicast message to turn off all lights in the
                    // room.
                    try {
                        GeneralLighting.setG().reqSetOperationStatus(new byte[]{0x31}).send();
                    } catch(IOException e){e.printStackTrace();} ;
                }
            }) ;

            // a notification will be issued when the lock status (locked or unlocked) changes,
            // so we only need to send a get request the first time.
            try{
                device.get().reqGetLockSetting1().send() ;
            } catch(IOException e){e.printStackTrace();} ;
        }
    }) ;
try {
        Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
        NodeProfile.informG().reqInformInstanceListNotification().send();
    } catch( IOException e){ e.printStackTrace(); }
    println("Started") ;
}
```

Chapter 8 Adding the WebAPI Interface

After having read the previous chapters, you can now freely control **ECHONET Lite** compatible devices using **OpenECHO for Processing** and have your program join the network as an **ECHONET Lite** device. If your main interest is in the world of Java, you can skip this and the next chapter, since you already have sufficient knowledge to use **OpenECHO**.

In this chapter, let's jump out of the world of Java and into the world of the web. Recently, many end-user applications run in web browsers and they provide various functions by synchronizing with other web services such as online databases and social networking sites. Web applications can easily run in combination with existing web services and currently run on the largest number of platforms. For example, iPhone does not support standard Java, but web applications written in JavaScript can run on iPhone. Considering that most people in modern society acquire information via web browsers, it is natural that each platform attempts to feature a standard web browser that is highly compatible with other platforms.

In such a trend, it is natural for people to want to control household appliances via web browsers. However, functions currently implemented in web browsers do not allow direct access to the **ECHONET Lite** network. As a rule, sockets are used to exchange information on the internet. Sockets that can be directly used from a web browser require a special handshake process called WebSocket and they cannot be used for lower-level UDP socket communication that **ECHONET Lite** adopts. For that reason, we will try to create a mechanism that connects the **ECHONET Lite** network and the web using **OpenECHO for Processing**.

Sample program: Tutorial8_WebAPI

What Protocol Do We Need to Implement?

First of all, let's think about what interface (protocol) is the best for access from web browsers. The best protocol that can be used by web browsers is, without a doubt, HTTP. HTTP allows you to send various types of data such as html, text, image, and video files by switching the attribute value called mime-type. It also lets JavaScript perform dynamic access. Among WebAPIs that can be implemented based on HTTP, we will adopt a WebAPI supporting JSONP in this chapter.

JSONP is a communication method that handles JSON (Java Script Object Notation) objects in the HTTP protocol. To call this API, simply send a request to an HTTP server in the form of a URL and receive a response. This procedure is basically the same as the case where you request data of a web page. However, the response is returned as a JSON object. We also need to modify URL so that it can call a function. By putting a single-byte question mark (?) in a URL, we can treat whatever follows as parameters. For example, let's suppose that the following URL appeared after entering a (Japanese) search string on the top page of Google.

https://www.google.co.jp/search?q=XXXX&...

It indicates that additional information such as the search string is contained after the question mark (?). In other words, the part before the question mark (*https://www.google.co.jp/search*) is the fixed URL, while the subsequent part changes each time and contains search strings and conditions. The following rule is applied when including such information:

key string=value string

The above format is used and can be repeated as many times as you want by separating them with ampersands (&). In the Google case above, the value XXXX is given to the key q. Let's make our API work in the same way. That is, switching functions by adding arguments at the end.

JSONP can also perform cross domain access. For details, see the description written on the **Kadecot** website (*http://kadecot.net/blog/1332/*). To put it all together, cross domain access requires a key such as jsoncallback and callback (differs depending on the web service), and inclusion of a value corresponding to the key in a response returned from the server. For example:

http://jsonp.server.com/?cmd=test&jsoncallback=cb

Let's suppose that the above call was made. Two key strings, namely, cmd and jsoncallback are given to this call. cmd was added as an example of realizing some kind of function that is not specifically defined. Let's look at jsoncallback. This key contains the value cb. If the server attempts to return a JSON object *{"result":"ok"}*, the server actually needs to enclose this JSON object with cb() as shown below.

cb({"result":"ok"})

We won't go into details here. Simply remember that this is necessary to make cross domain possible. For details, see blog posts on kadecot.net and the corresponding Wikipedia article (http://en.wikipedia.org/wiki/JSONP).

Implementing an HTTP Server

From here, we will actually implement a JSONP server. We first need to create an HTTP server since this function is implemented on a higher layer of the HTTP server. This can be done easily because Processing has the *processing.net.Server* class that lets you easily create a socket server. So you can simply add parts corresponding to the HTTP. As far as the basic functions of HTTP are concerned, just adding a specific header at the beginning of the response is sufficient. As the first step, let's create a web page that shows the string "Hello HTTP World!" The port number is 31413.

```
import processing.net.*;

Server myServer ;

void setup(){
  myServer = new Server(this,31413) ;
}

void draw(){
  Client c = myServer.available() ;
  if( c == null || c.available() == 0 ) return ;

  String st = "Hello HTTP World!" ;
  c.write( "HTTP/1.1 200 OK\nConnection: close\nContent-Length: "+st.length()+"\n"
      + "Content-Type: text/plain\n\n" ) ;
  c.write(st) ;
}
```

That's it! Run this program on Processing, and access the following address from the same PC:

http://localhost:31413

The string "Hello HTTP World!" should be displayed. The HTTP header just provides information that is interpreted by browsers and does not appear on the screen.

Turning an HTTP Server into a JSONP Server

Now let's modify the above program to turn it into a JSONP server. Take note of the following three points:

- Get arguments from strings following the question mark (?) in a URL
- Search for the key string jsoncallback and include the value in the response
- Set mime-type to application/json

It is necessary to analyze the HTTP header sent from a browser to get arguments in a URL. There are various HTTP access methods. It is very simple in the case of GET access. For example,

http://localhost:31413/?cmd=AAA&jsoncallback=cb

If you type the above URL in your browser, the browser sends the following string to the web server via a socket.

```
GET /?cmd=AAA&jsoncallback=cb HTTP/1.1
Host: localhost:31413
Connection: keep-alive
Cache-Control: max-age=0
   :
```

As shown above, various attributes separated by line feeds (LF=`0x0A`) are sent, but we only use the line that begins with GET in this case. Separate the line containing GET with a space and look at the second argument to get the requested folder path and URL argument contained at once. In the above example, the string "/?cmd=AAA&jsoncallback=cb", namely the root folder "/", is requested and arguments follow the question mark (?). Extract the part after the question mark (?), separate them with ampersands (&), and then check the left and right of the equal sign (=) to see the key and value of each argument. Analyze the URL this way and store the result in HashMap.

From this Map, find the jsoncallback key and use the string of the value corresponding to the key.

As for Mime-type, replace text/plain in the first example with application/json. The following draw() method is in a program that always returns the JSON object *{"result":"ok"}* regardless of the content of requests.

```
void draw(){
  Client c = myServer.available() ;
  if( c == null || c.available() == 0 ) return ;

  final int lf = 0xa ; //  改行コード
  String getstr = c.readStringUntil(lf) ; //  一行読み込む
  if( getstr == null || !getstr.startsWith("GET") ) return ; // GET の行じゃない場合は処理中断

  String pathall = getstr.split(" ")[1] ; //  空白で区切って２つ目を得る
  String[] args = pathall.substring(pathall.indexOf("?")+1).split("&") ; // ?より後を&で区切る

  HashMap<String,String> m = new HashMap<String,String>() ;   //  引数保存用 HashMap
  for( String term : args ){
    String[] lr = term.split("=");   // =で区切って右辺と左辺を保存
    if( lr.length < 2 ) continue ;
    m.put(lr[0],lr[1]) ;
  }

  String st = m.get("jsoncallback") + "( {\"result\":\"ok\"} )" ; //  返答はいつも ok
  c.write( "HTTP/1.1 200 OK\nConnection: close\nContent-Length: "+st.length()+"\n"
      + "Content-Type: application/json\n\n" ) ;
  c.write(st) ;
}
```

Note: The above code does not use the requested path information. Since the URL is usually encoded when it is sent, we need to add a decoding process to the pathall variable included in the above code. The program does not run properly if there is no jsoncallback in the URL arguments. We also need to add appropriate processes to handle such a case.

Design of the JSONP WebAPI

Now let's think about what API system to build since we have implemented the JSONP server functionality. Let's put ourselves in the shoes of API users. For simple use, devices to be used should be fixed. For example, the minimally required functions of an air conditioner are powering on/off and mode/temperature settings. In terms of **ECHONET Lite**, we need to set the values of EPC that we want to change and set the values after they are modified to implement a sending function. The following example URL calls the function.

http://localhost:31413/?prop=0x80&value=0x30&jsoncallback=cb

The key named prop indicates the epc of the property that we want to change and the key value indicates the (newly setting) value after modification. The calling method lets you change various attributes in a generic way by changing prop.

Implementing WebAPI for Air Conditioners

Now let's implement it. In this example, the target air conditioner is the last one found after a search. For details about this method, see the sample Tutorial3b_AllLightsAirconOff_Individual and the tutorial in Chapter 3. The complete source code is shown below.

```
import processing.net.*;
import com.sonycsl.echo.Echo;
import com.sonycsl.echo.eoj.device.DeviceObject;
import com.sonycsl.echo.eoj.profile.NodeProfile;
import com.sonycsl.echo.eoj.device.airconditioner.HomeAirConditioner;

import com.sonycsl.echo.processing.defaults.DefaultNodeProfile;
import com.sonycsl.echo.processing.defaults.DefaultController;

Server myServer ;
HomeAirConditioner airCond ;

void setup(){

  Echo.addEventListener(new Echo.EventListener() {
    public void onNewHomeAirConditioner (HomeAirConditioner device){
      super.onNewHomeAirConditioner (device);
      println( "HomeAirConditioner found.");
      airCond = device ;
   }
```

```
    });

    try {
      Echo.start( new DefaultNodeProfile(),new DeviceObject[]{new DefaultController()});
      NodeProfile.getG().reqGetSelfNodeInstanceListS().send();
    } catch( IOException e){
      e.printStackTrace();
    }

    //Start JSONP server
    myServer = new Server(this,31413) ;
}

void draw(){
    Client c = myServer.available() ;
    if( c == null || c.available() == 0 ) return ;

    final int lf = 0x0a ;
    String getstr = c.readStringUntil(lf) ;
    if( getstr == null || !getstr.startsWith("GET") ) return ;

    String pathall = getstr.split(" ")[1] ;

    String[] args = pathall.substring(pathall.indexOf("?")+1).split("&") ;

    HashMap<String,String> m = new HashMap<String,String>() ;
    for( String term : args ){
      String[] lr = term.split("=");
      if( lr.length < 2 ) continue ;
      m.put(lr[0],lr[1]) ;
    }

    String result ;

    // If an air-conditioner was found and prop/value are specified, set the value.
    if( airCond != null && m.get("prop")!=null && m.get("value")!=null){
        try {
          //16 進数限定
          airCond.set().reqSetProperty(
                Integer.decode(m.get("prop")).byteValue()
              , new byte[]{Integer.decode(m.get("value")).byteValue()}).send() ;
          result = "Success" ;
        } catch(Exception e){
          e.printStackTrace();
          result = "Error : "+e.toString() ;
        }
    } else result = "Airconditioner not found" ;

    // Returns JSON object. Enclose with jsoncallback()
    String st = m.get("jsoncallback") + "( {¥"result¥":¥""+result+"¥"} )" ;
    c.write( "HTTP/1.1 200 OK¥nConnection: close¥nContent-Length: "+st.length()+"¥n"
        + "Content-Type: application/json¥n¥n" ) ;
    c.write(st) ;
```

}

> In the program above, one calling method that has not been explained in this document appears below.
>
> *airCond.set().reqSetProperty([Property ID] , [Setting Value byte Array]).send()*
>
> `reqSetPropert` in the line above provides arguments with the target property ID as a numeric value without having to specify a property, for example, using the method name such as `reqSetOperationStatus`. It enables any property to perform generic access. The first argument is the value of the property ID and the second argument is an array representing the newly set value as in previous cases. Because we wanted to use a simple program, it can handle only one-byte arrays. If you want to set two-byte or larger values, you need to modify the program.

Run this program on the network on which an air conditioner supporting ECHONET Lite exists and access the following URL.

http://localhost:31413/?prop=0x80&value=0x30&jsoncallback=func

The air conditioner should be powered on and you should receive the following response (if there are multiple air conditioners, the last one found is the target).

func({"result":"Success"})

Note: If you have no compatible device, you can check how it runs on an emulator. Visit *http://kadecot.net/blog/1479/* for details. Due to restrictions on node ID described in Chapter 1, the emulator must run on a PC different from the one on which this program is running.

In the call above, changing the value to 0x31 turns off the power. To set the mode or temperature, give an appropriate value such as 0xb0 and 0xb3 to prop. See Appendix in the **ECHONET Lite** document.

The program above allows your web browser to access the **ECHONET Lite** network and lets you mash up existing web services and household appliances. The next chapter introduces such examples. Since we only made an air conditioner available on the network in this chapter, the next chapter will also use an air conditioner as an example. But it is not at all difficult to support other appliances since you just need to add code to *Echo.EventListener* for appliances you want to use. We encourage you to add code as needed.

Sony Computer Science Laboratories, Inc. (Sony CSL) distributes Android applications called Kadecot and KadecotCore that provide a JSONP API, which features functions to obtain the current status of a device, obtain a list of connected devices, and modify the status of a pre-determined device. For details, see the **Kadecot** website (*http://kadecot.net/blog/1633/*).

Risks of JSONP API

To conclude this chapter, we will discuss the risks of the JSONP API. The JSONP server we created in this chapter supports cross domain access, and thus, it can be accessed from any website on the internet. Also, if a PC or smartphone is infected with a virus, it is easy for the virus to access the server. Such malicious programs might access the server and perform unintended operations. If you extend this JSONP server or if the server becomes able to obtain information using the aforementioned kadecot/kadecotCore, the usage status of the air conditioner, lighting, and electricity in your house may be hacked.

You may wonder what the use of such information is, but usage information of household appliances reveals one's life, a kind of life log. In particular, whether or not someone is in a house is very useful information for criminals. For example, if lights are turned off at night and no air condition is running on a hot day, that information indicates that no one is in the house and provides opportunities for thieves. On the other hand, if a stalker or scammer knows a woman living on her own is in her house, it gives them perfect opportunities to visit. Moreover, if a device that causes heat is controlled via the network, someone might abuse it and cause a fire.

Needless to say, it is not possible to access the JSONP server without knowing the IP address. The types of IP address normally used are limited. For example, the IP address of the JSONP server used in this chapter can easily be identified by scanning all addresses.

The JSONP server created in this chapter is open and thus includes the above risks. Use this program only for evaluation purposes and do not leave it running on your computer. If the server denies access based on the socket origin information, the mechanism increases safety, although it is not perfect. Changing the port number has little effect, since it is meaningless if all ports are scanned.

There is active discussion concerning the security of household appliance control from the internet. There is no measure that ensures perfect security. Some even argue that no substantial security is taken into consideration for **ECHONET Lite**, and that it is too open and impractical. There are many opinions and the author of this document does not necessarily support such opinions. However, we would recommend that you at least understand that JSONP API increases the vulnerability of **ECHONET Lite** and requires careful use.

Chapter 9 Sample Application Using WebAPI

The JSON API implemented in the previous chapter was very simple. However, accessing household appliances via a browser creates immeasurable possibilities. In this chapter, we will create some web applications using the API introduced in the previous chapter. We basically use JavaScript because access will be made from a browser. Because html and JavaScript have their own syntaxes and because we have no time to explain everything from the fundamentals, we assume that you already have basic knowledge of HTML. If you don't know anything about HTML, there are numerous articles on it. Read them before reading this chapter.

We will create 3 applications in this chapter. First, we will create a web remote controller application that provides simple buttons, but we will make it a little unique by placing the remote controller in a blog as a blog component. Second, we will create a remote controller that works in sync with external temperature. The air conditioner automatically powers on when the external temperature is higher or lower than the threshold values. As a third application, we will create a program that lets you click on any position in the world map and use the temperature of the clicked place to set the air conditioner. You may feel like you are on an overseas trip.

JavaScript has a few dialects depending on the browser. There are libraries to hide such parts and libraries to them easier to use. In this chapter, we will use one of the more popular libraries, jQuery, to write samples. jQuery is distributed through *http://jquery.com/* under the MIT license. In this chapter, most JSONP-specific methods are rarely used, except for `$.getJSON()`.

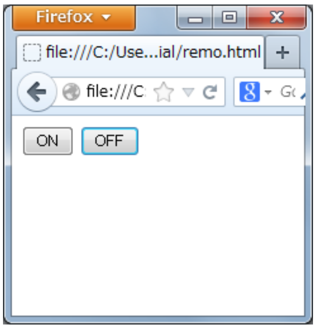
Sample 1 Creating a Remote Controller Blog Component

First of all, let's create a page with simply-arranged buttons that allow you to control an air conditioner. The source code is as shown below. It is very simple.

```
<html>
  <head>
    <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
    <script type="text/javascript">
        function ctrl(epc,edt){
             $.getJSON("http://localhost:31413/?prop="+epc+"&value="+edt+"&jsoncallback=?");
```

```
            }
        </script>
    </head>
    <body>
            <input type="button" value="ON" onclick="ctrl(0x80,0x30)"></input>
            <input type="button" value="OFF" onclick="ctrl(0x80,0x31)"></input>
    </body>
</html>
```

When you save this file as remo.html and open it on a browser, the following screen opens (we have checked the behavior on Firefox).



Run the JSONP server created in the previous chapter on the same PC and click the ON or OFF button. You should be able to power on or off the air conditioner this way.

Now let's move on to the explanation of the code. Firstly look at the <script> tag in <head>.

```
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
```

This tag loads the jquery library from the jquery official site. Although the above URL exists as of Dec., 2013, it is probably a good idea to download the jquery library and place it on your own server because the URL may change in the future and burden the jquery server.

The second <script> tag defines the function ctrl() used to access an air conditioner.

```
<script type="text/javascript">
   function ctrl(epc,edt){
      $.getJSON("http://localhost:31413/?prop="+epc+"&value="+edt+"&jsoncallback=?") ;
   }
</script>
```

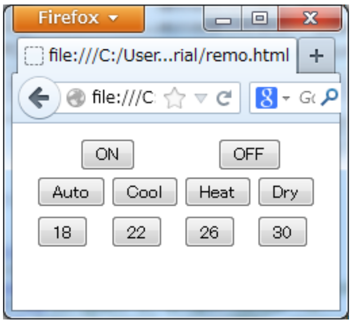
Because $.getJSON requires the access destination URL as the first argument, set the URL of the JSONP server, and add the property ID epc in hex and the setting value edt. ”*jsoncallback=?*” at the end is a predefined measure to call JSONP with jQuery. The actual access URL is placed at the position of the question mark (?) as an appropriate name that does not conflict with other names.

Two buttons are created in the <body> tag and the ctrl() method is called by the (onclick) handler when they are clicked. We have omitted details.

Let’s add mode and temperature settings in addition to the power setting. Simply add input tags in <body>. An example is shown below. They are placed in a table so that their positions can be aligned.

```
<body>
<table align="center">
<tr><td colspan="2" align="center">
<input type="button" value="ON"      onclick="ctrl(0x80,0x30)"></input>
</td><td colspan="2" align="center">
<input type="button" value="OFF"    onclick="ctrl(0x80,0x31)"></input>
</td></tr>
<tr>
<td><input type="button" value="Auto" onclick="ctrl(0xb0,0x41)"></input></td>
<td><input type="button" value="Cool" onclick="ctrl(0xb0,0x42)"></input></td>
<td><input type="button" value="Heat" onclick="ctrl(0xb0,0x43)"></input></td>
<td><input type="button" value="Dry"    onclick="ctrl(0xb0,0x44)"></input></td>
</tr>
<tr></tr>
<tr>
<td><input type="button" value="18" onclick="ctrl(0xb3,18)"></input></td>
<td><input type="button" value="22" onclick="ctrl(0xb3,22)"></input></td>
<td><input type="button" value="26" onclick="ctrl(0xb3,26)"></input></td>
<td><input type="button" value="30" onclick="ctrl(0xb3,30)"></input></td>
</tr>
</table>
</body>
```

It appears as shown below when opened with a browser. Click any button to check if the operation of the air condition actually changes.



Then simply embed it in any blog post. How it's embedded depends on the blog, but procedure is basically the same. Insert jQuery loading and the definition of the ctrl() function into a file that defines the entire theme of the blog (where <head> is located).

```
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
<script type="text/javascript">
  function ctrl(epc,edt){
          $.getJSON("http://localhost:31413/?"
                  +"prop=0x"+epc.toString(16)+"&value=0x"+edt.toString(16)
                  +"&jsoncallback=?") ;
  }
</script>
```

Next, insert the part enclosed by table tags in the blog component display area to show the buttons.

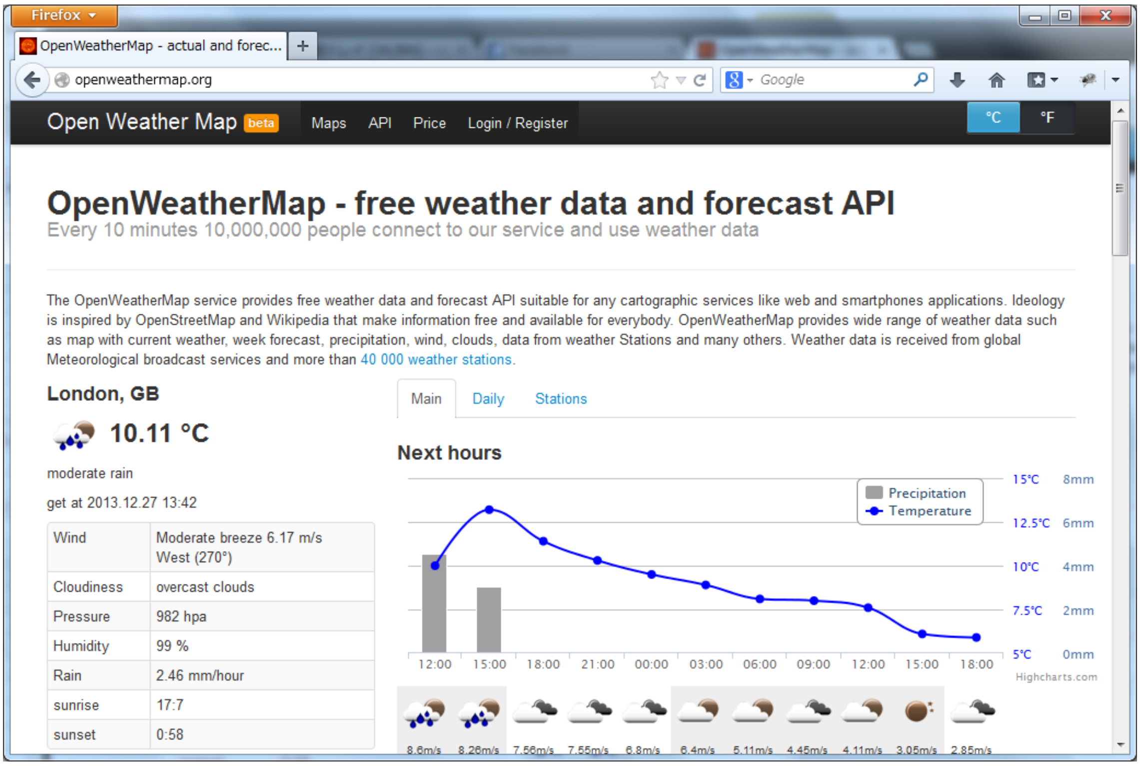
```
<table align="center">
<tr><td colspan="2" align="center">
<input type="button" value="ON"    onclick="ctrl(0x80,0x30)"></input>
</td><td colspan="2" align="center">
<input type="button" value="OFF"   onclick="ctrl(0x80,0x31)"></input>
</td></tr>
<tr>
<td><input type="button" value="Auto" onclick="ctrl(0xb0,0x41)"></input></td>
<td><input type="button" value="Cool" onclick="ctrl(0xb0,0x42)"></input></td>
<td><input type="button" value="Heat" onclick="ctrl(0xb0,0x43)"></input></td>
<td><input type="button" value="Dry"   onclick="ctrl(0xb0,0x44)"></input></td>
</tr>
<tr></tr>
<tr>
<td><input type="button" value="18" onclick="ctrl(0xb3,18)"></input></td>
<td><input type="button" value="22" onclick="ctrl(0xb3,22)"></input></td>
<td><input type="button" value="26" onclick="ctrl(0xb3,26)"></input></td>
<td><input type="button" value="30" onclick="ctrl(0xb3,30)"></input></td>
</tr>
</table>
```

Some blogs allow you to efficiently arrange html code as a plugin in a blog post. For example, major blog site FC2 provides a Free Area in their Official Plug-in. You just copy and paste the <table> tag part as setting data of the plugin to turn it into a blog component. An actual example is shown below.

Because no meticulous design was performed, the UI looks very simple, but you should be able to control the air conditioner in your room just by pressing these buttons in your blog post. It is not usual to keep your own blog open, but we believe you understand how a remote controller function can be added to an existing web service.

Sample 2 Synchronization with External Temperature

Now let's create an application that automatically powers the air conditioner ON/OFF according to the external temperature value obtained from a web service. We will use the `OpenWeatherMap` service to get the external temperature data. `OpenWeatherMap` is a very useful web service that can be used without registration. You can get weather information of a place you want to know just by throwing a JSONP request containing the city name or latitude/longitude data. Although information is not updated very frequently and may differ from present conditions, it is sufficient as a WebAPI service. We will use this service in this chapter.



OpenWeatherMap (http://openweathermap.org/)

Now let's try to acquire the weather data of Tokyo as a test. Enter the following URL in your browser.

http://api.openweathermap.org/data/2.5/weather?q=Tokyo,jp&callback=func

Note: This service uses "callback" instead of "jsoncallback".

You will receive the following response.

func({"coord":{"lon":139.69,"lat":35.69}
,"sys":{"message":0.11,"country":"JP","sunrise":1388094582,"sunset":1388129709}
,"weather":[{"id":500,"main":"Rain","description":"light rain","icon":"10d"}]
,"base":"gdps stations"
,"main":{"temp":281.15,"pressure":998,"humidity":81,"temp_min":281.15,"temp_max":281.15}
,"wind":{"speed":4.6,"deg":50}
,"rain":{"3h":0.5}
,"clouds":{"all":75}
,"dt":1388113200

```
,"id":1850147
,"name":"Tokyo"
,"cod":200
})
```

You see that the absolute temperature is returned as the main.temp property. If you examine it carefully, you can also see coord contains other information such as latitude/longitude, the time of sunrise and sunset, atmospheric pressure, and humidity in addition to the temperature. We can build various services based on the information, but for the time being, let's just use the temperature.

The following text shows the entire code of a program that controls an air conditioner by obtaining the temperature of Tokyo acquired from `OpenWeatherMap`.

```
<html>
<head>
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
<script type="text/javascript">

function ctrl(epc,edt){
	$.getJSON("http://localhost:31413/?prop="+epc+"&value="+edt+"&jsoncallback=?") ;
}

onload = function(){
	$.getJSON("http://api.openweathermap.org/data/2.5/weather?q=Tokyo,jp&callback=?"
			,function( rep ){
				$(document.body).append( (rep.main.temp-273.15) + ' degree' ) ;
				if( rep.main.temp-273.15 < 18 ){
					ctrl( 0x80,0x30 ) ;	// Power on
					ctrl( 0xb0,0x43 ) ;	// Heat mode
					ctrl( 0xb3,18 ) ;	// 18 degree
				} else if( rep.main.temp-273.15 > 28 ){
					ctrl( 0x80,0x30 ) ;	// Power on
					ctrl( 0xb0,0x42 ) ;	// Cool mode
					ctrl( 0xb3,28 ) ;	// 28 degree
				} else {
					ctrl( 0x80,0x31 ) ;	// Power off
				}
			}
	) ;
} ;

</script>
</head>
<body>
</body>
</html>
```

The code is surprisingly short. The ctrl function works in the same way as in the case of the blog component. The onload method is newly used in this code and is executed when the page is loaded in the browser.

The first `$.getJSON()` call is a call to `OpenWeatherMap`. Only 1 argument was given to `$.getJSON()` when used in the blog component, because the program ignored the response during device operation. We can actually receive a JSON object obtained from the JSONP server in response as the second argument of `$.getJSON()`. This argument is a callback function that receives a JSON object as an argument. It is function(rep){ … } in the example above. The response is returned from the JSONP server to this rep.

As mentioned before, the main.temp property in the response from `OpenWeatherMap` contains the absolute temperature. Read rep.main.temp in the callback function above to get this value.

$(document.body).append((rep.main.temp-273.15) + ' degree') ;

This command displays the temperature converted to Celsius in the page.

Next, we add code that turns on the heater if this value is lower than 18℃ or cooler if this value is higher than 28℃. We will not explain details here.

```
if( rep.main.temp-273.15 < 18 ){
        ctrl( 0x80,0x30 ) ;   // Power on
        ctrl( 0xb0,0x43 ) ;   // Heat mode
        ctrl( 0xb3,18 ) ;     // 18 degree
} else if( rep.main.temp-273.15 > 28 ){
        ctrl( 0x80,0x30 ) ;   // Power on
        ctrl( 0xb0,0x42 ) ;   // Cool mode
        ctrl( 0xb3,28 ) ;     // 28 degree
} else {
        ctrl( 0x80,0x31 ) ;   // Power off
}
```

This simple code automatically powers on the air conditioner when the external temperature is high or low and powers off the air conditioner when the temperature is normal. The code above checks the external temperature only when the page is opened. If the code periodically checks the temperature using `setInterval`(), it will be probably more useful. Because the JSON server currently has no method to get the current value, it attempts to power on the air conditioner even when it is already turned on. There are no practical problems, but you can probably reduce unnecessary processes by strengthening the server functionality.

Sample 3 Synchronizing with Google Maps API

The last example in this chapter may not be very practical, but it is different from the previous ones. This program lets you travel around the world using the Google Maps API and `OpenWeatherMap` API. Of course, it is just a virtual trip. This application lets you experience the temperature of any place in the world with a single click.

It is very simple. The world map is displayed and the user just clicks on a place that the user wants to experience. Then you get the latitude and longitude. Send the data to `OpenWeatherMap` to get the temperature of the place. And then set the air conditioner in your house according to the received temperature.

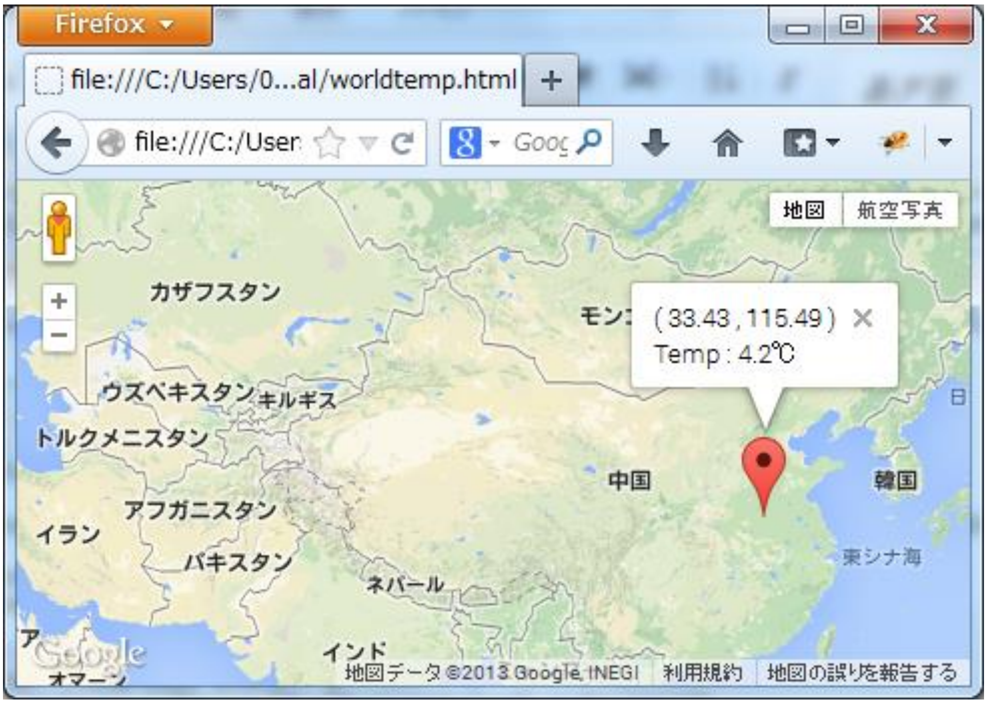
We will not explain details about how to use the Google Maps API since Google provides extensive information on it. The following code provides this function.

```
<html>
<head>
<style type="text/css">
  html { height: 100% }
  body { height: 100%; margin: 0; padding: 0 }
  #map_canvas { height: 100% }
</style>
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?key=%MAPS_API_KEY%&sensor=false">
</script>
<script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
<script type="text/javascript">

function ctrl(epc,edt){
     $.getJSON("http://localhost:31413/?prop="+epc+"&value="+edt+"&jsoncallback=?") ;
}

onload = function(){
  ctrl( 0x80,0x30 ) ;        // Power on
  ctrl( 0xb0,0x41 ) ;        // Auto mode

  var mapOptions = {
     center: new google.maps.LatLng(35.681004,139.767162),
     zoom: 3,
     mapTypeId: google.maps.MapTypeId.ROADMAP
  };
  var map =
     new google.maps.Map(document.getElementById("map_canvas"),mapOptions);

  var marker = null , infoWin = null ;
  google.maps.event.addListener(map, 'click', function(e) {
     if( marker === null )
       marker = new google.maps.Marker({ position: e.latLng, map: map });
     else
```

```
            marker.setPosition( e.latLng ) ;

         if( infoWin !== null ) infoWin.close() ;

         // Access OpenWeatherMap
         $.getJSON( 'http://api.openweathermap.org/data/2.5/weather?lat='
               +e.latLng.lat()+'&lon='+e.latLng.lng()+'&callback=?'
            ,function(r){ // Weather obtained.
                     r.main.temp = (r.main.temp - 273.15).toFixed(1) ;    // To integer

                     // Open information window
                     var ct = '( '+r.coord.lat+' , '+r.coord.lon+' )<br>'
                                 + 'Temp : '+r.main.temp+'℃' ;
                     if( infoWin === null )
                        infoWin = new google.maps.InfoWindow({content: ct})
                     else
                        infoWin.setContent(ct) ;

                     infoWin.open( map,marker ) ;

                     // Set the aircon temp
                     var rt = Math.round( r.main.temp ) ;
                     ctrl( 0xb3, (rt<0?0:(rt > 30 ? 30 : rt)) ) ;

         }) ; // OpenWeatherMap handler
      }); // Google Maps click handler
   } ;

   </script>
   </head>
   <body>
      <div id="map_canvas" style="width:100%; height:100%"></div>
   </body>
   </html>
```

A string called API key is necessary to use the Google Maps API. Obtain this string from Google's website yourself.

After obtaining the string, replace %MAPS_API_KEY% located at the upper part of the html source above with the API key you received from Google and open the source with your browser. Google Maps fills the whole screen. At the same time, the air conditioner is powered on and goes into AUTO mode (we have not checked the behavior of an air conditioner that doesn't have AUTO mode). Clicking on any position in the map will display the latitude, longitude, and current temperature in a balloon and the air conditioner's temperature setting changes accordingly. Since ECHONET Lite can only use integers as the temperature, the temperature is rounded down to 4 degree if the temperature is 4.2 degree. If the temperature is below 0 degree or over 30 degree, it is fixed at the limit.



Let's see the source code.

The style sheet at the top is for extending the map to the entire screen.

```
<style type="text/css">
   html { height: 100% }
   body { height: 100%; margin: 0; padding: 0 }
   #map_canvas { height: 100% }
</style>
```

The <script> tag that follows is for using the Google Maps API.

```
<script type="text/javascript"
src="http://maps.googleapis.com/maps/api/js?key=%MAPS_API_KEY%&sensor=false">
</script>
```

The <script> tag that follows reads jquery as with the case of the previous sample. The first ctrl() function in the script is for controlling the air conditioner.

In the onload handler, the air conditioner is powered on and then set to Auto mode.

```
ctrl( 0x80,0x30 );        // Power on
ctrl( 0xb0,0x41 );        // Auto mode
```

The next part sets the initial status of Google Maps and creates a Map object.

```
var mapOptions = {
    center: new google.maps.LatLng(35.681004,139.767162),
    zoom: 3,
    mapTypeId: google.maps.MapTypeId.ROADMAP
};
var map =
    new google.maps.Map(
document.getElementById("map_canvas"),mapOptions);
```

Next is the main logic of this program. It is a handler definition used when a point in the map is clicked.

google.maps.event.addListener(map, 'click', function(e) { …

This callback displays the marker icon at the clicked position. Because the latitude and longitude of the clicked position is contained in the e.latLng object, create a Marker object using it as an argument. If the marker is already displayed, it changes the position only.

```
if( marker === null )
   marker = new google.maps.Marker({ position: e.latLng, map: map });
else
   marker.setPosition( e.latLng ) ;
```

Mar

Call OpenWeatherMap to check the weather of the clicked position.

```
$.getJSON( 'http://api.openweathermap.org/data/2.5/weather?lat='
          +e.latLng.lat()+'&lon='+e.latLng.lng()+'&callback=?'
        ,function(r){ … } ) ;
```

As with the previous case, the weather is called with `getJSON` and the result is received using the callback function. The position was specified using the city name Tokyo,jp in the previous example, but here, it is specified using the latitude and longitude.

When the value returns, display the value in the balloon (`InfoWindow`) and change the air conditioner’s temperature setting according to the value.

```
var rt = Math.round( r.main.temp ) ;
ctrl( 0xb3, (rt<0?0:(rt > 30 ? 30 : rt)) ) ;
```

As previously mentioned, temperature is always an integer in **ECHONET Lite**. The value is rounded off. Because normal air conditioners cannot handle abnormal temperatures below 0 or over 30 degree Celsius, such values are fixed at the highest or lowest limit.

The explanation was rather long, but we did not mention anything extraordinary. The originally planned function is realized using a normal Google Maps API call and access to OpenWeatherMap.

Conclusion on WebAPI

In Chapters 8 and 9, we did not describe direct operation of **ECHONET Lite** using Java, but rather looked at examples of applications that link remote controller operation with the web. We aren't sure how practical the examples shown in these chapters can be, but hopefully you understand that various applications can be created as a result of the fusion between web services and household appliance operation.

Although some risks may arise when synchronizing household appliances with the network, new services may also be born from that synchronization. We hope applications for household appliances will thrive in the future while improving their security.

Afterword

This completes our tutorial. **ECHONET Lite** is a young protocol that only came into existence quite recently. Thus it is entirely possible that the specification will change in the future. Moreover, as of the beginning of 2013 there are not yet many **ECHONET Lite** - compliant devices. However, in the past there was no open protocol that supported such a large number of household appliances and sensors; now, with more and more companies participating in the **ECHONET Lite** world, the protocol offers unmistakable future potential. Today, a developer who wishes to create services that use home appliance networks needs only to combine **ECHONET Lite** with a communications layer. This will suffice to ensure that the developer's services enjoy a seamless transition to the increasing number of compliant devices expected to become available in the future. We sincerely hope that **OpenECHO for Processing** can play some role in furthering the widespread adoption of **ECHONET Lite**.

The **OpenECHO** software remains under active development. In the future, we hope to offer users a variety of improvements and modified specifications. The most recent version of the software may be found at GitHub, and we hope you will access our latest releases at that location. **OpenECHO for Processing** and this tutorial will henceforth be included in the **OpenECHO** distribution, and we plan to maintain these components in parallel with the development of the base **OpenECHO** project.

OpenECHO distribution site: *https://github.com/SonyCSL/OpenECHO/*

If you discover any bugs - or anything else that could use improvement - in **OpenECHO**, **OpenECHO for Processing**, or this tutorial, please do not hesitate to notify Sony Computer Science Laboratories: `info@kadecot.net`.

Note: Names of companies and products that appear in this document are trademarks or registered trademarks of the companies or organizations in question.