CHAPTER 9

Home Automation and Dynamic Web

This chapter discusses the implementation of a simple home automation system that transforms Intel Galileo into a web server with a dynamic web application. That application can be accessed using Internet browsers on personal computers and mobile phones, including Android and iPhone devices.

The system provides a web page that changes dynamically. It can show a temperature sensor reading, detect intruders with the PIR sensor, arm and disarm the system using a flexible keypad, and send commands to control devices in your home, like turning on and off lamps and TVs.

Each peripheral used in this chapter contains individual test code and schematics, so even if you are not interested in assembling the whole home automation system, you can use the four "micro-projects" that describe how to integrate a keypad, a temperature sensor, switch module relays, and PIR sensors individually.

Project Details

If you read Chapter 3, you will remember that SPI images are very small and do not contain all device drivers and software packages compared to SD card images. The sketches are not persisted.

The first requirement of this project is related to the firmware that you must add to the SD card image. This is because node.js is used to implement a web server running directly on Intel Galileo and the sketch must persist in the board.

A second requirement of this project is that you need to have Intel Galileo connected to some network using an Ethernet cable, with a WiFi module or LTE modem like the XMM7160. To reach Intel Galileo with your browser, you need to know its IP address as well, which means you need to have the serial cable open to the serial console.

The house in this case might be controlled using a web page that changes dynamically with new updates. This web page is dynamic because it will automatically be updated with sensor readings.

But why wasn't a desktop or mobile phone application created for this project? The dynamic application using a web page enables the program to run in most devices, including personal computers, tablets, laptops, and mobile phones.

The main component on this project is a web server that's implemented using node.js. If you do not know what node.js is, it is a very nice platform that provides a very simple way to quickly implement scalable applications like web servers. Node.js is based on the JavaScript language and doesn't require heavy software like Apache, which means it's good with Intel Galileo. For more details regarding node.js, visit `nodejs.org`.

This project controls two switch relays, receives events from a passive infrared (PIR) sensor responsible for motion detection, and monitors the temperature.

The Software Architecture

The architecture is simple and will help you understand the code. There are three components in this project: the web server, the web application, and the sketch.

The web server and the sketches run in Intel Galileo.

The web application runs in most browsers, including Chrome, Internet Explorer, Firefox, Safari, and others available on desktops, tables, phones, and laptops. Figure 9-1 shows this interaction.

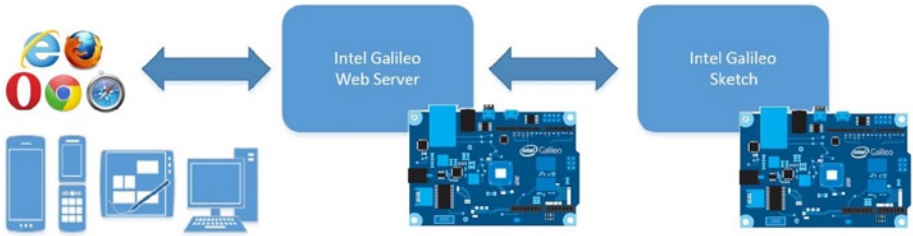

Figure 9-1. *The interaction with the web server component*

The Web Server

In this project, the web server acts as an intermediary component that communicates with two elements: the sketches and the browsers. Keep in mind that the web application can send commands and control the house when the user interacts with the user interface in HTML and the web page can receive events from Intel Galileo that report the status of the sensors installed in the house dynamically and asynchronously.

In both cases, the web server converts the messages sent by the web page to Intel Galileo and vice versa and the channels for the message exchange are implemented using sockets.

Figure 9-2 illustrates how communication works across all the software components.

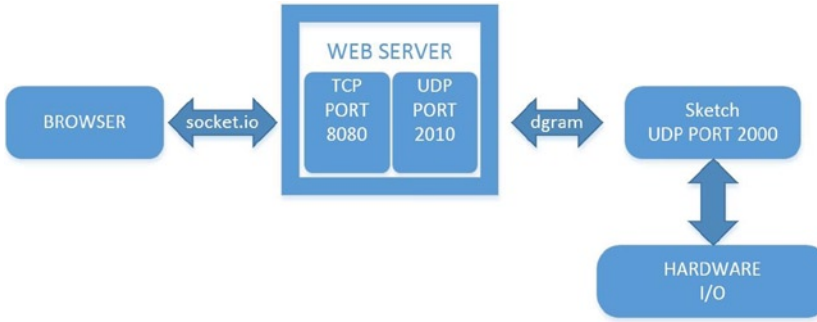

Figure 9-2. *Communication between the web server, browser, and skecthes*

The communication between Intel Galileo sketch and the web server is done using datagrams (UDP). For this function, the sketches keep a UDP server listening on port 2000 while the web server keeps a UDP server listening on port 2010. The communication between the web server and the browser uses socket.io, which essentially indicates a TCP connection.

Thus, when it is necessary to inform the sensors reading, UDP messages are sent by the sketch to port 2010 received by the web server. The web server is then responsible for transforming this message to an event transmitted to the browsers connected to the web server using port 8080. The browser receives the message through socket.io client pieces and the specific elements of the page are updated dynamically, thus avoiding having to refresh the whole page.

On the other hand, the user might send commands to control devices to turn on lamps, TVs, or any other AC device. In this case, the client socket.io in the web application sends a message to the web server using the socket.io function on port 8080 to the web server. The web server catches this message and converts it to a UDP message that is transmitted to port 2000. The UDP server running in the sketch and listening to port 2000 receives the message and interprets it by managing the Arduino's digital I/O headers in order to control devices through switch relays.

The web server and the sketch run in the same version of Intel Galileo, so you might wonder why you have to establish a connection using UDP instead of a simpler alternative, such as using signals or writing/reading operations in the file system. Remember that node.js is responsible for ensuring that the web server runs in a different instance than the sketch. That means the code is platform-dependent. If you decide to change the web server, the communication layer must be rewritten.

Regarding the communication with the web server and the browser, the choice regarding can be implemented with a few lines of code. As a result, you have a web page that changes dynamically and can avoid the annoyance of having to refresh the whole page like when you use `<meta http-equiv="refresh" content="number of seconds" >` in the HTML headers.

Materials List

This project does not require that much material and the sensors used are very affordable (see Table 9-1). You can build this system with around US $25 if you use an Ethernet cable and around US$ 40 if you need to buy the mPCIe WiFi card with the antennas (besides the Intel Galileo cost).

Table 9-1. *Optional Materials*

Quantity	Components
1	12-key membrane keypad
2	300 ohm 1/4 W resistor
1	LED green
1	LED red
1	HC-SR501 PIR sensor module
1	YwRobot module, two relays
1	TMP36 temperature sensor
1	0.1 uF ceramic
1	Intel Centrino Wireless-N 135 mPCIe or Ethernet
1	Breadboard (400 points are enough)
N (Several)	Dupont wires, female to male, and wire jumpers

In fact, you do not need to buy all the items if you do not want to have specific features like the movement detection provided by the PIR sensor.

Integrating the Components Individually

The home automation system contains software and hardware components and it is a good idea to test them individually in order to understand how each component works and how to write the code that interacts with each one. The next sections help you to test the keypad, the PIR sensor, the YwRobot relay module, and the TMP36 temperature sensor, all before you join all the pieces.

Testing the Keypad

The flexible membrane keypad used in this project has 12 keys, operates at 12V, is 76x60x0.8mm, and is very easy to integrate. However, you need only to know how the connector is mapped to the keys and have a simple way to write the code (see Figure 9-3).

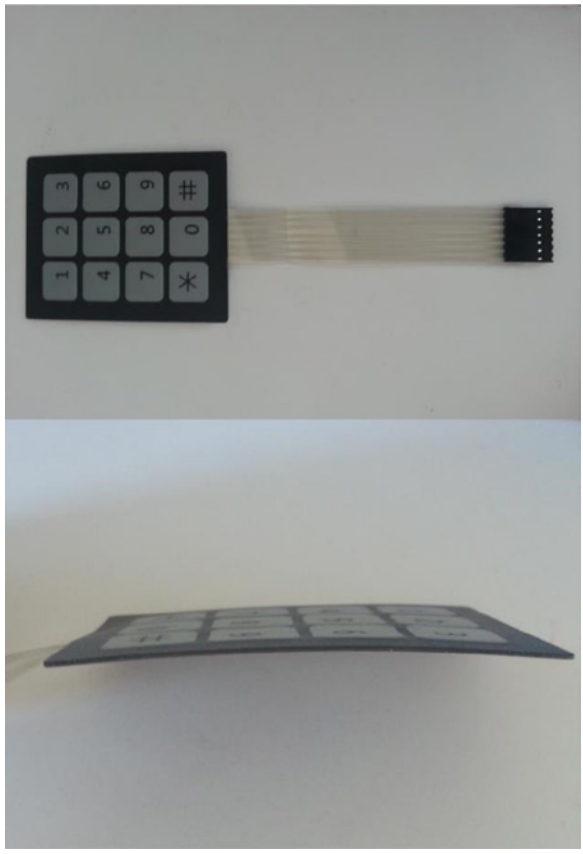

Figure 9-3. *Top view (top) and lateral view (bottom) of the flexible membrane keypad*

If you cannot find this keypad easily, you can use any other type, but make sure you adapt the connections represented in the following sections.

The next sections describe the software functionality, the hardware connections, and the code that will be integrated with the system in general.

The Keypad Functionality

Before you learn about the connection and code, it is necessary to understand how the system will work.

Once you boot Intel Galileo, the system will be *locked***,** which means if the PIR sensor detects movement, an alert will be sent. However, it's possible to unlock the system by entering the correct PIN. If the PIN entered matches the secret PIN, the system returns to the unlocked state and the PIR sensor doesn't send any alarms.

If the user enters the same PIN number again, the system is rearmed. In other words, the same PIN number is used to arm and disarm the system, locking and unlocking the system respectively.

There is a green LED that's on when the system is locked and off when the system is unlocked. It gives the user a visual indication as to whether the system is armed or not.

Every time the user press a key the LED blinks for a small period of time (the default is 500ms) to confirm the pressing event was recognized by the system. This is independent of whether the system is locked or not. In order words, the LED blinks even when it is already on (system is locked) or off (system is unlocked).

The keypad has a ENTER function that's used when the pound key (#) is pressed and the user must press this key every time he wants to confirm the PIN number. The star key (*)resets all the digits pressed and allows the user to re-enter the PIN if some wrong key was pressed.

The Keypad Connection

The rows 0 to 3 are related to the connector pins 1, 2, 3, and 4, respectively and columns 0 to 2 are related to the connector pins 5, 6, and 7, respectively. These pins must be connected to the Intel Galileo digital ports and are easily customizable using software. See Figure 9-4.

Figure 9-4. *Keypad disposition of pins and identification of rows and columns*

To test the keypad, an LED is being added. Figure 9-5 shows the keypad schematics.

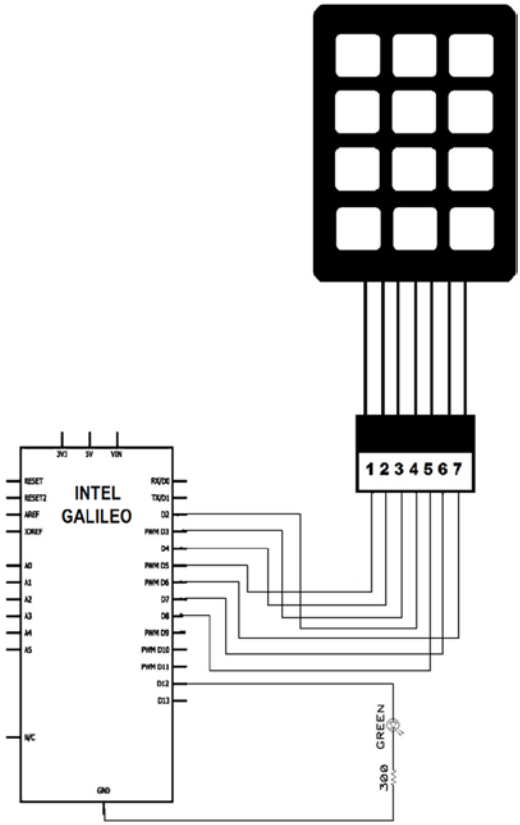

Figure 9-5. *The keypad with an LED connection*

Note for this project that the LED is being connected to the digital port 13, which means the LED is not really necessary because the built-in LED is also connected to pin 13. It is your choice to use an external LED or not.

Writing and Testing the Keypad Software

This section uses the code from Listing 9-1 (`keypad_testcode.ino`) contained in the `code` folder of this chapter. You also need to download the Keypad library from `http://playground.arduino.cc/Code/Keypad`.

Once the library is downloaded, integrate it with the Intel Galileo IDE as follows:

1. Locate where the Arduino IDE is installed on your computer and find the directory called `libraries`. For example, `c:\arduino-1.5.3-windows\arduino-1.5.3\libraries`.
2. Extract the ZIP file and make sure the library is in the same level as the other libraries (see Figure 9-6).

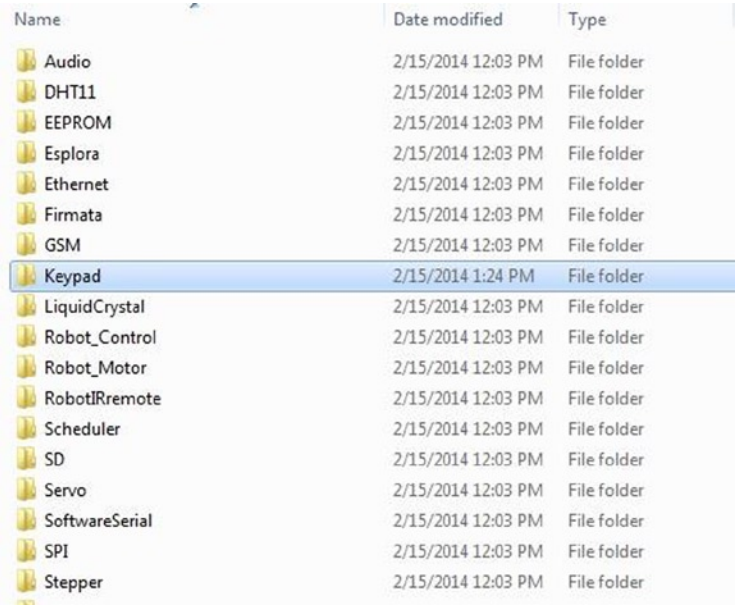

Figure 9-6. *Installation of the keypad library in the IDE*

3. Open `keypad_testcode.ino` and run it. Press CTRL+SHIFT+M to open the serial console debugger. See Listing 9-1.

Listing 9-1. keypad_testcode.ino

```
#include <Keypad.h>

enum SYSTEM_STATUS{
  LOCKED,     // 0
  UNLOCKED,   // 1
};

static SYSTEM_STATUS currentStatus = LOCKED;
const String password = "1968";
String input;
```

```
const byte ledPin = 12;

const byte ROWS = 4; // four rows
const byte COLS = 3; // three columns
char keys[ROWS][COLS] = {
    {'1','2','3'},
    {'4','5','6'},
    {'7','8','9'},
    {'*','0','#'}
};

byte rowPins[ROWS] = {5, 4, 3, 2}; // pins on Intel Galielo I/O
byte colPins[COLS] = {8, 7, 6};    // pins on Intel Galielo I/O

Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS );


void setup(){
    Serial.begin(115200);

    // in case there is an LED CONNECTED
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, HIGH);         // The default is system locked..
so, the LED must be HIGH
    keypad.addEventListener(handleKey); // this is the listener to handle
the keys
}

void loop(){

    // reading the keyboard
    char key = keypad.getKey();

    // if it's a valid key
    if (key) {

        if ((key != '#') &&  (key != '*'))
        {
          input += key;
        }
        Serial.print("key:");
        Serial.println(key);
    }
}
```

```
// this function is only called when the PIN code
// typed matches the secret PIN code and inverts
// the system logic. It means if the system was LOCKED
// it will be UNLOCKED and vice versa.
void updateLEDStatus() {
      if (currentStatus == LOCKED)
      {
           currentStatus = UNLOCKED;

           Serial.println("SYSTEM UNLOCKED");

           // turn OFF the LED
           digitalWrite(ledPin, LOW);
       }
       else
       {
           currentStatus = LOCKED;


           Serial.println("SYSTEM LOCKED");

           // turn ON the LED
           digitalWrite(ledPin, HIGH);
        }
}

// this function is responsible to handle
// the keypad events
void handleKey(KeypadEvent key){

    switch (keypad.getState())
    {
    case PRESSED:


        digitalWrite(ledPin, !digitalRead(ledPin));
        delay(500);
        digitalWrite(ledPin, !digitalRead(ledPin));

        // this is our ENTER
        if (key == '#') {
          Serial.println(input);
          if (input == password)
          {
               updateLEDStatus();
          }
```

```
          input = "";

        }

        break;

    case RELEASED:

        // this is our CLEAR
        if (key == '*') {
           input = "";
        }
        break;
    }
}
```

Reviewing the Code

The original code sets the secret PIN to **1968**, which is the year Intel was founded. Feel free to change the PIN. To do this, change this line:

```
const String password = "1968";
```

The first line of the code calls the `Keypad.h` file responsible for having the function calls from the library. The next lines are related to an enumerator and they describe the two possible states of the system—`LOCKED` and `UNLOCKED`.

The next lines determine the keypad design and to which pins in the digital port I/O the keys must be connected.

```
const byte ROWS = 4; // four rows
const byte COLS = 3; // three columns
char keys[ROWS][COLS] = {
    {'1','2','3'},
    {'4','5','6'},
    {'7','8','9'},
    {'*','0','#'}
};

byte rowPins[ROWS] = {5, 4, 3, 2}; // pins on Intel Galielo I/O
byte colPins[COLS] = {8, 7, 6};    // pins on Intel Galielo I/O

Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS );
```

Note the bidimensional array called `keys`. It describes the keypad design used in this project and the bytes' arrays called `rowPins` and `colPins` determine how the keypad is connected to the Intel Galileo digital port I/O.

If you are using a different keypad with different connections and designs, you need to change these lines accordingly.

The following line:

```
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS );
```

Is used only to "join the pieces" of the library. In other words, it is how the software informs the library of the keypad design and how the keypad's connector is connected to the Intel Galileo digital ports.

The `setup()` function contains a function callback that will be called when a key event like press, release, and hold is detected.

```
keypad.addEventListener(handleKey);
```

In this case, the callback function is named `handleKey()`.

In the function `loop()`, the keys are read through the function `getKey()`, as follows.

```
char key = keypad.getKey();
```

And still in the function `loop()`, if the keys are valid and different from # and `!`, they are accumulated in the variable called `input` because the intention is only to store numeric keys types in this variable.

```
if (key) {

        if ((key != '#') &&  (key != '*'))
        {
          input += key;
        }
```

Finally, you use the callback function `handleKey()` when the `PRESS` and `RELEASE` events are detected and some actions are done. For the `PRESS` event, if the user types #, this key acts similarly to an `ENTER` event. The pin sequence accumulated in the variable input is checked to see if it matches the secret PIN number. If it matches, the system is UNLOCKED or LOCKED, according to the current state. Recall that the LED is on when the system is LOCKED and off when it's UNLOCKED.

```
    switch (keypad.getState())
    {
    case PRESSED:
        // this is our ENTER
        if (key == '#') {

...
...
```

```
          if (input == password)
          {
               updateLEDStatus();
          }
...
...
```

For the RELEASE event, if the * key is pressed, the input variable is cleared and the user is allowed to re-enter the code.

```
    case RELEASED:

        // this is our CLEAR
        if (key == '*') {
           input = "";
        }
...
...
```

Running the Keypad Code

Run the code and press CTRL+SHIFT+M to see the serial console debugger.

You will immediately see the LED on, which means the initial state of the system is LOCKED. Type some keys and you should see the LED blinking for small period.

If you press # and the PIN typed matches the secret PIN code, then the LED will turn off and the system will be UNLOCKED.

If you type the secret code again, the system will be rearmed and the LED will turn on.

If you type the key *, all input is cleared.

Figure 9-7 shows the keypad test.

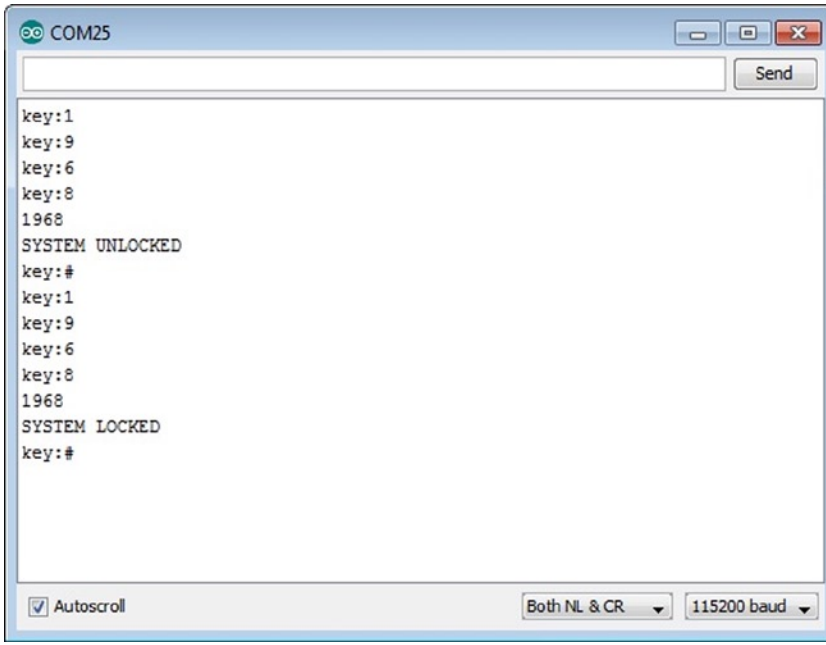

Figure 9-7. *Debugging the keypad functionality*

Testing the PIR Sensor

The human sensor passive infrared (PIR) module HC-SR501 is very simple to test. The purpose of this sensor is to detect movement in an environment. Figure 9-8 shows the PIR sensor used in this project.

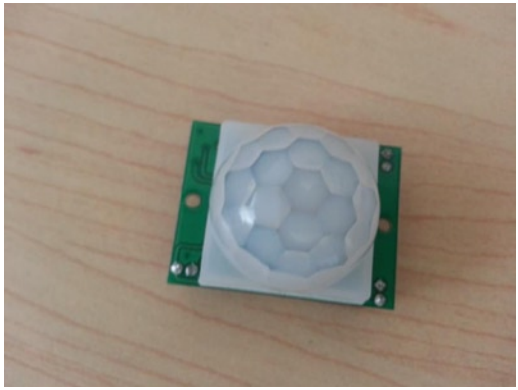

Figure 9-8. *PIR sensor*

The module requires some calibration according to your preferences or your needs. It's necessary to adjust the maximum distance the sensor must operate and to set how long the pulse indicating a detection must flash.

The module HC-SR501 does not have encapsulation and consequently it's not recommended for outdoor applications.

For more details about this module, check out `http://www.mpja.com/download/31227sc.pdf`.

The PIR Sensor Functionality

The module has enough connections for three signals, as demonstrated in Figure 9-9.

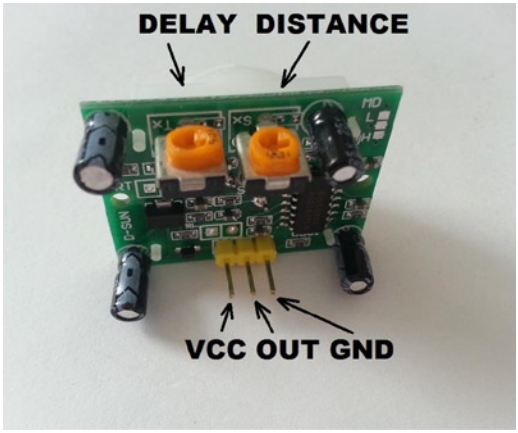

Figure 9-9. *PIR sensor headers*

When a movement is detected, the OUT pin remains HIGH during an interval of 5 to 300 seconds according to the adjustments made in the micro-potentiometer Tx.
The distance in the detection can be set between 2m (6.56ft) and 7m (22.9ft) using the micro-potentiometer Sx. See Figure 9-9 for details.

The PIR Sensor Connection

This PIR sensor can receive an input voltage between 5 and 20V. It means the sensor can use the +5V voltage supplied by Intel Galileo. If your intention is to create a demo or to understand how it all works, using the +5V provided by the Intel Galileo headers is more than enough. However, if you plan to use this sensor in your house and need to monitor a room quite far from Intel Galileo, it's better to use a 9V battery instead to pass wires all around your house and avoid the impedance provided by the wires interfere with the sensor functionality. According to the information related to this sensor, the quiescent current is lower than 50uA, and that's okay for a 9V battery.

If you want to connect the sensor directly to Intel Galileo, check out Figure 9-10.

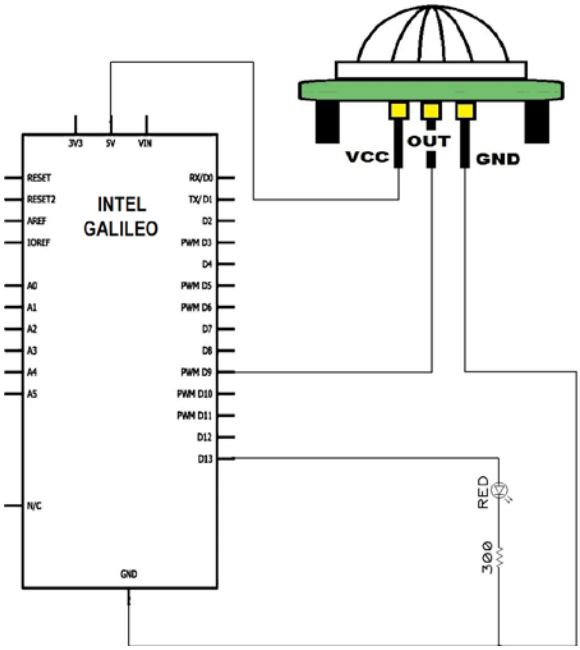

Figure 9-10. *PIR sensor connected directly to Intel Galileo*

Otherwise, if you want to use a 9V battery, check out the schematics represented in Figure 9-11.

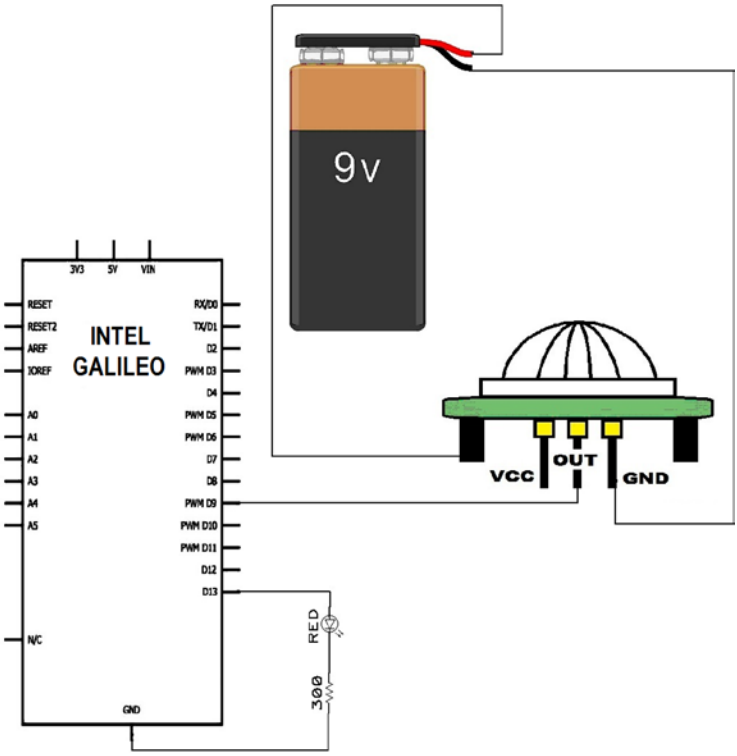

Figure 9-11. *PIR sensor using a 9V battery as the power supply*

Note the sensor is connected to the digital port number 9, which means if you were previously testing the keypad you can keep the keypad connected. Also, there is an LED connected to the pin 13, which means you can use the built-in LED to check the sensor functionality or you can connect an external LED, as used by the keypad test described earlier in this chapter.

Writing and Testing the PIR Sensor Software

For this, you'll use PIR_sensor_testcode.ino from Listing 9-2.

Listing 9-2. PIR_sensor_testcode.ino

```
//
// For testing Infrared HC-SR501 Pyroelectric Infrared Sensor
//

const byte ledPIRpin = 13;    // LED pin for the LED
const byte sensorPIR_Pin = 9; // input pin
byte pirState = LOW; //

void setup() {
    pinMode(ledPIRpin, OUTPUT);    // declare output
    pinMode(sensorPIR_Pin, INPUT); // declare input

    Serial.begin(115200);
}

void loop(){

    if (digitalRead(sensorPIR_Pin) == HIGH) { // input HIGH
        digitalWrite(ledPIRpin, HIGH);        // LED ON
        if (pirState == LOW)
        {

            // we have just turned on
            Serial.println("OPS!!! Someone here!!! motion DETECTED!");

           // We only want to print on the output change, not state
           pirState = HIGH;
      }
    }
    else
    {

       digitalWrite(ledPIRpin, LOW); // turn LED OFF
       if (pirState == HIGH){

       // we have just turned of
       Serial.println("Waiting for next moviment");

       // We only want to print on the output change, not state
       pirState = LOW;
       }
   }
}
```

The only thing the code does is detect the level changes in pin 9, which is connected to the sensor header OUT using the digital port. The sketch has a variable called `pirState` that starts LOW. When the sensor detects movement, the variable assumes the HIGH state. Then when the delay expires, the sensor header OUT goes to LOW and the `pirState` variable changes the state to LOW, thereby indicating there is no presence.

Run the sketch and press CTRL+SHIFT+M to see the serial debugger. The serial debugger will print messages when movement is detected.

This is the time to make adjustments! The potentiometers indicated in Figure 9-9 must be changed this way:

1. Distance Settings (Sx): Turn to the right and distance (sensitivity) increases; turn to the left and distance decreases.
2. Time Setting (Tx): Turn to the right and time delay increases; turn to the left and time reduces.

Remember the limitations: A distance between 2m (6.56ft) and 7m (22.9ft) and a delay between 5 and 300 seconds.

Testing the YwRobot Relay Module

The two-channel relay module YwRobot used in this chapter is one of the simplest components to test, but it is necessary to be aware of the limitation of the relays in terms of amperes and the maximum voltage that can operate when connected to lamps and other AC devices. Otherwise, your house may end up in flames.

This is the same relay that was mentioned in Chapter 5 as an improvement for the moisture sensor system to activate and deactivate electric valves.

The YwRobot Relay Module Functionality

In this chapter, the relay module YwRobot can support a maximum of 250V AC and 10A AC, or 30V DC with 10DC relay. This is enough to support simple lamps.

The input operation is 5V with a TTL level for relays. That means the Intel Arduino digital ports can be used to operate the module.

The module used in this project contains two relays, each of which is commanded according to the schematics shown in Figure 9-12.

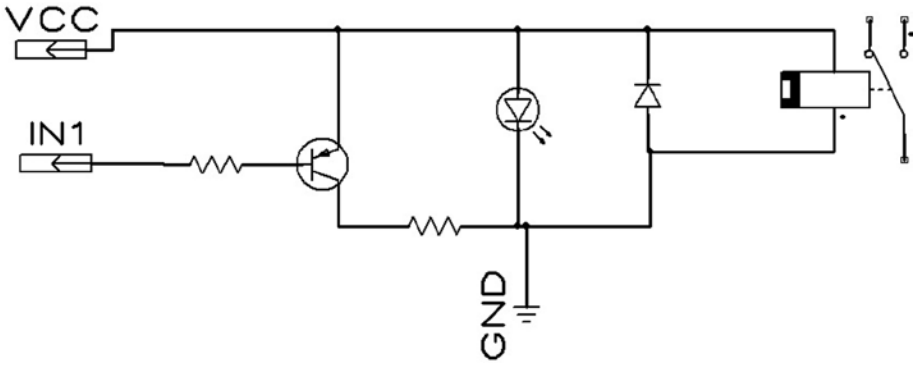

Figure 9-12. *Relay module command*

Each relay in this module contains a simple drive circuit with a PNP 8550 transistor that operates with 5V DC. When IN1 is LOW, the circuit is active and the relay is switched; otherwise, when IN1 is HIGH the circuit is off and the relay is inactive.

The module offers some LEDs that help debug because they indicate whether the module is powered on and indicate whether the relay is active or not, by setting LOW and HIGH respectively in the IN headers (see Figure 9-13).

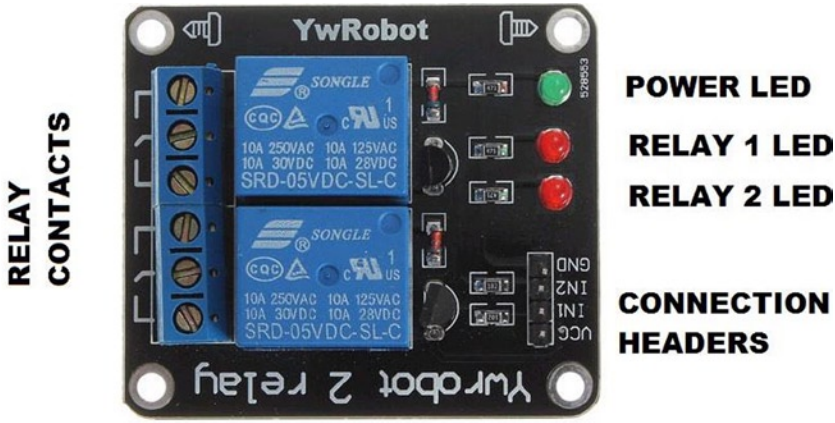

Figure 9-13. *YwModule LED and headers*

The YwRobot Relay Module Connection with Intel Galileo

The version used in this chapter contains two relays commanded by the header pins IN1 and IN2. You can buy modules with more relays and the only difference is the number of "INs" because the module contains unique VCC and GND (ground) headers, independent of the quantity of relays.

The VCC in the schematics is connected to Intel Galileo 5V. The GND is connected to the Intel Galileo ground and IN1 is connected to a digital port I/O of your preference. The driver circuit is active when the module input (IN) is set to LOW and inactive when it's set to HIGH.

For this project, the commands IN1 and IN2 are connected to Intel Galileo's ports 10 and 11, respectively. See Figure 9-14 as a reference.

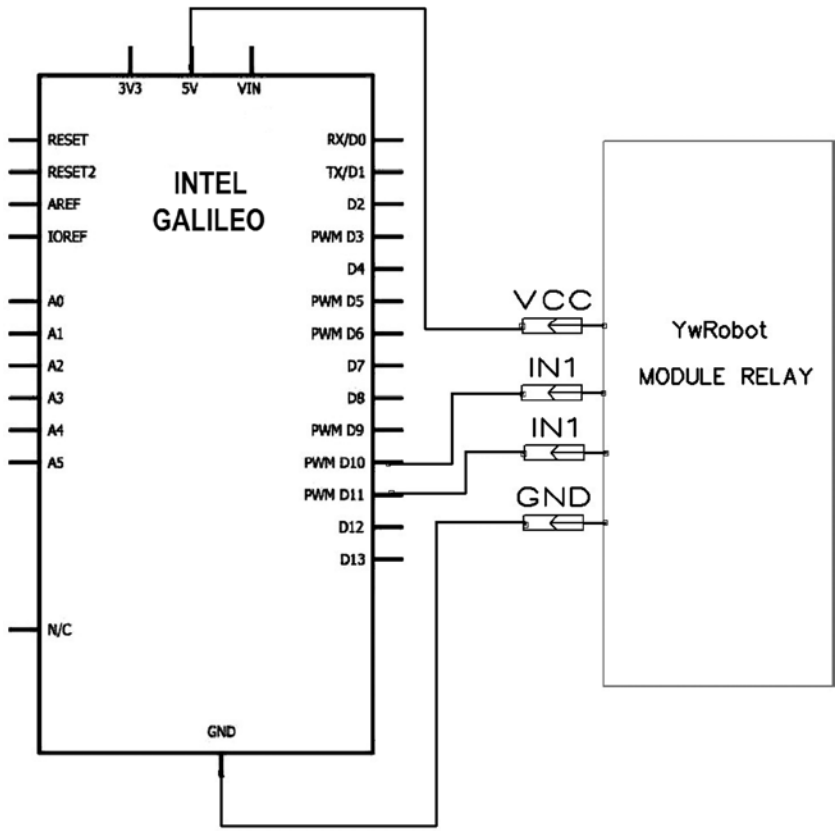

Figure 9-14. *Connecting the module relays to Intel Galileo*

The YwRobot Relay Module Connection with External Lamps

Warning Before connecting your lamps to the AC with the module relay, it's important to note that 110 VAC, 127 VAC, and 220 VAC can cause severe personal injury, death, or substantial property damage.

Make sure the lamp or device you want to command with this project requires a current and voltage that fits the capability provided by the relay module's specification and that all the installation is completed using the proper wires.

The connection of the lamp or device is shown in Figure 9-15.

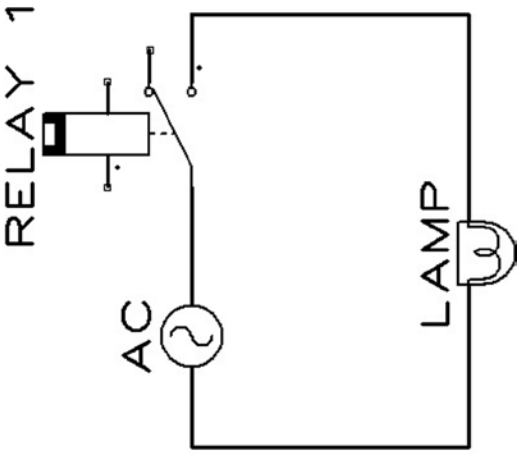

Figure 9-15. *Connecting a lamp to the one relay in AC*

When testing your system, it is not necessary to connect the lamps or any other AC component because the YwRobot module has LEDs that indicate connection, as explained in Figure 9-13. Therefore, you can consider this connection the last one to be made in your system.

Writing and Testing the YwRobot Relay Module Software

This section uses the code you can find listed as

Listing 9-3 provides the code for testing the module relay. `relaymodule_testcode.ino` is the simplest one in this chapter.

Listing 9-3. relaymodule_testcode.ino

```
//
// For testing YwRobot module relay
//

const byte relay1 = 10; // relay 1 command
const byte relay2 = 11; // relay 2 command


void setup() {
    pinMode(relay1, OUTPUT); // declare output
    pinMode(relay2, OUTPUT); // declare output

    Serial.begin(115200);
}

void loop(){

   digitalWrite(relay1, LOW);  // turn ON
   digitalWrite(relay2, HIGH); // turn OFF
   delay(5000);

   digitalWrite(relay1, HIGH); // turn OFF
   digitalWrite(relay2, LOW);  // turn ON
   delay(5000);
}
```

As you can see in the code, there is nothing special because the module relay input IN1 and IN2 are connected to the digital ports 10 and 11 on Intel Galileo.

The `loop()` function turns on and off the output of relay 1 controlled by IN1 and relay 2 controlled by IN2 in intervals of five seconds.

If you have the lamps connected to the relays, you will be able to see that when a lamp is on the other lamp is off and vice versa. If you do not have the lamps connected, you can see the LED in the YwRobot indicating which relay is active at the moment.

Testing the TMP36 Temperature Sensor

The TMP36 temperature sensor looks like a transistor. It doesn't require any calibration and provides an output voltage linearly proportional to the temperature in Celsius.

The TMP36 Temperature Sensor Functionality

The TMP36 has three pins, a VIN that must be between 2.7 and 5.5V DC, a GND, and a voltage output (VOUT).

There are different circuits with different precision described in the datasheet at http://www.analog.com/static/imported-files/data_sheets/TMP35_36_37.

The TMP36 Temperature Sensor Connection with Intel Galileo

Before you start to make a connection, it is very important to note, at the moment this book is being developed, that the datasheet revision G represents the TMP36 pin showing the *bottom* view instead the top.

Several developers claim that the sensor is extremely hot, but in fact it is because the sensor is connected inverted.

Figure 9-16 shows the bottom view of the sensor.

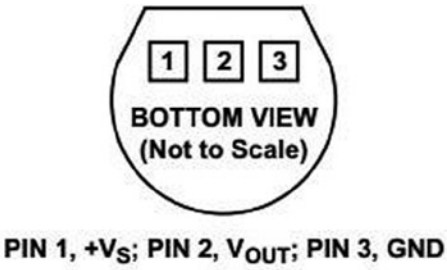

Figure 9-16. *TMP38, bottom view*

You need to connect the power to the sensor with 5V, connect the ground, and choose one of the analog ports to connect to the sensor VOUT. For this project, the analog port A0 is being used with VOUT and a capacitor with 0.1uF is used between the ground and the VCC (see Figure 9-17).

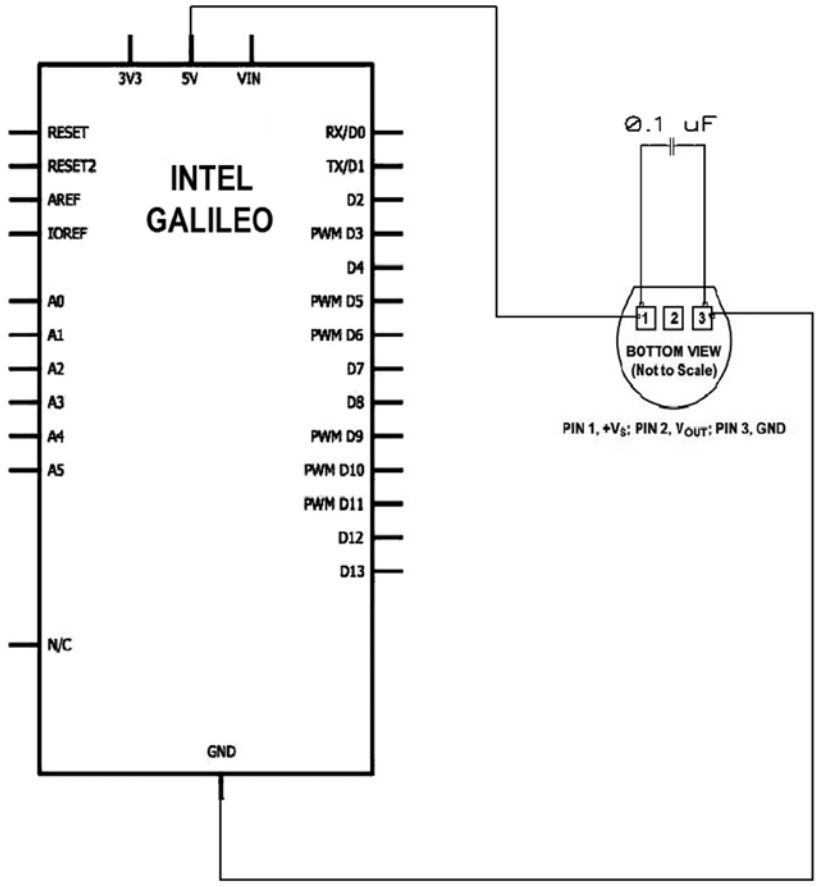

Figure 9-17. *The TMP36 connected to Intel Galileo*

For more details about this sensor, go to http://www.analog.com/static/imported-files/data_sheets/TMP35_36_37.pdf.

Writing and Testing the TMP36 Temperature Sensor Software

The code for testing TMP36 is provided in Listing 9-4.

Listing 9-4. tmp36_testcode.ino

```
//TMP36 VOUT pin connection
const byte sensorAnalogPin = 0;

/*
 * setup() - this function runs once you turn your Arduino on
 * We initialize the serial connection with the computer
 */
void setup()
{
  Serial.begin(115200);

}

void loop()
{
 //getting the voltage reading from the temperature sensor
 int reading = analogRead(sensorAnalogPin);

 float VOUT = (reading * 5.0)/1024.0;

 Serial.print(" volts");
 Serial.println(VOUT);

 // converting to Celsius according to the datasheet
 float tempCelsius = (VOUT - 0.5) * 100 ;

 Serial.print(" degrees Celsius:");
 Serial.println(tempCelsius);

 // converting to Fahrenheit
 float tempF = (tempCelsius * 9.0 / 5.0) + 32.0;

 Serial.print("degrees Fahrenheit:");
 Serial.println(tempF);

 delay(1000);
}
```

The code converts the real voltage in the output after reading the A0 port:

```
int reading = analogRead(sensorAnalogPin);

float VOUT = (reading * 5.0)/1024.0;
```

To convert to Celsius, use this formula:

```
Temperature Celsius = (VOUT - 0.5V)*100
```

The code to get the temperature in Celsius is as follows:

```
// converting to Celsius according the datasheet
float tempCelsius = (VOUT - 0.5) * 100 ;
```

Creating the Sketch

With all hardware components tested, it is time to create the sketch. The code will be only a junction of all the test code used to test the peripherals, including only the portion necessary to communicate with the web server using datagrams (UDP).

Intel Galileo does not require any special Arduino shield to get network connections and you can set the connection using an Ethernet cable or WiFi card with a mPCIe bus such as the Intel Centrino wireless N-135 or even using a modem card.

Considering that no shield is used, it is not necessary to use the libraries and write code for such a connection. This allows you to make the connection with a simple setting in Intel Galileo, as explained in the Chapter 5.

The sketch will need be able to send and receive datagrams. Linux libraries offer functions to do that, which means you don't have to use libraries created specifically for shields, which aren't used in this project.

Sending UDP Messages

The following code snippet represents a function responsible for sending the datagram. Note a socket descriptor `socketfd` is opened with the parameters `SOCK_DGRAM` and `IPPROTO_UDP` in order to specify the datagram and non-oriented connection.

The port the messages must be sent through is specified by `WEBSERVER_UPD_PORT` in the `htons()` function.

The IP address used to send the message is `loopback 127.0.0.1` because the sketch and the web server run in Intel Galileo, which means the same device is sharing the same loopback port. You could use the IP provided by the element adapter when the connection is established but you should change the code all the time or implement some mechanism to pass this information parameterized.

To send the message, the function `sendTo()` is used.

```
#define WEBSERVER_UDP_PORT 2010 // this port is used to send message events
to Node.js

void sendUDPMessage(String protocol)
{
  struct sockaddr_in serv_addr;
  int sockfd, i, slen=sizeof(serv_addr);

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
  {
    printError("socket");
    return;
  }

  bzero(&serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(WEBSERVER_UDP_PORT);

  // considering the sketch and the web server run into Galileo
  // let's use the loopback address
  if (inet_aton("127.0.0.1", &serv_addr.sin_addr)==0)
  {
      printError("inet_aton() failed\n");
      close(sockfd);
      return;
  }

   char send_msg[BUFFERSIZE];  // more than enough
   memset((void *)send_msg, sizeof(send_msg), 0);
   protocol.toCharArray(send_msg, sizeof(send_msg), 0);

  if (sendto(sockfd, send_msg, strlen(send_msg), 0, (struct sockaddr
*)&serv_addr, sizeof(serv_addr))==-1)
        printError("sendto()");

  close(sockfd);

}
```

Receiving UDP Messages

To receive the datagrams, the socket descriptor is opened in the same way as when sending UDP messages. However, as soon as the socket is opened, the function `bind()` must be called. It specifies the port that will be used to receive messages from the web server, named `SKETCH_UDP_PORT` in this case.

```
#define SKETCH_UDP_PORT 2000    // this port is used to receive message
events from Node.js

int populateUDPServer(void)
{

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
      printError("socket");
    else
      Serial.println("Server : Socket() successful\n");

    bzero(&my_addr, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(SKETCH_UDP_PORT);
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sockfd, (struct sockaddr* ) &my_addr, sizeof(my_addr))==-1)
      printError("bind");
    else
      Serial.println("Server : bind() successful\n");

    memset(msg_buffer, 0, sizeof(msg_buffer));
}
```

Once the socket descriptor is opened and the `bind()` function is called, it is possible to receive the datagrams from the web server by calling the `recvfrom()` function periodically. The `loop()` function is perfect for this.

However, the `recvfrom()` function is used and is a *blocking* function. When the web server isn't sending messages, the function will block the execution of whole sketch, thus invalidating the project.

To resolve this issue, another function called `select()` must be used in conjunction with `recvfrom()` in order to implement a timeout in the blocking process. The `recvfrom()` function will respect the timeout and allow the program to be executed.

A flag is used to control this process and uses functions like `FD_ZERO()` for reset and `FD_SET()` for binding the flag with the socket descriptor. Such functions must be called before the `select()` function.

There is a detail when `select()` is called regarding the first parameter that must be the socket descriptor plus one, according to the documentation.

In the next code snippet, the timeout is set to 1,000 microseconds. If no message is received and the timeout occurs, the sensor's values are read and sent to the web server every second so that the web page is constantly updated.

If some message is received, the function `FS_ISSET()` will state it and the `recvfrom()` will be called and the data will be received.

```
void loop() {

      if (time0 == 0) time0 = millis();
...
...
...

        // clear the set ahead of time
      FD_ZERO(&readfds);
      FD_SET(sockfd, &readfds);

      // wait until either socket has data ready to be recvfrom()
         (timeout 1000 usecs)
      tv.tv_sec = 0;
      tv.tv_usec = 1000;

      rv = select(sockfd + 1, &readfds, NULL, NULL, &tv);

        if(rv==-1)
        {
            Serial.println("Error in Select!!!");
        }
        if(rv==0)
        {

            // TIMEOUT!!!!

            if ((millis()-time0) >= 1000)
            {
                // reached 1 seconds.. let's reads the sensor and
                   send a message!!!
                time0 = millis();


...
...
...

               sendUDPMessage(protocol);

            }
        }
```

```
        // checking if the UDP server received some message from the web page
        if (FD_ISSET(sockfd, &readfds))
        {

            if (recvfrom(sockfd, msg_buffer, BUFFERSIZE, 0,
            (struct sockaddr*)&cli_addr, &slen)==-1)
                printError("recvfrom()");
...
...
...

            // let's clear the message buffer
            memset(msg_buffer, 0, sizeof(msg_buffer));

        }

}
```

Joining All Code in a Single Sketch

If you tested all the peripherals in this chapter, you should realize that the connection of the components to the Intel Galileo headers uses different ports. This allows you to keep all the peripherals connected and test them individually, thus making the hardware integration very easy.

Listing 9-5 is a simple join of all the test code. It integrates the UDP code for sending and receiving messages, but also reads the sensors and the system status after using the keypad to arm and disarm the system. Such readings are done every second. In other words, when you have the timeouts provided by `select()` and FD functions, as described.

The code also parses the commands received through the UDP server and changes the switch relays in the YwRobot module.

Listing 9-5. sketch_client.ino

```
#include <stdio.h>

// includes for the UDP connections
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

// keypad
#include <Keypad.h>
```

```
// debugging
#define DEBUG 1                 // 1 to see the debug messages in the serial
                                   console, or 0 to disable

#define BUFFERSIZE 512          // UDP is limited and must be very
                                   short. 512 bytes is more than enough
#define SKETCH_UDP_PORT 2000    // this port is used to receive message
                                   events from Node.js
#define WEBSERVER_UDP_PORT 2010 // this port is used to send message events
                                   to Node.js
#define SENSOR_READ_INTERVAL 10 // number of seconds to read sensors and
                                   report to website

// for the UDP server
   struct sockaddr_in my_addr, cli_addr;
    int sockfd, i;
    socklen_t slen=sizeof(cli_addr);
    char msg_buffer[BUFFERSIZE];
    fd_set readfds;
    struct timeval tv;
    int rv,n;

// pin connections
const byte sensorAnalogPin = 0;

// Keypad
enum SYSTEM_STATUS{
  LOCKED,     // 0
  UNLOCKED,   // 1
};

static SYSTEM_STATUS currentStatus = LOCKED;
const String password = "1968";     // Intel foundation year..
String input;

const byte ledPin = 12;

const byte ROWS = 4; // four rows
const byte COLS = 3; // three columns
char keys[ROWS][COLS] = {
    {'1','2','3'},
    {'4','5','6'},
    {'7','8','9'},
    {'*','0','#'}
};

byte rowPins[ROWS] = {5, 4, 3, 2}; // pins on Intel Galielo I/O
byte colPins[COLS] = {8, 7, 6};    // pins on Intel Galielo I/O
```

```
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS, COLS );

// PIR sensor
const byte sensorPIR_Pin = 9; // input pin
byte pirState = LOW; //
const byte ledPIRpin = 13;

// Relays
const byte relay1 = 10; // relay 1 command
const byte relay2 = 11; // relay 2 command

// time control
unsigned long time0 = 0;

// this function is only called when some error happens
void printError(char *str)
{
    Serial.print("ERROR: ");
    Serial.println(str);
}

// this function is reponsible for sending UDP datagrams
void sendUDPMessage(String protocol)
{
  struct sockaddr_in serv_addr;
  int sockfd, i, slen=sizeof(serv_addr);

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
  {
    printError("socket");
    return;
  }

  bzero(&serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(WEBSERVER_UDP_PORT);

  // considering the sketch and the web server run into Galileo
  // let's use the loopback address
  if (inet_aton("127.0.0.1", &serv_addr.sin_addr)==0)
  {
      printError("inet_aton() failed\n");
      close(sockfd);
      return;
  }
```

```
   char send_msg[BUFFERSIZE];  // more than enough
   memset((void *)send_msg, sizeof(send_msg), 0);
   protocol.toCharArray(send_msg, sizeof(send_msg), 0);

  if (sendto(sockfd, send_msg, strlen(send_msg), 0, (struct sockaddr *)
&serv_addr, sizeof(serv_addr))==-1)
        printError("sendto()");

  close(sockfd);

}

// this function is responsible to init the UDP datagram server
int populateUDPServer(void)
{

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
      printError("socket");
    else
      if (DEBUG) Serial.println("Server : Socket() successful\n");

    bzero(&my_addr, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(SKETCH_UDP_PORT);
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sockfd, (struct sockaddr* ) &my_addr, sizeof(my_addr))==-1)
      printError("bind");
    else
      if (DEBUG) Serial.println("Server : bind() successful\n");

    memset(msg_buffer, 0, sizeof(msg_buffer));

}

// reading the temperature sensor in Celsius
float readTemperatureSensor()
{
  // getting the voltage reading from the temperature sensor
 int reading = analogRead(sensorAnalogPin);

 float VOUT = (reading * 5.0)/1024.0;

 if (DEBUG) {
     Serial.print(" volts");
     Serial.println(VOUT);
 }
```

```
 // converting to Celsius according to the datasheet
 float tempCelsius = (VOUT - 0.5) * 100 ;

 if (DEBUG) {
     Serial.print(" degrees Celsius:");
     Serial.println(tempCelsius);
 }

 return tempCelsius;
}

// convert celsius to fahrenheit
float convertTempToF(int celsius) {
  // converting to Fahrenheit
 float tempF = (celsius * 9.0 / 5.0) + 32.0;

 if (DEBUG) {
     Serial.print("degrees Fahrenheit:");
     Serial.println(tempF);
 }

 return tempF;

}

// update the LED status when the system is armed or disarmed
void updateLEDStatus() {
      if (currentStatus == LOCKED)
      {
           currentStatus = UNLOCKED;

           if (DEBUG)
           {
              Serial.println("SYSTEM UNLOCKED");
           }

           //turn OFF the LED
           digitalWrite(ledPin, LOW);
       }
       else
       {
           currentStatus = LOCKED;

           if (DEBUG)
           {
               Serial.println("SYSTEM LOCKED");
           }
```

```
            // turn ON the LED
            digitalWrite(ledPin, HIGH);
        }
}

// this is the key handler for the PRESS, RELEASE, and HOLD event
void handleKey(KeypadEvent key){


    switch (keypad.getState())
    {
    case PRESSED: // this is our ENTER

        digitalWrite(ledPin, !digitalRead(ledPin));
        delay(500);
        digitalWrite(ledPin, !digitalRead(ledPin));

        if (key == '#') {
          if (DEBUG) Serial.println(input);
          if (input == password)
          {
               updateLEDStatus();
          }
          input = "";

        }

        break;

    case RELEASED: // this is our CLEAR
        if (key == '*') {
           input = "";
        }
        break;
    }
}

void setup() {

  Serial.begin(115200);

  delay(3000);

  // init variables for UDP server
  populateUDPServer();
```

```
  // keypad
  pinMode(ledPin, OUTPUT);
  pinMode(ledPIRpin, OUTPUT);

  digitalWrite(ledPin, HIGH);         // The default is system locked.. so,
the LED must be HIGH
  digitalWrite(ledPIRpin, LOW);       // Let's let the PIR sensor change the
LED state

  keypad.addEventListener(handleKey); // this is the listener to handle the
keys

  // relays
  pinMode(relay1, OUTPUT); // declare output
  pinMode(relay2, OUTPUT); // declare output

  digitalWrite(relay1, HIGH);
  digitalWrite(relay2, HIGH);

}

void loop() {

    if (time0 == 0) time0 = millis();

    // checking the keypad
     char key = keypad.getKey();

    if (key) {

        if ((key != '#') &&  (key != '*'))
        {
          input += key;
        }
        if (DEBUG)
        {
            Serial.print("key:");
            Serial.println(key);
        }
    }

    // PIR sensor
     if (digitalRead(sensorPIR_Pin) == HIGH) { // input HIGH
        digitalWrite(ledPIRpin, HIGH);         // LED ON
```

```
        if (pirState == LOW)
        {

            // we have just turned on
            Serial.println("OPS!!! Someone here!!! motion DETECTED!");

           // We only want to print on the output change, not state
           pirState = HIGH;
       }
    }
    else
    {

       digitalWrite(ledPIRpin, LOW); // turn LED OFF
       if (pirState == HIGH){

       // we have just turned off
       if (DEBUG) Serial.println("Waiting for next moviment");

       // We only want to print on the output change, not state
       pirState = LOW;
       }
  }

        // clear the set ahead of time
      FD_ZERO(&readfds);
      FD_SET(sockfd, &readfds);
      // wait until either socket has data ready to be recv()d
         (timeout 1000 usecs)
      tv.tv_sec = 0;
      tv.tv_usec = 1000;

      rv = select(sockfd + 1, &readfds, NULL, NULL, &tv);

        if(rv==-1)
        {
               if (DEBUG)
               {
                     Serial.println("Error in Select!!!");
               }
        }
```

```
if(rv==0)
{

    // TIMEOUT!!!!

    if ((millis()-time0) >= 1000)
    {
        // reached 1 seconds  let's read the sensor and send a message!!!
        time0 = millis();

        String protocol = "";

        if (pirState == HIGH)
        {
             protocol += "*INTRUDER!!!*";
        }
        else
        {
             protocol += "*NO DETECTION*";
        }

                  // reading the temperature sensor
        int tempC = readTemperatureSensor();
        int tempF = convertTempToF(tempC);

        char msg[20];
        memset(msg, 0, sizeof(msg));
        sprintf(msg, "%dC - %dF", tempC, tempF);

        protocol += "*";
        protocol += msg;


        // checking the system status
        if (currentStatus == LOCKED)
        {
            protocol += "*ARMED*";
        }
        else
        {
            protocol += "*DISARMED*";
        }
        sendUDPMessage(protocol);

    }
}
```

```
// checking if the UDP server received some message from the web page
if (FD_ISSET(sockfd, &readfds))
{

    if (recvfrom(sockfd, msg_buffer, BUFFERSIZE, 0,
       (struct sockaddr*)&cli_addr, &slen)==-1)
    {
        printError("recvfrom()");
        return; // let's abort the loop
    }
    if (DEBUG)
    {
         Serial.println("Received packet from %s:%d\nData:");
         Serial.println(inet_ntoa(cli_addr.sin_addr));
         Serial.println(msg_buffer);
    }

    String checkResp = msg_buffer;
    if (checkResp.lastIndexOf("L1ON", 0) < 0)
    {
         // There is no L1ON in the string.. let's switch off the relay
         digitalWrite(relay1, HIGH);

         if (DEBUG) Serial.println("The lamp 1 is OFF");

    }
    else
    {
         // Oops.. let's switch relay 1 to ON
         digitalWrite(relay1, LOW);

         if (DEBUG) Serial.println("The lamp 1 is ON");

    }

    if (checkResp.lastIndexOf("L2ON", 6) < 0)
    {
         // There is no L2ON in the string.. let's switch off the relay
         digitalWrite(relay2, HIGH);

         if (DEBUG) Serial.println("The lamp 2 is OFF");
    }
```

```
            else
            {
                 // Oops.. let switch relay 2 to ON
                 digitalWrite(relay2, LOW);

                 if (DEBUG)Serial.println("The lamp 2 is ON");

            }

            // let's clear the message buffer
            memset(msg_buffer, 0, sizeof(msg_buffer));

        }
}
```

Creating Your Own Web Server with node.js

By now you should have tested all the hardware components with the small code snippets, so it is time to start writing and testing the main component—the web server.

This project uses node.js. As mentioned, node.js tried to simplify the development of the backend software. Maybe if you program in HTML before you should have played with JavaScript running in your web pages. If you need something very powerful, you should have implemented something using Python, Java, Perl scripts in a remote server, and so on. However, node.js brings the simplicity of JavaScript usage into the context of a web server in a light and easy implementation.

Updating node.js

Before writing and testing the code, make sure that the node.js is version equal or greater than 0.10.25. To check this, you can type the following in the terminal shell:

```
root@clanton:~# node --version
v0.10.25
```

or

```
root@clanton:~# node -v
v0.10.25
```

If you do not have the recommended version, you can update your system in three ways.

Option 1: Updating node.js with opkg (recommended)

Edit this file:

```
root@clanton:~# vi /etc/opkg/base-feeds.conf
```

And add the following lines:

```
src/gz all      http://repo.opkg.net/galileo/all
src/gz clanton  http://repo.opkg.net/galileo/clanton
src/gz i586     http://repo.opkg.net/galileo/i586
```

The next step is to update opkg and node.js by typing:

```
root@clanton:~# opkg update
```

and then typing:

```
root@clanton:~# opkg upgrade nodejs --force-overwrite
```

Option 2: Updating node.js with the Source

It is possible to build node.js directly into Intel Galileo. Suppose you want to install version 0.10.25 (the minimum recommended), which is available at http://nodejs.org/dist/.

You need to add the following:

```
root@clanton:~# wget http://nodejs.org/dist/v0.10.25/node-v0.10.25.tar.gz
root@clanton:~# tar -zxvf node-v0.10.25.tar.gz
root@clanton:~# cd node*
root@clanton:~#./configure
root@clanton:~# make
root@clanton:~# make install
```

If the "make" command above fails, you need to install the build tools. For this, just update the /etc/opkg/base-feeds.conf as done in the "Option 1" and execute the command "opkg install packagegroup-core-buildessential --force-overwrite".

Option 3: Updating node.js in the Yocto Build

If you are building your own images and do not want to update node.js through the prompt shell, you can add a new recipe to your Yocto build and have node.js be automatically updated.

There is a thread on the Intel community that includes step-by-step instructions for this process. If you are interested, check out https://communities.intel.com/thread/48416.

About the npm

When you install node.js, it comes with some modules installed as dependences. For example, there is the `dgram` used for the UDP connection, the `fs` for file system operations, the `http` to provide HTTP server and client functionality, and others.

A module is necessary in a program when it's explicitly called by the `require` instruction, as shown:

```
var http = require("http").createServer(onRequest),
    fs = require('fs'),
    url = require('url'),
    cheerio = require('cheerio'),
    dgram = require('dgram'),
```

However, not all modules are pre-installed when you install node.js, because the community is constantly developing new packages to provide new modules. You can install them according your project's needs. Before you learn about the packages that will be used with this project, it is necessary to understand how to install them using the `npm` tool.

The node.js file has a package manager named `npm` that allows you to add and remove external packages and install modules created for node.js.

Most developers think that `npm` stands for "node package manager," but this is not exactly accurate. According to the author, `npm` is a recursive bacronymic abbreviation for "`npm` is not an acronym." William Shakespeare writes in *Romeo and Juliet* that "A rose by any other name would smell as sweet." Thus, you can call `npm` a node package manager if you want, because it works like one.

This project uses `npm` version 1.3.24. You can check your version with the following command:

```
root@clanton:~# npm -v
1.3.24
```

If you want to update `npm`, use this command:

```
root@clanton:~# npm update npm -g
```

To check the version installed in your system, you need to use the `ls`, `list`, or `la` arguments, followed by the package name. For example:

```
root@clanton:~# npm ls socket.io
/home/root
└── socket.io@0.9.16
```

If the package does not exist, the word `empty` will be displayed:

```
root@clanton:~# npm ls idonotknow
/home/root
└── (empty)
```

To list *all* packages and the dependences of each one installed on your system, use only the argument `list`:

```
root@clanton:~# npm list
/home/root
├─┬ cheerio@0.13.1
│ ├─┬ CSSselect@0.4.0
│ │ ├── CSSwhat@0.4.1
│ │ └─┬ domutils@1.3.0
│ │   └── domelementtype@1.1.1
│ ├── entities@0.3.0
│ ├─┬ htmlparser2@3.4.0
│ │ ├── domelementtype@1.1.1
│ │ ├── domhandler@2.2.0
│ │ ├── domutils@1.3.0
│ │ └─┬ readable-stream@1.1.10
│ │   ├── core-util-is@1.0.1
│ │   ├── debuglog@0.0.2
│ │   └── string_decoder@0.10.25
│ └── underscore@1.5.2
└─┬ socket.io@0.9.16
  ├── base64id@0.1.0
  ├── policyfile@0.0.4
  ├── redis@0.7.3
  └─┬ socket.io-client@0.9.16
    ├─┬ active-x-obfuscator@0.0.1
    │ └── zeparser@0.0.5
    ├── uglify-js@1.2.5
    ├─┬ ws@0.4.31
    │ ├── commander@0.6.1
    │ ├── nan@0.3.2
    │ ├── options@0.0.5
    │ └── tinycolor@0.0.1
    └── xmlhttprequest@1.4.2
```

Before you install any package, first make sure the date and time of your Intel Galileo is properly set. You can use the command `date` and pass an argument formatted as `MMDDhhmmYYYY` (`MM` is the month, `DD` is the day, `hh` is the hour, `mm` is the minute, and `YYYY` is the year).

For example, the following command sets Intel Galileo to March 15, 2014 at 11:30AM.

```
root@clanton:~# date 0315113014
```

To install a package, use the argument `install` followed by the package name:

```
root@clanton:~# npm install socket.io
```

You can also install a specific version using `install` with `<name>@<version>`. For example:

```
root@clanton:~#  npm install sax@latest
```

More information regarding `npm` can be found at https://www.npmjs.org/doc/misc/npm-faq.html.

Installing Cheerio

The package named `cheerio` is a light and fast solution that replaces the `jsdom` package and allows you to parse and change HTML elements more easily.

To install `cheerio`, type the following command in the terminal shell:

```
root@clanton:~# npm install cheerio
```

Cheerio will be used to change the elements in the HTML page with the information sent by sketch. More details regarding `cheerio` can be found at https://github.com/MatthewMueller/cheerio.

This project uses the following `cheerio` version:

```
root@clanton:~# npm list cheerio
/home/root
└── cheerio@0.13.1
```

Installing socket.io

The package named socket.io is used in this project to receive data through sockets between the web server and the web pages. Essentially the connection of socket.io is TCP. The server implementation will be listening to the connections in the web server and the client implementation is done in the web page. More details related to socket.io can be found at https://www.npmjs.org/package/socket.io.

To install the socket.io package, use the following command line:

```
root@clanton:/# node install socket.io
```

This project uses the following socket.io version:

```
root@clanton:~# npm list socket.io
/home/root
└── socket.io@0.9.16
```

The Web Page

If node.js is used to write to the web server, this web server must be able to change dynamically. That means the web page must be defined first.

As mentioned, this web page will be able to control two switch relays (commands), receive information about the temperature and PIR sensors, and indicate whether the user armed or disarmed the system using the keypad.

The essential requirements for this page is illustrated in Figure 9-18.

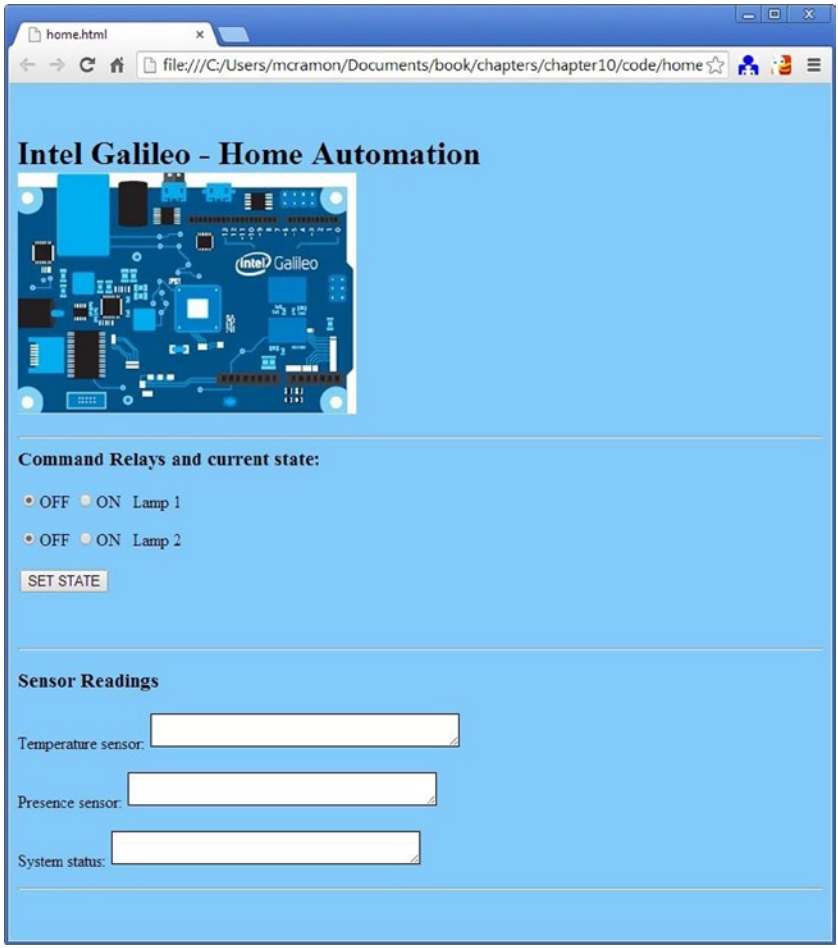

Figure 9-18. *Web page design for home automation*

The web page must be dynamic, which means the browser cannot refresh the page all the time because the sensors will send data every second and if the page is entirely refreshed the web application will be awful and will block the user interaction to send commands.

To avoid this problem, the page uses a socket connection to the web server that will be responsible for binding the socket channels.

This web page will receive information about the sensors and keypad in one message second by second in order to optimize the communication.

The message received will be very simple. Its fields are delimited by the * character, and will include the PIR sensor message, the temperatures, and whether the user armed or disarmed the system using the keypad. Something like the following:

```
"* INTRUDER * 32C - 89F * UNLOCKED *"
```

The web application must parse the string received through the socket's message and update each HTML element in the page properly.

The elements can be updated as is done with jQuery, accessing the elements using the `$()` function with the `class` name used by each element of web page. For example, you can check the `textarea` element:

```
<p>
Temperature sensor: <textarea class="txtsensor" id="temp" cols="1"
maxlength="10" name="txtsensor" readonly="readonly" style="margin: 2px;
width: 300px; height: 32px;"></textarea></p>
```

The `textarea` has a classname called `txtsensor`. To access this element using `cheerio`, you would use the following:

```
$('p .txtsensor').text('HELLO ADDING A TEXT HERE!!!');
```

To have all these functionalities, it is possible to create a web page using HTML, JavaScript, socket.io.js (the client in this case), and jQuery libraries. The HTML code is shown in Listing 9-6.

Listing 9-6. home.html

```
<html>
<head>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/
jquery.min.js"></script>
        <script src="/socket.io/socket.io.js"></script>
        <script>
                var socket = io.connect();
                socket.on('server-event-info', function (data) {
                        console.log(data);


                                var rawPIRSensorText = new String();
                                var rawTempText = new String();
                                var rawKeypadText = new String();
                                var len = data.length;

                                // extracting the sensor frame
                                var i = data.indexOf("*",1);

                                // extracting the PIR sensor data
                                rawPIRSensorText = data.substring(1, i);


                                // extracting the temp sensor data
                                var i_old = i;
                                i = data.indexOf("*",i_old+1);
                                rawTempText = data.substring(i_old+1, i);


                                // extracting the keypad sensor data
                                var i_old = i;
                                i = data.indexOf("*",i_old+1);
                                rawKeypadText = data.substring(i_old+1, i);


                                console.log(rawPIRSensorText);
                                console.log(rawTempText);
                                console.log(rawKeypadText)

                                $('p .txtsensor').text(rawTempText);
                                $('p .presencesensor').text(rawPIRSensorText);
                                $('p .systemstatus').text(rawKeypadText);

                });
        </script>
```

```
</head>
<body bgcolor="#82CAFA">
        <form method="post" name="form1" target="_self">
                         
                        <h1>
                                        Intel Galileo - Home Automation
                                        <img class="img" src="galileo.jpg">
                                        </h1>
                        <h3>
                        <hr>
                                        Command Relays and current state:</h3>
                        <p>
                                        <input checked="checked"  class="l1"
name="l1" type="radio" value="0">OFF
                                        <input class="l1" name="l1"
type="radio" value="1">ON   Lamp 1</p>
                        <p>
                                        <input checked="checked" class="l2"
name="l2" type="radio" value="0">OFF
                                        <input class="l2" name="l2"
type="radio" value="1">ON   Lamp 2</p>
                        <p>
                                        <button name="commandButton"
type="submit" value="SET STATE">SET STATE</button></p>
                        <p>
                                         </p>
                        <hr>
                        <h3>
                                        Sensor Readings</h3>
                        <p>
                                        Temperature sensor: <textarea
class="txtsensor" id="temp" cols="1" maxlength="10" name="txtsensor"
readonly="readonly" style="margin: 2px; width: 300px; height: 32px;">
</textarea></p>
                        <p>
                                        Presence sensor: <textarea
class="presencesensor" cols="1" id="sensor" maxlength="10"
name="presencesensor" style="margin: 2px; width: 300px; height: 32px;">
</textarea></p>
                        <p>
                                        System status: <textarea
class="systemstatus" cols="1" id="systemstatus" maxlength="10"
name="systemstatus" style="margin: 2px; width: 300px; height: 32px;">
</textarea></p>
                        <div>
                                        <hr>
                        </div>
        </form>
</body>

</html>
```

Note that each HTML element defines a `class` ID that will be used for access, similar to the approach used with jQuery. The `cheerios` in the web server allow you to access and change such elements similarly.

The page uses the jQuery library version 2.1.0 and invokes socket.io to implement the client that will be connected to the web server.

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js">
</script>
        <script src="/socket.io/socket.io.js"></script>
```

The JavaScript parses the string in the socket's message, separating the elements with the delimiter * and updating each element in the web page:

```
$('p .txtsensor').text(rawTempText);
$('p .presencesensor').text(rawPIRSensorText);
$('p .systemstatus').text(rawKeypadText);
```

Note that the relays can be managed through the radio buttons L1 and L2 and the web server will be notified. This is accomplished using a regular `POST` via a `submit` of the input button in an HTML form.

```
<form method="post" name="form1" target="_self">
...
...
...
<button name="commandButton" type="submit" value="SET STATE">SET
STATE</button></p>
```

The parameter `target="_self"` prevents a new page tab from being opened in your browser when the pages refreshes.

Writing the Web Server Code

Now that all the necessary packages are installed, you can write to the web server with a few lines of code.

For example if you want to develop a web server using node.js you just need the following lines of code. Suppose you created the file named `mywebserver.js` provided in this book:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World.. my first web server running on GALILEO!!!!!!\n');
}).listen(8080);
console.log('Server running and listening port 8080');
```

To run the web server, you just need to call the node passing the file as an argument. For example:

```
root@clanton:~/livro# node mywebserver.js
Server running and listening port 8080
```

And if your Intel Galileo is connected to the Internet, you can see the web page in your favorite browser using Galileo's IP number. For example, type the web address `<GALILEO IP>:8080` into your web browser.

See Figure 9-19 for an example using Chrome.

Figure 9-19. *Simple web server running node.js on Intel Galileo*

Defining the Ports

It is not the intention of this book to provide a detailed guide about how to use all the features of node.js needed to write a web server. However, the best way to understand how the web server works is to take a look at the code and discuss it.

```
var http = require("http").createServer(onRequest),
    fs = require('fs'),
    url = require('url'),
    cheerio = require('cheerio'),
    dgram = require('dgram'),

socketid = 0,
SKETCH_PORT=2000,
WEBSERVER_PORT=2010;
```

Note the definition of the port numbers to change datagrams between the sketch and the web server. The web server will listen for datagrams in the port `WEBSERVER_PORT` and will be able to send datagrams to the port `SKETCH_PORT`.

The next step is to read the page and the image used in the HTML page using cheerio and fs calls.

```
// reads the html page
var page = fs.readFileSync('home.html').toString()

// reads the image (static)
var img = fs.readFileSync('./galileo.jpg').toString("base64");

// using cheerio to transverse the page
var $ = cheerio.load(page);
    $('img').attr('src',''+img);

// getting the html string
page = $.html();
```

The static reading of the web page, in this case home.html, is loaded in the page variable and parsed by cheerio to be manipulated by the **$** variable**.**

The image used on the page is read statically because the web server will not provide methods specifically to load images when your browser asks for and changes the HTML at the same time. With node.js, it's easier to load statically and then change the properties of the img element in the HTML page to receive the image converted to base64 using cheerio, as follows:

```
 "$('img').attr('src',''+img)".
```

Creating the Sockets

It is necessary to create the UDP server using the dgram module and keep listening to the port defined by WEBSERVER_PORT in order to receive messages from sketch.

```
//
//  UDP server
//

var server = dgram.createSocket("udp4");

// this 'message' event receives the sketch datagram
server.on("message", function (msg, rinfo) {

    udp_msg = msg.toString();
    console.log("from " +
                rinfo.address + " message:" + udp_msg);

    // just bypassing the message sent by the sketch
    io.emit("server-event-info", udp_msg);

});
```

```
// this is to bind the socket
server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

server.bind(WEBSERVER_PORT);
```

The UDP server, when it receives a `message` event from the sketch, simply bypasses this event to the web page using the socket.io call:

```
io.emit("server-event-info", udp_msg);
```

The web page will provide the scripts responsible for parsing the message and updating the element dynamically in the screen. The `socketid` is the identification number received. The web page that connects to the web server and the event that contains the message to be parsed by the scripts in the web page are called `server-event-info`.

At this point we have described the code in the web server that handles message exchanges using datagrams with the sketch and how the web server sends the messages to web pages using socket.io. However, the web server needs to keep listening and perform the bind with web page as well. The following code is responsible for this task:

```
// declaring the socket.io server using the "http"
var io = require('socket.io').listen(http);

// http will listen in port 8080
http.listen(8080);

// TCP socket

io.sockets.on('connection', function (socket) {

  socketid =  socket.id;

  socket.on('client-event', function (data) {
    console.log('just to debug the connection done, ' + data.name);
  });
});
```

Note that socket.io is managed by the variable `io`, which depends on the HTTP module provided by the variable `http`. That variable opens the channel to listen to port 8080. In other words, when the browsers request the pages, that request needs to be sent through port 8080.

When the connection is establish between the browser and the socket.io server, the socket.id is saved in the `socketid` variable in order to allow messages to be repassed using the `emit` method.

Creating the GET and POST Methods

The only thing missing are the GET and POST methods in the web server. The onRequest() function is responsible for receiving the GET and POST requests.

GET doesn't do anything special because the web page was already loaded statically. The GET method simply needs to provide the page, as shown in the following code:

```
function onRequest(request, response) {
  console.log("Request received.");

  var url_parts = url.parse(request.url,true);

//
// GET methods
//
  if (request.method == 'GET') {
     console.log('Request found with GET method');

     request.on('data',function(data)
       { response.end(' data event: '+data);
       });

     if(url_parts.pathname == '/')
         // when this message is displayed your browser
         // will be able to read the HTML page.
         console.log('Showing the home.html');
         response.end(page);
  }
/
//  POST methods
//
  else if (request.method == 'POST') {
...
...
...
}
```

However, POST is a little bit tricky in this case, because when the user presses the commandButton on the web page, the form with the radio buttons that specify whether lamp 1 and lamp 2 are set to on or off is received by the web server.

When this happens, the web server must transform this information from the form into a message that's sent to the sketch, which will parse it and change the relays according to the command sent.

The idea is to send a message with a simple string that indicates the status of each lamp. "L1" is lamp 1 and "L2" is lamp 2, and "ON" and "OFF" indicate their statuses. The message also uses the & delimiter character to create a single message.

For example, the message might be:

```
"L1ON &L2OFF"
"L1OFF&L2OFF"
"L1OFF&L2ON "
"L1ON &L2ON "
```

For simplicity in the algorithm, the space characters are used to keep the messages the same size and make the logic simpler.

The POST starts by saving the form as "chunks" and then the event data is received and is accumulated by the variable's body. The "chunks" are used because the page might not be received as a single data event.

```
  else if (request.method == 'POST') {

       // the post we need to parse the L1 and L2
           // and assemble a nice message that will be received by
           // sketch UDP server
       console.log('Request found with POST method');

            // handling data received
        request.on('data', function (data) {
            body = "";
            body += data;
            console.log('got data:'+data);

        });

        request.on('end', function () {
...
...
...
}
```

When the form is received, the event named event is called with the data from the form. In this case, the states of lamps 1 and 2 are reported per their radio buttons.

The data received is a group element of the HTML element delimited by &. Each element is followed by the equals character (=).

It's necessary to extract element by element, observing the delimiter & and extract the values of each one by the =.

The easiest way do to this is to implement a utility function to parse the data received in the hash element, wherein the element name is the key and data of each element the value received.

```
hash4me = function(data){
    var firstSplits = data.split('&'),
        finalHash = [];
```

```
    // scanning first list
    for (i = 0; i < firstSplits.length; i++)
    {
        var lastSplits = firstSplits[i].split('=');
        finalHash[lastSplits[0]] = lastSplits[1];
    }
    return finalHash;
}
```

Using the utility function to "hash" the data received and accumulated in the variable body, it's possible to identify the value of each element and assemble the message that will be sent to the sketch. This is being done with the variable message in the following snippet:

```
else if (request.method == 'POST') {
...
..
..

                    var hash = hash4me(body);
                if (hash["l1"] == "0") {
...
...
...

                   // command message
                   message.write("L1OFF&");
                } else if (hash["l1"] == "1") {
                  console.log("LAMP 1 is ON");
...
...
...

                   // command message
                   message.write("L1ON &");
                }
```

The same is done with the l2 element until the message variable is ready to be sent to the sketch using the datagram, as shown in the following snippet:

```
                // informing sketch about the changes
        // this is the message sent from the web server to sketch
                server.send(message, 0, message.length, SKETCH_PORT,
"localhost", function(err, bytes) {
```

The Final Web Server Code

The final code with all details is shown in Listing 9-7.

Listing 9-7. server.js

```
var http = require("http").createServer(onRequest),
    fs = require('fs'),
    url = require('url'),
    cheerio = require('cheerio'),
    dgram = require('dgram'),
    page = "",
    body = "",
    udp_msg="",
        socketid = 0,
        SKETCH_PORT=2000,
        WEBSERVER_PORT=2010;

// reads the HTML page
var page = fs.readFileSync('home.html').toString()

// reads the image (static)
var img = fs.readFileSync('./galileo.jpg').toString("base64");

// using cheerio to transverse the page
var $ = cheerio.load(page);
    $('img').attr('src',''+img);

// getting the html string
page = $.html();

//
//  UDP server
//

var server = dgram.createSocket("udp4");

// this 'message' event receives the sketch datagram
server.on("message", function (msg, rinfo) {

    udp_msg = msg.toString();
    console.log("from " +
                rinfo.address + " message:" + udp_msg);

    // just bypassing the message sent by the sketch
    io.emit("server-event-info", udp_msg);

});
```

```
// this is to bind the socket
server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

server.bind(WEBSERVER_PORT);

//
// This function is to hash the response
//
hash4me = function(data){
    var firstSplits = data.split('&'),
        finalHash = [];

    // scanning first list
    for (i = 0; i < firstSplits.length; i++)
    {
        var lastSplits = firstSplits[i].split('=');
        finalHash[lastSplits[0]] = lastSplits[1];
    }
    return finalHash;
}

//
// Checking the GET and POST methods and
// respective responses
//

function onRequest(request, response) {
  console.log("Request received.");

  var url_parts = url.parse(request.url,true);

//
// GET methods
//
  if (request.method == 'GET') {
     console.log('Request found with GET method');

     request.on('data',function(data)
       { response.end(' data event: '+data);
       });
```

```
    if(url_parts.pathname == '/')
        // when this message is displayed your browser
        // will be able to read the HTML page.
        console.log('Showing the home.html');
        response.end(page);
  }

//
//  POST methods
//
  else if (request.method == 'POST') {

       // the post we need to parse the L1 and L2
           // and assemble a nice message that will be received by
           // sketch UDP server
       console.log('Request found with POST method');

            // handling data received
        request.on('data', function (data) {
            body = "";
            body += data;
            console.log('got data:'+data);


        });


        request.on('end', function () {

            var message = new Buffer(20);

                       message.fill(0);

            if (body != '') {

                var command = "";

               // dividing the commands to understand the state of each one
               // note in the radio buttons L1 and L2 the parameter "checked"
               // must be removed. However, we are removing it twice
                  because there
               // is a bug. Some versions of node.js and cheerio even when you
               // remove the item checked="checked", sometimes the tag checked
               // remains in the HTML element and the browser becomes confused
               //
```

```
        // $(the element).attr("checked", null);
        // $(the element).removeAttr("checked");

                          var hash = hash4me(body);
                if (hash["l1"] == "0") {

                  console.log("LAMP 1 is OFF");
                                $('input[name="l1"][value="0"]').
attr("checked", "checked");
                                $('input[name="l1"][value="1"]').
attr("checked", null);
                                $('input[name="l1"][value="1"]').
removeAttr("checked");

                    // command message
                    message.write("L1OFF&");
                } else if (hash["l1"] == "1") {
                  console.log("LAMP 1 is ON");
                        $('input[name="l1"][value="0"]').attr("checked", null);
                                $('input[name="l1"][value="0"]').
removeAttr("checked");
                                $('input[name="l1"][value="1"]').
attr("checked", "checked");

                    // command message
                    message.write("L1ON &");
                }

                             console.log("len:"  + message.toString().length);
                if (hash["l2"] == "0") {

                  console.log("LAMP 2 is OFF");
                                $('input[name="l2"][value="0"]').
attr("checked", "checked");
                                $('input[name="l2"][value="1"]').
attr("checked", null);
                    $('input[name="l2"][value="1"]').removeAttr("checked");

                    // command message
                    message.write("L2OFF",  6);

                } else if (hash["l2"] == "1") {
                  console.log("LAMP 2 is ON");
                                $('input[name="l2"][value="0"]').
attr("checked", null);
                                $('input[name="l2"][value="0"]').
removeAttr("checked");
                                $('input[name="l2"][value="1"]').
attr("checked", "checked");
```

```
                    // command message
                    message.write("L2ON ",  6);
                }

                                // informing sketch about the changes
                                // this is the message sent from web
server to sketch
                server.send(message, 0, message.length, SKETCH_PORT,
"localhost", function(err, bytes) {

                                                                if (err) {
                                                                   console.
log("Ops... some error sending UDP datagrams:"+err);
                                                                   throw err;
                                    }
                                });

                                body = "";
                            }

                  // update the page with the command
                  response.writeHead(200);
            response.end($.html());
        });
  }
}

// declaring the socket.io server using the "http"
var io = require('socket.io').listen(http);

// http will listen in port 8080
http.listen(8080);

// TCP socket

io.sockets.on('connection', function (socket) {

  socketid =  socket.id;

  socket.on('client-event', function (data) {
    console.log('just to debug the connection done, ' + data.name);
  });
});

console.log("Home automation server running...");
```

Running the Home Automation System

If all the peripherals tested okay and are connected to the Intel Galileo headers, and the sketch and web server are ready, the system is ready for testing.

Open a terminal shell and transfer the three files—`home.html`, `galileo.jpg`, and `server.js`—to a subdirectory in the home directory. In my tests, I created a subfolder called `auto`.

Then transfer the files using ftp or ssh, as explained in Chapter 5.

Remember, this project requires that you have Intel Galileo connected to a network device like WiFi, Ethernet, or an LTE modem. This example uses WiFi with Intel Centrino N-135.

The first thing to do is check the IP number you set for your device. Using the terminal shell, type `ifconfig eth0` if you are using Ethernet cables, `ifconfig wlan0` if you are using WiFi, or `ifconfig` to see all adapters:

```
root@clanton:~/auto# ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 0C:D2:92:58:F8:27
          inet addr:192.168.1.7  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::ed2:92ff:fe58:f827/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:845 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:254690 (248.7 KiB)  TX bytes:1857 (1.8 KiB)
```

Test if the connection is working using a command like `ping`:

```
root@clanton:~/auto# ping www.intel.com
PING www.intel.com (63.80.4.74): 56 data bytes
64 bytes from 63.80.4.74: seq=0 ttl=60 time=30.617 ms
64 bytes from 63.80.4.74: seq=1 ttl=60 time=29.061 ms
64 bytes from 63.80.4.74: seq=2 ttl=60 time=28.823 ms
64 bytes from 63.80.4.74: seq=3 ttl=60 time=28.025 ms
```

With the IP number in hand and the Internet connection working, go to the directory where you transferred the three files mentioned previously and type `node server.js`, as follows:

```
root@clanton:~/auto# node server.js
   info  - socket.io started
Home automation server running...
server listening 0.0.0.0:2010
```

It takes a few seconds to see the message `Home automation server running...`, which means your web server is listening to the ports and can accept connection to the socket.io and the browser and UDP datagrams exchanges between the web server and the sketch.

Using a browser of your preference, type the following address in the address bar, including your IP address:

```
http://<YOUR IP NUMBER>:8080
```

In this case, you'd type:

```
http://192.168.1.7:8080
```

You can use the browser from your personal computer or mobile phone, including Android and iPhone versions. This project was tested using Chrome, Internet Explorer 10, Firefox, Android phones 4.4.1, and iPhones 4 and 5S.

The browser will take a few seconds to connect to this page because the socket.io is in the "binding process." Remember that there is static image used in the web page as well.

If you are under a proxy you might have to set up the proxy in Intel Galileo and make sure there is no firewall blocking its connection. For example, you can export HTTP connections using `export http=<your proxy>:<your port>`.

In few seconds you will be able to see the web page shown in Figure 9-20.

Figure 9-20. *Dynamic web page is working*

The PIR and temperature sensors must be working and they must integrate with your keypad and update the web page dynamically.

The relays must switch according to your command after you press the `SET STATE` button.

If you have reached this point in this book, congrats, your system is working!

Ideas for Improving the Project

The preceding material was developed to give the reader a very basic idea of how to begin building a sample project, but obviously there is much more you can do to extend the project in the direction of your particular needs. What follows are several possibilities to consider.

Power of Ethernet (PoE)

If you are building this project with Intel Galileo generation 2 and using Ethernet to connect the board to the Internet, be sure to check out Chapter 11. You want to use energy directly from the Ethernet cable, which is more reliable.

Using express and node.js

You can change the code to support the `express` web application framework instead of the regular HTML in the node.js.

If you are building a new web page with more images and more functionalities, it can be a little harder to implement things using only the regular modules available in the standard distribution of node.js.

Express has several advantages, including how to deal with static images and not having to reinvent the wheel.

According to the tests done in this chapter, Intel Galileo was not able to install the `express` modules using the node.js 0.8.0, so it's better to update to version 0.10.25 (as explained in the section called "Updating node.js" in this chapter).

To install `express` only, type the following in your terminal shell:

```
npm install express -g
```

The successful installation creates the following modules:

```
express@3.4.8 ../node_modules/express
├── methods@0.1.0
├── merge-descriptors@0.0.1
├── range-parser@0.0.4
├── cookie-signature@1.0.1
├── fresh@0.2.0
├── debug@0.7.4
├── buffer-crc32@0.2.1
├── cookie@0.1.0
├── mkdirp@0.3.5
├── commander@1.3.2 (keypress@0.1.0)
└── send@0.1.4 (mime@1.2.11)
```

For more information about `express`, check out the link https://www.npmjs.org/package/express.

Changing the Web Page and Web Server Without Experience with the Web

If you are not familiar with jQuery, JavaScript, or the other resources used in web development, you might have some questions about how to change the web page, add new elements, and make the cheerio parse the web page so it finds the elements to be changed or updated.

My recommendation is to install a tool called `cheerio REPL` available at https://github.com/kuhnza/cheerio-repl. This tool allow you to parse the web page while your web server is running and helps you understand how to change the code.

To install `cheerio REPL`, type the following on the command line:

```
npm install -g cheerio-repl
```

After you install cheerio, it should appear in `/usr/bin/cheerio`. If your web server is not running, start it manually as you did before but let it run in the background with & and send the debug messages to a null device so they don't bother your prompt shell.

```
root@clanton:~/auto# node server.js > /dev/null &
   info  - socket.io started
Home automation server running...
server listening 0.0.0.0:2010
```

Then call the `cheerio-repl` tool using your IP address or your loopback IP and the port number. Then you can reach the cheerio prompt as follows:

```
root@clanton:~/auto# cheerio http://127.0.0.1:8080
Request received.
{}
Request found with GET method
Showing the home.html
cheerio>
```

If you type `$.html()` in the prompt, you will see the HTML content of the page, as follows:

```
cheerio> $.html()
'<html>\n        <head>\n\t\t        <script src="http://ajax.googleapis.
com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>\n                <script
src="/socket.io/socket.io.js"></script>\n\t\t\t\t<script>\n
var socket = io.connect();\n                    socket.on(\'server-event-
info\', function (data) {\n                        console.log(data);\
n\t\t\t\t\t\t \n\t\t\t\t\t\t\t   \n\t\t\t\t\t\t\t \tvar rawPIRSensorText =
new String();\n                            var rawTempText = new String(
);\n                         var rawKeypadText = new String();\t\n\t
var len = data.length;\n\n                            // extracting the
sensor frame\n\t                        var i = data.indexOf("*",1);\
n\n                        // extracting the PIR sensor data\
n\t                      rawPIRSensorText = data.substring(1, i);\n
\n  \n                          // extracting the temp sensor data\n\t
var i_old = i;\n                            i = data.indexOf("*",i_old+1);\
n\t                      rawTempText = data.substring(i_old+1, i);\n
\n  \n                          // extracting the keypad sensor data\
n\t                      var i_old = i;\n                            i
= data.indexOf("*",i_old+1);\n\t                        rawKeypadText =
data.substring(i_old+1, i);\n  \n  \n                          console.
log(rawPIRSensorText);\n                            console.log(rawTe
mpText);\n                        console.log(rawKeypadText)\n\t\
t\t\t\t\t \n\t\t\t\t\t\t\t    $(\'p .txtsensor\').text(rawTempText);\n\
t\t\t\t\t\t    $(\'p .presencesensor\').text(rawPIRSensorText);\n\t\t\
t\t\t\t    $(\'p .systemstatus\').text(rawKeypadText);\n\t\t\t\t\t\t\t
\n                    });\n                </script>\n\n        </head>\n
<body bgcolor="#82CAFA">\n                <form method="post" name="form1"
target="_self">\n                         \n                        <
h1>\n                        Intel Galileo - Home
Automation \n\t\t\t\t\t\t\t\t\t<img class="img" src="
EBAMFBwYHBwcGBwcICQsJCAgKCAcHCg0KCgsMDAwMBwkODw0MDgsMDAz/2wBDAQICAgMDAwYDAwY
MCAcIDAwMDAwMDAwMDA
```

The cheerio REPL can parse the content provided by the web server, in this case the `home.html` page. For example, suppose you wanted to parse the element used to control lamp 1, which is named `l1` in the HTML:

```
<p>
   <input checked="checked"  class="l1" name="l1" type="radio" value="0">OFF
   <input class="l1" name="l1" type="radio" value="1">ON   Lamp 1</p>
```

This radio button can assume two states. You need to type the following into the cheerio prompt:

```
cheerio> $('p .l1')
{ '0':
   { type: 'tag',
     name: 'input',
     attribs:
      { checked: 'checked',
        class: 'l1',
        name: 'l1',
        type: 'radio',
        value: '0' },
     children: [],
     prev:
      { data: '\n                                    ',
        type: 'text',
        parent: [Object],
        prev: null,
        next: [Circular] },
     next:
      { data: 'OFF\n                                  ',
        type: 'text',
        parent: [Object],
        prev: [Circular],
        next: [Object] },
     parent:
      { type: 'tag',
        name: 'p',
        attribs: {},
        children: [Object],
        prev: [Object],
        next: [Object],
        parent: [Object] } },
  '1':
   { type: 'tag',
     name: 'input',
     attribs:
      { class: 'l1',
        name: 'l1',
        type: 'radio',
        value: '1' },
     children: [],
```

```
    prev:
     { data: 'OFF\n                                    ',
       type: 'text',
       parent: [Object],
       prev: [Object],
       next: [Circular] },
    next:
     { data: 'ON   Lamp 1',
       type: 'text',
       parent: [Object],
       prev: [Circular],
       next: null },
    parent:
     { type: 'tag',
       name: 'p',
       attribs: {},
       children: [Object],
       prev: [Object],
       next: [Object],
       parent: [Object] } },
  length: 2 }
```

You can obtain the same result using the following:

```
cheerio> $('input[name="l1"]')
```

In this case, `cheerio` parsed the whole element given the attributes (`attribs`) of both states 0 and 1 as the child and parent object elements as well.

Suppose you want to check a specific attribute, such as `checked`:

```
cheerio> $('input[name="l1"][value="0"]').attr("checked")
'checked'
```

Then you want to remove this attribute:

```
cheerio> $('input[name="l1"][value="0"]').removeAttr("checked")
{ '0':
   { type: 'tag',
     name: 'input',
     attribs:
      { '0': 1,
        checked: false,
        class: 'l1',
        name: 'l1',
        type: 'radio',
        value: '0' },
     children: [],
```

```
    prev:
     { data: '\n                                ',
       type: 'text',
       parent: [Object],
       prev: null,
       next: [Circular] },
    next:
     { data: 'OFF\n                                 ',
       type: 'text',
       parent: [Object],
       prev: [Circular],
       next: [Object] },
    parent:
     { type: 'tag',
       name: 'p',
       attribs: {},
       children: [Object],
       prev: [Object],
       next: [Object],
       parent: [Object] } },
  length: 1 }
```

The value checked was removed but the attribute checked=false is still there. To remove that property, you need to set it to null:

```
cheerio> $('input[name="l1"][value="1"]').attr("checked", null);
{ '0':
   { type: 'tag',
     name: 'input',
     attribs:
      { '0': 1,
        class: 'l1',
        name: 'l1',
        type: 'radio',
        value: '1' },
     children: [],
     prev:
      { data: 'OFF\n                                 ',
        type: 'text',
        parent: [Object],
        prev: [Object],
        next: [Circular] },
     next:
      { data: 'ON   Lamp 1',
        type: 'text',
        parent: [Object],
        prev: [Circular],
        next: null },
```

```
    parent:
     { type: 'tag',
       name: 'p',
       attribs: {},
       children: [Object],
       prev: [Object],
       next: [Object],
       parent:
```

Now check the HTML contents of this element specifically:

```
cheerio> $.html('p .l1')
'<input O="1" checked class="l1" name="l1" type="radio" value="O"><input
O="1" class="l1" name="l1" type="radio" value="1">'
```

The command used to manipulate the HTML content using this tool is the same one you might use in your code. The difference is that it's much faster to check if the syntax is correct instead of starting and reloading the web server or JavaScript all the time.

Creating an Analogic Keypad and Having More I/Os Available

The keypad uses eight digital port I/Os, which is a lot compared to the 14 digital pins. If you want to connect more relay and more PIR sensors, the keypad might cause a problem.

One idea is to use an R-2R resistor ladder and one of the analog ports as the input to convert analog-to-digital signals when keys or switches are pressed.

The R-2R resistor ladder is a circuit that contains only two types of resistors. One type value is the double of the other, and they are arranged sequentially (see Figure 9-21).

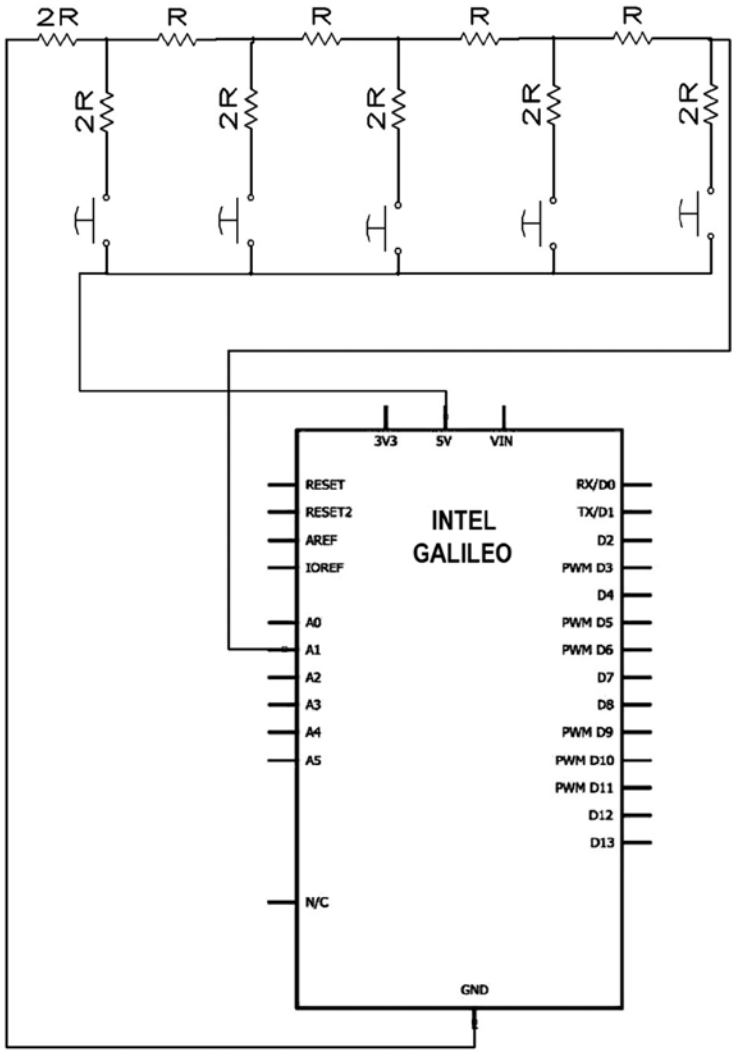

Figure 9-21. *Analog keypad with R-2R resistor ladder*

Each time a key is pressed a different voltage level is applied to input port A1. The value of resistor R is the double of resistor 2R. You can choose 10K ohm and 20k ohm, for example.

It's recommended that you use the resistor with the maximum precision possible, so try to get a resistor with a tolerance of 1% if you can.

You need to create a code in the sketch that can read the port A1, test the maximum and minimum reading of each key, and convert them to a correspondent event. You can reach such values experimentally pressing the keys and checking the maximum and minimum reading. You then add your code or resolve the Thevenins theorem for each condition, following the standard electronic theory.

Adding a Username and Password

The project hasn't incorporated a username or password, which means anyone could potentially access your system once they discover the IP.

There are several ways to implement this protection. Due to the simplicity of the code provided by the framework and the diversity in terms of methods for authenticating the user, I recommend that you read the passport found at https://github.com/jaredhanson/passport.

Using the DHT11 Sensor

In this project the TMP36 sensor was used to measure the temperature in the environment. However, if you are interested in monitoring the humidity instead, you can use the DHT11 sensor. For more details on how to use this sensor, read the section called "Project - DHT Sensor Library with Fast I/O APIs" in Chapter 4.

Summary

In this chapter you learned how Intel Galileo communicates with different sensors, including the integration of sketches with a keypad.

You also learned how to use Linux to create a web server using node.js, which is different from other Arduino boards based on microcontrollers. Thus, by using a web client, you can control and monitor the sensor connected to Intel Galileo. This enables you to control and monitor your house when you're away.

This project provides very basic functionality in terms of home automation, but it is a starting point from which you can expand the system and make it more sophisticated.