BeagleBone Home Automation Blueprints

BeagleBone is a microboard PC that runs Linux. It can connect to the Internet and can run various OSes such as Android and Ubuntu. BeagleBone is used for a variety of different purposes and projects, from simple projects such as building a thermostat to more advanced ones such as home security systems.

Packed with real-world examples, this book will provide you with how to connect several sensors and actuators to the BeagleBone Black. You'll learn how to give BeagleBone access to them in order to realize simple-to-complex monitoring and control systems that will help you take control of your house. You will also find software examples of implementing web interfaces using the classic PHP/HTML/TM+ pair with JavaScript, and using complex APIs to interact with Google Docs account, WhatsApp, Facebook, or Twitter.

Who this book is written for
This book is for developers who know how to use BeagleBone and are just above the beginner level. If you want to learn how to get access to a Linux device driver to collect data from a peripheral or to control a device, this is the book for you.

What you will learn from this book
- Build a carbon monoxide (and other gases) sensor with a buzzer/LED alarm to signal high concentrations
- Log environment data and plot it in a fancy manner
- Develop a simple web interface with a LAMP platform
- Prepare complex web interfaces in JavaScript and get to know how to stream video data from a webcam
- Use APIs to get access to a Google Docs account or a WhatsApp/Facebook/Twitter account to manage a home automation system
- Add custom device drivers to manage a blinking LED or an infrared remote controller
- Discover how to work with electronic components to build small circuits
- Use an ultrasonic distance sensor, a temperature/humidity sensors and other peripherals to monitor and control your surroundings

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 *Dangerous Gas Sensors*
- A synopsis of the book’s content
- More information on **BeagleBone Home Automation Blueprints**
About the Author

Rodolfo Giometti is an engineer, IT specialist, and GNU/Linux expert. He has over 15 years of experience working with platforms based on x86 (PC), ARM, MIPS, and PowerPC. He is the first author and a maintainer of the LinuxPPS project (Linux’s pulse per second subsystem.) He actively contributes to the Linux source code with several patches and new device drivers for several devices. His areas of core expertise are in writing device drivers for new peripherals, realizing special Linux ports for several custom embedded boards, and developing control automation and remote monitoring systems. He is the cofounder of the Cosino Project, where he develops software for industry control automation systems. He is also the co-CEO at HCE Engineering S.r.l., a leading industrial embedded systems-based company.
The BeagleBone Black is an embedded system that is able to run an embedded GNU/Linux distribution as well as normal (and powerful) distributions like Debian or Ubuntu, and to which the user can connect several external peripherals via two dedicated expansion connectors.

Because it has powerful distribution capabilities with an easily expandable embedded board, the BeagleBone Black system is a state-of-the-art device that allows the user to build powerful and versatile monitoring and controlling applications.

This book presents several home automation prototypes in both hardware and software in order to explain to you how we can use the BeagleBone Black board connected with several devices to control your home.

Each prototype is discussed in its respective chapter in both hardware and software, explaining all the connections and the software packages necessary to manage several peripherals. Then, the code to glue all of it together is explained in detail till the final test of each project.

The hardware devices used in this book have been chosen in order to cover all the possible connection types we can encounter while working with a BeagleBone Black board, so you will find I2C, SPI, USB, 1-Wire, serial, and digital and analog devices.

The programming languages used in this book have been chosen according to the rule to find the quickest and easiest solution to solve the current problem; in particular, you can find example code in Bash, C, PHP, Python, HTML, and even JavaScript.
Warning! All the projects in this book are prototypes and cannot be used as final applications.
Neither the author of this book nor Packt Publishing recommend or endorse these products to be used alone or as a component in any application without the necessary modifications to turn these prototypes into final products.
Neither the author of this book nor Packt Publishing will be held liable for unauthorized use of these prototypes. The user can use both the hardware and software of these devices at their own risk!

In the chapters where we will need to use a daemon or a kernel module, or where we will need to recompile the whole kernel, I've added a short description about what the reader should do and where they can get more information regarding the tools used; however, some basic skills in managing a GNU/Linux system, the kernel's modules, or the kernel itself are required (the reader can take a look at the book BeagleBone Essentials, Packt Publishing, written by the author of this book, in order to have more information about these topics).

**What this book covers**

*Chapter 1, Dangerous Gas Sensors,* will show how to use the BeagleBone Black to monitor some dangerous gases such as CO, Methane, LPG, and so on in a room and how to enable an acoustic and visual alarm in case of danger. Also, by using a GSM module, the user will be able to send an SMS message to a predefined phone number to alert, for instance, a relative.

*Chapter 2, Ultrasonic Parking Assistant,* will show how to use the BeagleBone Black to implement a park assistant. We will use an ultrasonic sensor to detect the distance between our car and the garage wall, and some LEDs to give feedback to the driver about the car's position in order to avoid collisions.

*Chapter 3, Aquarium Monitor,* will show how to make an aquarium monitor through which we'll be able to record all the environment data as well as control the life of our beloved fish from a web panel on our PC, smartphone, or tablet.

*Chapter 4, Google Docs Weather Station,* will take a look at a simple weather station that can also be used as an Internet-of-Things device. This time, our BeagleBone Black will collect environment data and will send them to a remote database (a Google Docs spreadsheet) in order to be reworked and presented into a shared environment.

*Chapter 5, WhatsApp Laundry Room Monitor,* will present an implementation of a laundry monitor room with several sensors capable of alerting the user directly on their WhatsApp account when a specific event occurs.
Chapter 6, Baby Room Sentinel, will present a possible implementation of a baby room sentinel capable of monitoring the room by detecting if the baby is crying or if the baby is actually breathing during sleep. Also, as a special feature, the system will be able to measure the baby’s temperature with a contactless temperature sensor.

Chapter 7, Facebook Plant Monitor, will show how to implement a plant monitor capable of measuring light, soil moisture, and soil temperature (in the soil and outside it), and how to take some pictures, at specific intervals, via a webcam, and then publishing them on a Facebook timeline.

Chapter 8, Intrusion Detection System, will show how we can implement a low cost intrusion detection system with a reasonable quality level by using our BeagleBone Black and two (or more) webcams. The system will be able to alert the user by sending an e-mail message with a photo of the intruder.

Chapter 9, Twitter Access Control System with Smart Card and RFID, will show how to use a smart card reader as well as two kinds of RFID reader (LF and UHF) in order to show different ways to implement a minimal identification system for access control that can send an alert message to a Twitter account.

Chapter 10, A Lights Manager with a TV Remote Controller, will show how to manage a simple on/off device connected to our BeagleBone Black by using a TV remote controller or any infrared capable device.

Chapter 11, A Wireless Home Controller with Z-Wave, will present how to implement a little wireless home controller by using a Z-Wave controller connected to our BeagleBone Black and two Z-Wave devices: a wall plug and a multisensor device.
In this chapter, we will learn how to use the **BeagleBone Black** to monitor some dangerous gases in a room, such as CO, methane, LPG, and so on, and then enabling an acoustic and visive alarm in case of danger. Also, by using a GSM module, the user will be able to send an SMS message to a predefined phone number to alert, for instance, a relative.

Also, the user will be able to control, log, and display the measured concentrations from the system console/terminal by using a command-line interface (this to keep the code simple).

We'll see how to build the circuitry to manage the sensors and how to get the gas concentration from them. Then, we'll take a look at how to manage a GSM module in order to send SMS messages.

**The basic of functioning**

In this project, our BeagleBone Black will periodically read the environmental data from the sensors, comparing them with user selectable ranges, and then generate an alarm in case one (or more) data read is out of that range.

The sensors will be connected to the BeagleBone Black ADCs with a dedicated circuitry, while the alarms will be enabled with dedicated GPIO lines. Then a GSM module will be connected to our BeagleBone Black's serial port in order to send other alarm messages via SMS.
Dangerous Gas Sensors

Setting up the hardware
As just stated, all devices are connected with the BeagleBone Black, which is the real core of the system, as shown in the following screenshot:

The data flow is from the sensors to the alarm actuators (LED, buzzer, and GSM module) and the user will be able to send commands to the system, or check the system status and the collected data, by using the system console.

Connecting the gas sensors
The gas sensors are used to monitor the environment and we can choose different kinds of such devices. I decided to use the ones shown in the following screenshot due to the fact they act as a variable resistor according to the gas concentration, so they can be easily read with a normal ADC:
In the prototype presented here, the gas sensors are actually four, but the ones named **MQ-2** (smoke detector), **MQ-4** (methane detector), and **MQ-7** (LPG detector) look very similar to each other (except the label on each sensor), so I reported only one of them in the preceding screenshot, while the carbon monoxide detector is the red device labeled with MQ-7.

The devices can be purchased at the following links (or by surfing the Internet):

- MQ-7: [http://www.cosino.io/product/mq-7-gas-sensor](http://www.cosino.io/product/mq-7-gas-sensor)

The following are the URLs where we can get the datasheet for each GAS sensor:


Looking carefully at the datasheet of the gas sensors, we can see exactly how these sensors’ class varies their internal resistance according to the gas concentration (in reality, it depends on environment humidity and temperature too; but for an indoor functioning, we can consider these values as constants). So, if we put it in series with a resistor and apply a constant voltage. We can get an output voltage that is proportional to the actual gas concentration.

The following diagram shows a possible schematics where the gas sensor is connected to **5V** power supply and the **RL** resistor is formed by two resistors (**R1** & **R2**) due to the fact we cannot put more than **1.8V** at a BeagleBone Black's ADC pin. So, by choosing these two resistors in such a way that **R1 ≥ 2*R2**, we can be sure we have no more than **5.0V/3 ≈ 1.67V** at the ADC input pin on every possible functioning condition, even if the sensor’s internal resistance is shorted. However, to be completely sure we can add a **Zener diode (Z)** with a reverse threshold on **1.8V** (but I didn't use it on my prototype).
The following diagram shows the circuitry I used to connect each sensor:

Note that the GAS sensors have six pins labeled in pairs as A, B, and H; while the A and B pair pins are shortened, the H labeled pairs must be connected at one end to the input voltage (5V in our case) and the other end to the GND (see the datasheet for further information).

Another important issue regarding these sensors is the calibration we should perform before using them. This last adjustment is very important; as reported in the MQ-2 datasheet, we read the following recommendation:

*We recommend that you calibrate the detector for 1000 ppm liquefied petroleum gas (LPG), or 1000ppm iso-butane (i-C\(_4\)H\(_{10}\)) concentration in air and use value of load resistance that (RL) about 20K (5K to 47K).*

This step can be done by replacing resistors R1 or R2 with a varistor and then fine tuning its resistance. However, I decided to use normal resistors \(R1 = 15K\Omega, R2 = 6.8K\Omega\) in such a way that \(RL = R1 + R2 \approx 20K\Omega\), as suggested by the datasheet) and then implemented a little translation in software (see the following section), that is, we can translate raw data from the ADCs into a ppm (part-per-million) value in such a way the user can work with physic data.

This translation can be done using a gain and an offset value in the following formula for each sensor:

* \(ppm = raw \ast gain + offset\)
During the calibration procedure, we just need to use two known points \((ppm1\) and \(ppm2\)), read the corresponding raw data \((raw1\) and \(raw2\)), and then apply the following formulas:

\[
\begin{align*}
gain &= \frac{ppm1 - ppm2}{raw1 - raw2} \\
offset &= ppm1 - raw1 \times gain
\end{align*}
\]

Of course, we need four gain/offset couples, one per sensor (the calibration procedure is quite long!)

Once we have fixed the input circuits, we simply have to connect each \(Vout\) to the BeagleBone Black's ADC input pins. Our board has 8 ADCs inputs, so we can use the following connections:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Gas sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>P9.39 - AIN0</td>
<td>Vout @MQ-2</td>
</tr>
<tr>
<td>P9.37 - AIN2</td>
<td>Vout @MQ-4</td>
</tr>
<tr>
<td>P9.35 - AIN6</td>
<td>Vout @MQ-5</td>
</tr>
<tr>
<td>P9.33 - AIN4</td>
<td>Vout @MQ-7</td>
</tr>
</tbody>
</table>

To enable them, we use the following command:

```
root@beaglebone:~# echo cape-bone-iio > /sys/devices/bone_capemgr.9/slots
```

If everything works well, we should get the following kernel messages:

```
part_number 'cape-bone-iio', version 'N/A'
slot #7: generic override
bone: Using override eeprom data at slot 7
slot #7: 'Override Board Name,00A0,Override Manuf,cape-bone-iio'
slot #7: Requesting part number/version based 'cape-bone-iio-00A0.dtbo'
slot #7: Requesting firmware 'cape-bone-iio-00A0.dtbo' for board-name 'Override Board Name', version '00A0'
slot #7: dtbo 'cape-bone-iio-00A0.dtbo' loaded; converting to live tree
slot #7: #1 overlays
helper.12: ready
slot #7: Applied #1 overlays.
```
Then, the files AIN0, AIN1, ..., AIN7 should become available as follows:

```
root@beaglebone:~# find /sys -name '*AIN*'
/sys/devices/ocp.3/helper.12/AIN0
/sys/devices/ocp.3/helper.12/AIN1
/sys/devices/ocp.3/helper.12/AIN2
/sys/devices/ocp.3/helper.12/AIN3
/sys/devices/ocp.3/helper.12/AIN4
/sys/devices/ocp.3/helper.12/AIN5
/sys/devices/ocp.3/helper.12/AIN6
/sys/devices/ocp.3/helper.12/AIN7
```

These settings can be done using the `bin/load_firmware.sh` script in the book's example code repository, as follows:

```
root@beaglebone:~# ./load_firmware.sh adc
```

Then, we can read the input data by using the `cat` command:

```
root@beaglebone:~# cat /sys/devices/ocp.3/helper.12/AIN0
1716
```

Note that the ADC can also be read by using other files still into the `sysfs` filesystem. The following command, for instance, reads from AIN0 input pin:

```
root@beaglebone:~# cat /sys/bus/iio/devices/iio:device0/in_voltage0_raw
```

**Connecting the alarm actuators**

Now, we have to connect the alarm actuators in such a way the user can have a visual and acoustic feedback of any possible dangerous gas concentration. Also, we have to connect the GSM module to a serial port to communicate with it.
LED and buzzer

The LED and buzzer connections are very simple. The LEDs can be directly connected (with a resistor) with the BeagleBone Black's GPIO pins without problems, while the buzzer needs a little more work due to the fact that it needs a higher current than the LED to work. However, we can resolve the problem by using a transistor as shown in the following diagram to manage the buzzer with a higher current.

Note that the buzzer can't be a simple piezo without an internal oscillator, otherwise an external oscillator circuit or a PWM signal must be used!

In my circuitry, I used an R (470Ω) resistor for the LED (L), an R (2KΩ), Rd (4.7KΩ) resistors for the buzzer, and a BC546 transistor (T). Note that, regarding the LEDs, an R = 100Ω resistor can result in a higher brightness, so you may change it according to the LED color to have different results.

Note also that the resistor Rd in the buzzer circuitry is needed to pull-down the GPIO during the boot. In fact, during this stage it is set as input, and even in such configuration the current that flows out from the pin can turn on the buzzer!

The BeagleBone Black has a lot of GPIOs lines, so we can use the following connections:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Actuator</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8.9 - GPIO69</td>
<td>R @LED</td>
</tr>
<tr>
<td>P8.10 - GPIO68</td>
<td>R @Buzzer</td>
</tr>
</tbody>
</table>
Now, to test the connections, we can set up the GPIOs by exporting them and then setting these lines as outputs with the following commands:

```
root@beaglebone:~# echo 68 > /sys/class/gpio/export
root@beaglebone:~# echo out > /sys/class/gpio/gpio68/direction
root@beaglebone:~# echo 0 > /sys/class/gpio/gpio68/value
root@beaglebone:~# echo 69 > /sys/class/gpio/export
root@beaglebone:~# echo out > /sys/class/gpio/gpio69/direction
root@beaglebone:~# echo 0 > /sys/class/gpio/gpio69/value
```

Note that it will be a good idea to use blinking LEDs to do this job. However, for this first chapter I'm going to use normal GPIO lines, leaving this topic for the following chapters.

Now, to turn on and off both the LED and the buzzer, we simply write 1 or 0 into the proper files, as follows:

```
root@beaglebone:~# echo 1 > /sys/class/gpio/gpio68/value
root@beaglebone:~# echo 0 > /sys/class/gpio/gpio68/value
root@beaglebone:~# echo 1 > /sys/class/gpio/gpio69/value
root@beaglebone:~# echo 0 > /sys/class/gpio/gpio69/value
```

These settings can be done by using the `bin/gpio_set.sh` script in the book's example code repository, as follows:

```
root@beaglebone:~# ./gpio_set 68 out
root@beaglebone:~# ./gpio_set 69 out
```

**GSM module**

As stated in the introduction of this chapter, we wish to add a GSM module to be able to alert the user remotely too. In order to do this, we can connect this device with a normal serial port with TTL level signals. In this case, we have only to choose one of the serial ports available on our BeagleBone Black.
The following screenshot shows the GSM module I decided to use:

![GSM module image]

The device can be purchased at the following link (or by surfing the Internet):
http://www.cosino.io/product/serial-gsmgprs-module


The BeagleBone Black has four available serial ports. By deciding to use the device /dev/ttyO1, we can use the following connections:

<table>
<thead>
<tr>
<th>Pin</th>
<th>GSM module</th>
</tr>
</thead>
<tbody>
<tr>
<td>P9.24 - TX-O1</td>
<td>RX</td>
</tr>
<tr>
<td>P9.26 - RX-O1</td>
<td>TX</td>
</tr>
<tr>
<td>P9.1 - GND</td>
<td>GND</td>
</tr>
<tr>
<td>P9.3 - 3.3V</td>
<td>3.3V</td>
</tr>
<tr>
<td>P9.5 - 3.3V</td>
<td>5V</td>
</tr>
</tbody>
</table>

To enable the serial port, we have to use the following command:

```
root@beaglebone:-# echo BB-UART1 > /sys/devices/bone_capemgr.9/slots
```
Dangerous Gas Sensors

If everything works well, we should get the following kernel messages:

part_number 'BB-UART1', version 'N/A'
	slot #8: generic override

bone: Using override eeprom data at slot 8
	slot #8: 'Override Board Name,00A0,Override Manuf,BB-UART1'
	slot #8: Requesting part number/version based 'BB-UART1-00A0.dtbo'
	slot #8: Requesting firmware 'BB-UART1-00A0.dtbo' for board-name 'Override Board Name', version '00A0'

slot #8: dtbo 'BB-UART1-00A0.dtbo' loaded; converting to live tree

slot #8: #2 overlays

48022000.serial: tty01 at MMIO 0x48022000 (irq = 73) is a OMAP UART1

slot #8: Applied #2 overlays.

The device file /dev/tty01 should now become available.

These settings can be done by using the bin/load_firmware.sh script in the book's example code repository, as follows:

```bash
root@beaglebone:~# ./load_firmware.sh tty01
```

To verify that the new device is ready, we can use the ls command as follows:

```bash
root@beaglebone:~# ls -l /dev/tty01
```

The reader can take a look at the book BeagleBone Essentials, Packt Publishing, which was written by the author of this book, in order to have more information regarding how to activate and use the GPIO lines and the serial ports available on the system.

Now, we can test whether we actually talk with the modem by using the screen command as follows:

```bash
root@beaglebone:~# screen /dev/tty01 115200
```

The screen command can be installed by using the aptitude command as follows:

```bash
root@beaglebone:~# aptitude install screen
```
After pressing the *ENTER* key, you should get a blank terminal where, if you enter the *ATZ* string, you should get the string *OK* as answer, as shown in the following code:

```
ATZ
OK
```

It's the GSM module that answers that it's okay and fully functional. To quit from the *screen* command, you have to enter the *CTRL + A + \* keys sequence and then answer *yes* by pressing the *y* key when the program asks you to *Really quit and kill all your windows [y/n]*.

**The final picture**

Well, now we have to put it all together! The following image shows the prototype I made to implement this project and to test the software:

![Prototype Image](image)

Note that we need an external power supplier due to the fact that the external circuitry (and especially the GSM module) needs the 5V power supply.
Dangerous Gas Sensors

Setting up the software

Now, it’s time to think about the software needed to implement the desired functioning, that is, checking the gas concentrations, logging them, and eventually activating the alarms. We need the following:

1. A periodic procedure (read_sensors.php) that periodically scans all the sensors and then logs their data into a database.
2. A periodic procedure (monitor.php) that reads the sensors’ data, checks them against preset thresholds, and then sets some internal status.
3. A periodic procedure (write_actuators.php) that enables the alarms according to the previously saved status.

The following diagram shows the situation:

![Diagram showing the flow of data between read_sensors.php, monitor.php, write_actuators.php, and the database]

The core of the system is the database, where we store both the data we wish to log and the system’s status. In this manner, all periodic functions can be realized as separate tasks that talk to each other by using the database itself. Also, we can control all the tasks from the system console by just altering the config table at runtime.

I used MySQL to implement the database system, and the preceding configuration can be created by using the my_init.sh script, where we define the proper tables.
The MySQL daemon can be installed by using the **aptitude** command as follows:

```
root@beaglebone:~# aptitude install mysql-client mysql-server
```

Here is a snippet of the script:

```sql
CREATE TABLE status (
    n VARCHAR(64) NOT NULL,
    v VARCHAR(64) NOT NULL,
    PRIMARY KEY (n)
) ENGINE=MEMORY;

# Setup default values
INSERT INTO status (n, v) VALUES('alarm', 'off');

# # Create the system configuration table
#

CREATE TABLE config (
    n VARCHAR(64) NOT NULL,
    v VARCHAR(64) NOT NULL,
    PRIMARY KEY (n)
);

# Setup default values
INSERT INTO config (n, v) VALUES('sms_delay_s', '300');

INSERT INTO config (n, v) VALUES('mq2_gain', '1');
INSERT INTO config (n, v) VALUES('mq4_gain', '1');
INSERT INTO config (n, v) VALUES('mq5_gain', '1');
INSERT INTO config (n, v) VALUES('mq7_gain', '1');
INSERT INTO config (n, v) VALUES('mq2_off', '0');
INSERT INTO config (n, v) VALUES('mq4_off', '0');
INSERT INTO config (n, v) VALUES('mq5_off', '0');
INSERT INTO config (n, v) VALUES('mq7_off', '0');

INSERT INTO config (n, v) VALUES('mq2_th_ppm', '2000');
INSERT INTO config (n, v) VALUES('mq4_th_ppm', '2000');
INSERT INTO config (n, v) VALUES('mq5_th_ppm', '2000');
INSERT INTO config (n, v) VALUES('mq7_th_ppm', '2000');
```
# Create one table per sensor data
#

CREATE TABLE MQ2_log (  
    t DATETIME NOT NULL,  
    d float,  
    PRIMARY KEY (t)  
);

CREATE TABLE MQ4_log (  
    t DATETIME NOT NULL,  
    d float,  
    PRIMARY KEY (t)  
);

CREATE TABLE MQ5_log (  
    t DATETIME NOT NULL,  
    d float,  
    PRIMARY KEY (t)  
);

CREATE TABLE MQ7_log (  
    t DATETIME NOT NULL,  
    d float,  
    PRIMARY KEY (t)  
);

The `my_init.sh` script is stored in the `chapter_01/my_init.sh` file in the book's example code repository.

The reader should notice that we define a `status` table with the `MEMORY` storage engine since we don't need to preserve it at reboot but need a good performance in accessing it, while the `config` table and the per-sensor logging tables (`MQ2_log`, `MQ4_log`, `MQ5_log`, and `MQ7_log`) are defined as normal tables since we need to save these data even during a complete restart. Note that we defined one table per variable in order to easily get access to the logged data; however, nothing changes, even if we decide to keep the logged data into a global logging table.
Note also that during the database initialization, we can define some default settings by simply recording these values by using an `INSERT` command. For the `status` table, we just need the alarm variable to be set to `off`, while into the `config` table, we can set up the minimum delay in seconds (`sms_delay_s`) to wait before resending a new SMS alarm, the gain/offset translation couple variables (`mq2_gain/mq2_off` and friends), and the per-sensor threshold variables (`mq2_th_ppm` and friends) to be used to activate the alarms.

## Managing the ADCs

Now, to get data from the ADC and save them into the database, we have to write a periodic task. This is quite easy and the following code snippet shows a PHP implementation of the main function of the file `read_sensors.php`, which does this:

```php
function daemon_body() {
    global $loop_time;
    global $sensors;

    # The main loop
dbg("start main loop (loop_time=${loop_time}s)");
    while (sleep($loop_time) == 0) {
        dbg("loop start");

        # Read sensors
        foreach ($sensors as $s) {
            $name = $s['name'];
            $file = $s['file'];
            $var = $s['var'];
            $log = $s['log'];

            # Get the converting values
            $gain = db_get_config($var . "_gain");
            $off = db_get_config($var . "_off");

            dbg("gain[$var]=$gain off[$var]=$off");

            # Read the ADC file
            $val = file_get_data($file);
            if ($val !== false) {
                err("unable to read sensor $name");
                continue;
            }
        }
    }
}
```
The function is quite simple. It starts the main loop to periodically read the ADC data, get the gain and offset conversion values for the current variable needed to convert it into the corresponding ppm number, then alters the current status variables, and adds a new value into the logging table of the read sensor.

If we execute the script enabling all debugging command-line options, we get:

```
root@beaglebone:# ./read_sensors.php -d -f -l -T 5
read_sensors.php[5388]: signals traps installed
read_sensors.php[5388]: start main loop (loop_time=5s)
read_sensors.php[5388]: loop start
read_sensors.php[5388]: gain[mq2]=0.125 off[mq2]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN0 val=810 ppm=101.25
read_sensors.php[5388]: gain[mq4]=1 off[mq4]=0
```

The complete script is stored in the `chapter_01/read_sensors.php` file in the book's example code repository.
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN2 val=1477
ppm=1477
read_sensors.php[5388]: gain[mq5]=1 off[mq5]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN6 val=816
ppm=816
read_sensors.php[5388]: gain[mq7]=1 off[mq7]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN4 val=572
ppm=572
read_sensors.php[5388]: loop end
read_sensors.php[5388]: loop start
read_sensors.php[5388]: gain[mq2]=0.125 off[mq2]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN0 val=677
ppm=84.625
read_sensors.php[5388]: gain[mq4]=1 off[mq4]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN2 val=1456
ppm=1456
read_sensors.php[5388]: gain[mq5]=1 off[mq5]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN6 val=847
ppm=847
read_sensors.php[5388]: gain[mq7]=1 off[mq7]=0
read_sensors.php[5388]: file=/sys/devices/ocp.3/helper.12/AIN4 val=569
ppm=569
read_sensors.php[5388]: loop end
...

Note that only the first sensor has been (more or less) calibrated!

The process can be stopped as usual with the CTRL + C sequence.

Now, we can read the system status (in this case, the last read sensors datum) by using the my_dump.sh script, as follows:

root@beaglebone:~# ./my_dump.sh status
n  v
alarm  off
mq2  84.625
mq4  1456
mq5  815
mq7  569
The my_dump.sh script is stored in the chapter_01/my_dump.sh file in the book's example code repository.

The same script can be used to dump a logging table too. For instance, if we wish to see the MQ-2 logged data, we can use the following command:

```bash
root@beaglebone:~# ./my_dump.sh mq2_log
t  v
2015-05-15 17:39:36 101.25
2015-05-15 17:39:41 84.625
2015-05-15 17:39:46 84.625
```

### Managing the actuators

When a sensor detects a dangerous gas concentration, the alarm status variable is set to on state. Therefore, when this happens, we have to turn both the LED and the buzzer on, and we must send an SMS message to the user's predefined number.

In order to do these actions, we have to properly set up the GPIO lines that manage the LED and the buzzer as shown previously, and then we have to talk with the GSM module through the serial port to send the SMS message. To do this last step, we have to to install the gsm-utils package where we can find the gsmsendsms command, which is used to actually send the SMS. In order to install the package, we use the following command:

```bash
root@beaglebone:~# aptitude install gsm-utils
```

Then, after placing a functioning SIM into the module, we can verify to be able to talk with the GSM module with the gsmctl command, as shown in the following code:

```bash
root@beaglebone:~# gsmctl -d /dev/ttyO1 me
<ME0>  Manufacturer: Telit
<ME1>  Model: GL865-QUAD
<ME2>  Revision: 10.00.144
<ME3>  Serial Number: 356308042878501
```

Then, we can verify the current PIN status by using the following command:

```bash
root@beaglebone:~# gsmctl -d /dev/ttyO1 pin
<PIN0> READY
```
The preceding message shows us that the GSM module is correctly configured and the SIM in it is ready to operate; however, the SIM must be enabled by inserting the proper PIN number if we get the following message:

```
gsmendsms[ERROR]: ME/TA error 'SIM PIN required' (code 311)
```

In this case, we must use the following command:

```
root@beaglebone:~# gsmctl -d /dev/ttyO1 -I "+cpin=NNNN"
```

In the preceding command, \texttt{NNNN} is the PIN number of your SIM. If the command hangs with no output at all, it means that the connection is wrong.

Now that we've checked the connection and the SIM is enabled, we can start to send SMS messages by using the following command:

```
root@beaglebone:~# gsmendsms -d /dev/ttyO1 "+NNNNNNNNNNNN" 'Hello world!'
```

In the preceding command, the \texttt{NNNNNNNNNNNN} string is the number where the SMS must be sent.

| If the module answers as follows it means that **SMS Service Centre Address (SCA)**; which is the phone number of the centre that is accepting SMS for delivery is not set correctly in your phone: |
|---|---|
| \texttt{gsmendsms[ERROR]: ME/TA error 'Unidentified subscriber' (code 28)} |

In this case, you should ask to your GSM operator and then try the following command:

```
root@beaglebone:~# gsmctl -o setsca "+SSSSSSSSSS"
```

In the preceding command, the \texttt{SSSSSSSSSS} string is the number of your centre.

Okay, now we have all the needed information to control our actuators. A possible implementation of main function of the managing task is as follows:

```bash
function daemon_body()
{
    global $loop_time;
    global $actuators;

    $sms_delay = db_get_config("sms_delay_s");
```
Dangerous Gas Sensors

```php
$old_alarm = 0;
$sms_time = strtotime("1970");

# The main loop
dbg("start main loop (loop_time=${loop_time}s)");
while (sleep($loop_time) == 0) {
dbg("loop start");

# Get the "alarm" status and set all alarms properly
$alarm = db_get_status("alarm");
foreach ($actuators as $a) {
    $name = $a['name'];
    $file = $a['file'];

    dbg("file=$file alarm=$alarm");
    $ret = gpio_set($file, $alarm);
    if (!$ret)
        err("unable to write actuator $name");
}

# Send the SMS only during off->on transition
if ($alarm == "on" && $old_alarm == "off" &&
    strtotime("-$sms_time seconds") > $sms_delay) {
    do_send_sms();
    $sms_time = strtotime("now");
}

$old_alarm = $alarm;

dbg("loop end");
}
```

The complete script is stored in the chapter_01/write_actuators.php file in the book's example code repository.
Again, the function is really simple—we simply have to read the current `alarm` variable status from the database and then set up the actuators according to it. Note that a special job must be done for the SMS management; in fact, the system must send one SMS at time and only during the off-to-on transition and not before `sms_delay` seconds. To do the trick, we use the `old_alarm` and `sms_time` variables to save the last loop status.

To test the code, we can control the `alarm` variable by using the `my_set.sh` command as follows:

```
root@beaglebone:~# ./my_set.sh status alarm on
root@beaglebone:~# ./my_set.sh status alarm off
```

The script is stored in the `chapter_01/my_set.sh` file in the book's example code repository.

So, let's start the script with the command:

```
root@beaglebone:~# ./write_actuators.php -d -f -l -T 5
write_actuators.php[5474]: signals traps installed
write_actuators.php[5474]: start main loop (loop_time=5s)
write_actuators.php[5474]: loop start
write_actuators.php[5474]: file=/sys/class/gpio/gpio68 alarm=off
write_actuators.php[5474]: file=/sys/class/gpio/gpio69 alarm=off
write_actuators.php[5474]: loop end
write_actuators.php[5474]: loop start
write_actuators.php[5474]: file=/sys/class/gpio/gpio68 alarm=off
write_actuators.php[5474]: file=/sys/class/gpio/gpio69 alarm=off
write_actuators.php[5474]: loop end
```

On another terminal, we can change the `alarm` variable, as already stated, by using the following command:

```
root@beaglebone:~# ./my_set.sh status alarm on
```

After this we notice that the script does its job:

```
write_actuators.php[5474]: loop start
write_actuators.php[5474]: file=/sys/class/gpio/gpio68 alarm=on
write_actuators.php[5474]: file=/sys/class/gpio/gpio69 alarm=on
write_actuators.php[5474]: send SMS...
write_actuators.php[5474]: loop end
```
Regarding how to send an SMS message in PHP, I simply used the following code:

```php
function do_send_sms()
{
    dbg("send SMS...");
    system('gsmsendsms -d /dev/ttyO1 "' . PHONE_NUM . ' "GAS alarm!"');
}
```

Basically, here we use the `system()` function to call the `gsmsendsms` command.

You may note that `gsmsendsms` takes a while to send the SMS. It's normal.

### Controlling the environment

Now, we only need the glue between the sensors and actuators managing tasks, that is, a periodic function that according to the user inputs periodically checks whether the alarms must be activated according to the information read, or not.

A possible implementation of the main function of the `monitor.php` script is as follows:

```php
function daemon_body()
{
    global $loop_time;
    global $actuators;

    # The main loop
    dbg("start main loop (loop_time=${loop_time}s)");
    while (sleep($loop_time) == 0) {
        dbg("loop start");

        # Get the gas concentrations and set the "alarm" variable
        $mq2 = db_get_status("mq2");
        $mq2_th_ppm = db_get_config("mq2_th_ppm");
        dbg("mq2/mq2_th_ppm=$mq2/$mq2_th_ppm");
        $mq4 = db_get_status("mq4");
        $mq4_th_ppm = db_get_config("mq4_th_ppm");
        dbg("mq4/mq4_th_ppm=$mq4/$mq4_th_ppm");
        $mq5 = db_get_status("mq5");
        $mq5_th_ppm = db_get_config("mq5_th_ppm");
        dbg("mq5/mq5_th_ppm=$mq5/$mq5_th_ppm");
        $mq7 = db_get_status("mq7");
```
The complete script is stored in the `chapter_01/monitor.php` file in the book's example code repository.

The function starts the main loop where, after getting the sensors' thresholds, it simply gets the last sensor's values and sets up the `alarm` variable accordingly.

Again, we can change the gas concentration thresholds by using the `my_set.sh` command as follows:

```
root@beaglebone:# ./my_set.sh config mq2_th_ppm 5000
```

We can test the script by executing it in the same manner as the previous two, as follows:

```
root@beaglebone:# ./monitor.php -d -f -l -T 5
monitor.php[5819]: signals traps installed
monitor.php[5819]: start main loop (loop_time=5s)
monitor.php[5819]: loop start
monitor.php[5819]: mq2/mq2_th_ppm=84.625/5000
monitor.php[5819]: mq4/mq4_th_ppm=1456/2000
monitor.php[5819]: mq5/mq5_th_ppm=815/2000
monitor.php[5819]: mq7/mq7_th_ppm=569/2000
monitor.php[5819]: alarm=0
monitor.php[5819]: loop end
monitor.php[5819]: loop start
monitor.php[5819]: mq2/mq2_th_ppm=84.625/5000
monitor.php[5819]: mq4/mq4_th_ppm=1456/2000
```
To stop the test, just use the CTRL + C sequence. You should get an output as follows:

```
^Cmonitor.php[5819]: signal trapped!
```

**Final test**

Once everything has been connected and the software is ready, it’s time to do a little test of our new system. The demonstration can be done by using a lighter. In fact, our system is really sensitive to the gas inside the lighter!

First of all, we have to check the system configuration:

```
root@beaglebone:~# ./my_dump.sh config

n   v
mq2_gain   0.125
mq2_off   0
mq2_th_ppm   150
mq4_gain   0.125
mq4_off   0
mq4_th_ppm   150
mq5_gain   0.125
mq5_off   0
mq5_th_ppm   150
mq7_gain   0.125
mq7_off   0
mq7_th_ppm   150
sms_delay_s   300
```

Note that I used a very weak calibration setting; however, these are okay for a demo.
Then, we can take a look at the system's current status:

```
root@beaglebone:~# ./my_dump.sh status
n   v
mq2  73.5
mq4  121.75
mq5  53
mq7  80.5
alarm 0
```

Then, we can do all hardware settings at once by using the `chapter_01/SYSINIT.sh` script in the book's example code repository as follows:

```
root@beaglebone:~# ./SYSINIT.sh
done!
```

Okay, now let's start all the required process daemons:

```
root@beaglebone:~# ./read_sensors.php -d -T 2
root@beaglebone:~# ./write_actuators.php -d -T 2
root@beaglebone:~# ./monitor.php -d -T 2
```

Note that all the daemons are running in background in this way; however, the debugging messages are enabled and they can be viewed into the system log with the following command:

```
# tail -f /var/log/syslog
```

Now, we have to approach the lighter to the sensors and press the button on the lighter in order to allow the sensor to detect the gas. After a while, the alarms should be turned on, and looking at the system status, we should get the following:

```
root@beaglebone:~# ./my_dump.sh status
n   v
mq2  203.875
mq4  166.5
mq5  52.5
mq7  122.625
alarm 1
```
Also, if we have set up a phone number, we should receive an SMS on the phone!

As last step, let's display the data logged by plotting them. We can use the following command to extract the data from the database:

```
root@beaglebone:~# ./my_dump.sh mq2_log | awk '{ print $2 " " $3 }' > mq2.log
```

In the `mq2.log` file, we should find something like the following:

```
root@beaglebone:~# cat mq2.log
15:02:07 75.25
15:02:10 74.25
15:02:12 74.25
15:02:14 74.375
15:02:16 74.25
...
```

Now, using the next command, we're going to create a PNG image holding a plot of our data:

```
$ gnuplot mq2.plot
```

Note that in order to execute this command, you need the `gnuplot` command, which can be installed by using the following command:

```
# aptitude install gnuplot
```

Also, both the `mq2.log` and `mq2.plot` files are need. The former is created by the preceding command line, while the latter can be found in the `chapter_01/mq2.plot` file in the book's example code repository. It holds the `gnuplot` instructions to effectively draw the plot.
The plot of the MQ-2 data of my test is shown in the following screenshot:

As you can see, the sensors are very sensitive to the gas; as soon as I opened my lighter and the gas reached them, the ppm concentration went to high values very quickly.

To stop the test, we can use the following commands:

```
root@beaglebone:~# killall read_sensors.php
root@beaglebone:~# killall write_actuators.php
root@beaglebone:~# killall monitor.php
```

**Summary**

In this chapter, we discovered how to manage the ADCs, the GPIOs lines, a GSM module thought, and a serial port. We also saw how to implement a simple monitoring program that can communicate (through a database server) with a sensors reader task to collect input data, and with an actuators manager to alert the user in case of emergency.

In the next chapter, we’ll see how to manage an **ultrasonic distance** sensor to implement a parking assistant that can communicate to the driver, the distance between the car and the garage’s wall. However, the really interesting part of the next chapter is about how to manage the distance sensor in two different setups: one with all the peripherals near the BeagleBone Black, and another with a remote connection of the sensor through a USB cable.
Where to buy this book

You can buy BeagleBone Home Automation Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.