

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

BeagleBone Home Automation

Live your sophisticated dream with home automation
using BeagleBone

Juha Lumme

[PACKT] open source*
PUBLISHING community experience distilled

BeagleBone Home Automation

Live your sophisticated dream with home automation
using BeagleBone

Juha Lumme



BIRMINGHAM - MUMBAI

BeagleBone Home Automation

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2013

Production Reference: 1181213

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-573-0

www.packtpub.com

Cover Image by Juha Lumme (juha.lumme@gmail.com)

Credits

Author

Juha Lumme

Project Coordinator

Akash Poojary

Reviewers

Raymond Boswel

Dr. Philip Polstra

Proofreader

Faye Coulman

Acquisition Editors

Edward Gordon

Joanne Fitzpatrick

Indexer

Priya Subramani

Commissioning Editor

Sharvari Tawde

Graphics

Ronak Dhruv

Abhinash Sahu

Technical Editors

Ritika Singh

Rohit Kumar Singh

Production Coordinator

Adonia Jones

Cover Work

Adonia Jones

Copy Editors

Janbal Dharmaraj

Gladson Monteiro

Aditya Nair

Deepa Nambiar

Karuna Narayanan

About the Author

Juha Lumme is an engineer with over 10 years' experience in the telecommunications field in various roles. He has been developing platform software for mobile phones and also working on the telecommunication networks side. Embedded systems are his passion, and a hobby he is working on in free time as well.

He is passionate about Linux and open source software in general. The open hardware movement in the recent years is also close to his heart, and he hopes we can all soon hack and build our dreams in a world free of patent abuse.

When not working on his computer, he loves traveling and riding mountain roads on his motorbike around Kanto prefecture in Japan.

First and foremost I would like to thank Rika for her patience and understanding during those nights of writing and coding; this book would not have been possible without her support. I would also like to thank all the people who contributed to this book by reviewing, giving advice, and/or other help. In no particular order, my gratitude goes to Shaon Basu, Raymond Boswel, Dr. Philip Polstra, Akash Poojary, Ritika Singh, Rohit Kumar Singh, Sharvari Tawde, and all the people at Packt Publishing who have contributed to this, some of whom I might not have had direct dealing with.

About the Reviewers

Raymond Boswel hails from sunny South Africa. He has an Engineering degree in Electronics from the University of Pretoria and won the award for best project in the Microelectronics department. Currently, Raymond works for a telecommunications company in the Systems Engineering department, where he is exposed to both frontend and backend server technologies and is fast becoming a jack of all trades.

When Raymond is not slaving away at his day job, he enjoys being active outdoors. His particular interests include soccer, surfing, slacklining, and dancing.

I would like to thank my family and friends; without you, life would be very dull indeed!

Dr. Philip Polstra (known to his friends as Dr. Phil) is an internationally recognized hardware hacker. His work has been presented at numerous conferences around the globe including repeat performances at DEFCON, Black hat, 44CON, Maker Faire, and other top conferences. Dr. Polstra is a well-known expert on USB forensics and has published several articles on this topic.

Recently, Dr. Polstra has developed a penetration testing Linux distribution known as The Deck for the BeagleBone and BeagleBoard family of small computer boards. He has also developed a new way of doing penetration testing with multiple low-power devices including an aerial hacking drone. This work is described in his book *Hacking and Penetration Testing With Low Power Devices*, which is slated for summer 2014 release.

Dr. Polstra has recently developed degree programs in digital forensics and ethical hacking at the university where he serves as a professor and Hacker in Residence. In addition to teaching, he provides training and performs penetration tests on a consulting basis. When not working, he has been known to fly, build aircrafts, and tinker with electronics. His latest happenings can be found on his blog: <http://polstra.org>. You can also follow him at @ppolstra on Twitter.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: The Initial Setup	7
The hardware required	7
The software required	8
Preparing the host machine	9
Windows	10
Mac OS X	10
Linux	10
Starting the target board for the first time	11
Logging in to the system	12
Operating Linux from the console	14
Basic filesystem operations	15
File permissions	16
Running a Hello World program on BeagleBone	18
Summary	20
Chapter 2: Input and Output	21
Hardware interfaces	21
General-Purpose Input/Output pins (GPIOs)	22
On-board LEDs	24
GPIO library for Python	28
Setting the proper time	29
External output	31
External hardware input	34
Pulse width modulator	37
Summary	40

Chapter 3: Creating the Client and Server Applications	41
Sockets	42
An example socket application	43
Echo server	45
Echo client	48
Summary	53
Chapter 4: Extending Server Capabilities	55
Environmental sensors	55
Light sensor	56
Temperature sensor	60
Advanced server	62
Defining our Beagle protocol	62
The new server code	64
The new client code	67
Transistors	71
Summary	75
Chapter 5: Implementing Periodic Tasks	77
Implementing a save/load framework	77
Retrieving and changing permanent settings	81
The client side	81
The server side	83
Periodic tasks on the server	85
Movement-detection alarm system	88
Hardware extensions	93
BeagleBone HD camera cape	94
Changing the boot media	94
Controlling cameras with Python	97
Summary	100
Chapter 6: Creating an Android Client	101
Setting up our Android project	101
Creating an emulator	104
The socket client on Android	107
Defining the UI components	107
The support classes	114
The main UI	116
The network thread	123
The new server features	132
Working from outside your home network	139
Summary	141

Appendix: Security, Debugging, and I2C and SPI	143
Kernel traces and advanced debugging	143
Boot time kernel traces	144
JTAG debugging	145
The I2C and SPI buses	145
The I2C bus	145
Generating the start signal	146
The Slave address transfer	146
Transferring data	147
Generating the STOP signal	147
The SPI bus	147
Considering the security aspects	148
Making the client identify his intentions first	148
Implementing the encrypted password login	149
Version 2.1 modifications to the server code	150
Version 2.2 modifications to Android client code	153
Encrypting all of the communication	158
The GPIO mapping of the P8 and P9 headers	159
Index	161

Preface

This book is written by an embedded systems enthusiast for other like-minded souls. It is meant for makers, inventors, hackers, and generally for people who like to create new things by themselves. Together, we will start a journey into embedded systems, with the aim of setting up a home automation solution, using BeagleBone Black as our platform.

We will start with the very basics that you need to know in order to connect to BeagleBone from your home computer and work on it. After that, we will very quickly start connecting different electronic components and different types of sensors to start providing our BeagleBone platform with capabilities to interact with and sense the world around it.

The approach in this book will be very practical, and the chapters will contain electronic schematics, wiring diagrams, and the necessary operation code written in Python. All the examples will give you a deeper understanding of that specific area, and we will provide some additional pointers and ideas so you can feel comfortable experimenting with it by yourself with a clear sense of direction.

What this book covers

Chapter 1, The Initial Setup, introduces the basics of how to operate a Linux system over a terminal connection and demonstrates a Hello World program executed on the BeagleBone platform.

Chapter 2, Input and Output, introduces LEDs and push buttons to illustrate how general-purpose inputs and outputs work.

Chapter 3, Creating the Client and Server Applications, provides an introduction to Socket programming and creating client and server applications that can talk to each other.

Chapter 4, Extending Server Capabilities, provides an introduction to transistors, light and temperature sensors, and enabling the server to transmit environmental data to the client.

Chapter 5, Implementing Periodic Tasks, introduces the movement sensor. Autonomous operations are implemented and server interfaces are extended for remote reconfigurability.

Chapter 6, Creating an Android Client, sets up the Android development environment, and our client code is rewritten to an Android application that can connect to the server over the Internet.

Appendix, Security, Debugging, and I2C and SPI, includes Linux debugging and talks about the need for advanced security when connecting to the Internet.

What you need for this book

Most of the programming in this book is done using Python, and since the board already has Python installed, and several text-based editors are available, only a proper setup for connecting to the board will be necessary to start developing on the board. However, of course, you might want to use your favorite editor, one that you are comfortable with; this can be easily done with a few exceptional cases that we will talk about later. The Python version used here is the older production version (2.7), and we will use one external general-purpose I/O control library called Adafruit_BBIO.

In *Chapter 6, Creating an Android Client*, we will build an Android application, and for that purpose, we will use Android Development Tools. At the time of writing, it was in Version 22, so Version 22 or newer is recommended. The chosen Android API level is 14, but you can adjust it to suit your device.

Who this book is for

This book is for anyone wishing to learn how to start working with embedded systems, even young programmers without years of professional experience. Readers interested in home automation, or physically small standalone systems, will probably find many interesting aspects in it.

It's a good book for experienced programmers who would also like to wet their toes in the embedded systems field but might have been held back because of all the hardware-specific and cryptic abbreviations or just the general differences of the electronics environment.

Readers would benefit from previous knowledge of electronics and programming in Python and/or Java languages. All the code and topics are explained in detail, so extensive programming experience is not a prerequisite to start learning from this book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Now you should have a new rule file in `/etc/udev/rules.d` called `73-beaglebone.rules`."

A block of code is set as follows:

```
#!/usr/bin/python

import Adafruit_BBIO.GPIO as GPIO
import time


GPIO.setup("P8_8", GPIO.OUT) #Set P8.8 as output pin


while True:
    GPIO.output("P8_8", GPIO.HIGH) #P8.8, aka GPIO 67 aka GPIO2_3
    time.sleep(0.5)
    GPIO.output("P8_8", GPIO.LOW)
    time.sleep(0.5)
```

Any command-line input or output is written as follows:

```
j1umme@simppa:~$ cd Downloads/
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The analog pins are the **VDD_ADC**, **GND_A_ADC**, and **A-INx** pins."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

The Initial Setup

So your new shiny BeagleBone (black?) is out of its packaging, and you have marveled enough at how it is engineered — now what? This is the question we will dive right into in this chapter. We will set up the necessary driver software on your host computer (be it Windows, Linux, or Mac), remotely connect to our board to look around a bit, and go over some basics on how to operate Beagle. The main topics in this chapter are:

- Introducing the BeagleBone hardware
- The operating system controlling the board
- Logging into Linux and performing basic file operations
- Creating a Hello World Python program

The hardware required

As a brain, our board has an ARM Cortex-A8 processor. BeagleBone runs at 720 MHz on the original version (white-colored PCB), and on the newer version, it runs at 1 GHz. You might have heard about this processor before as it is used in some of today's smartphones as well. However, the numbers by themselves might not necessarily tell us much, but it suffices to say that it's powerful enough to run a full-blown operating system.

We have compiled a list of the main hardware differences between the **White** and **Black** models:

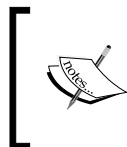
	Black	White
CPU, CortexA8	1 GHz	720 MHz
Memory	512 MB	256 MB
Video output	Micro HDMI + Add-on	Add-on
Audio output	Micro HDMI + Add-on	Add-on
Storage	MicroSD + eMMC	MicroSD

Most importantly, the black version has the HDMI output which means that one could easily connect it to a TV or monitor and send audio through the same cable. Both boards can also get additional audio features via add-on boards, which are called capes in the Beagle world. We will talk more about capes in *Chapter 5, Implementing Periodic Tasks*.

Both boards also contain a plethora of other hardware features. The most interesting of those are 4 **Universal Asynchronous Receivers/Transmitters (UARTs)**, 8 **Pulse Width Modulators (PWMs)**, 65 **General-Purpose Input/Output (GPIO)** pins, 2 SPI buses, 2 I2C buses, 1 **Analog-to-Digital converter (ADC)**, and 4 timers. We will start using GPIO pins and PWMs in the next chapter. The **Serial Peripheral Interface (SPI)** and **Inter-Integrated Circuit (I2C)** buses have been adopted by the semiconductor industry to serve as communication buses when very high transmission speeds are not required. During the course of this book, we will not integrate any kind of external hardware using these buses per se; we will explain their operating principles in *Appendix, Security, Debugging, and I2C and SPI*. Generally, in this book, we will focus on the "Black" BeagleBone version. While the foundation is basically the same on these two boards, there are some significant differences as well. For example, we can see that the extension headers and their GPIO pins are different! You will have to double-check the header pins for the White model and adjust your code accordingly.

The software required

Your new development board comes with a pre-prepared operating system image from the BeagleBoard.org Foundation. The original White model comes with a 4 GB microSD card that contains the bootable OS image, and the Black model has it preprogrammed into the internal eMMC memory.



If you will think about changing the OS yourself, it's good to keep in mind that eMMC is only 2 GB in size, so it might not be enough for all the OS configurations (especially since you should leave some meaningful empty space also for your own files).

The BeagleBoard.org Foundation has selected a Linux distribution called **Angstrom** to be shipped with their BeagleBoard and BeagleBone development boards.

Since the Linux kernel supports ARM natively, you are not limited to a certain distribution or any distribution for that matter. In this book, we will stick to the current default selection as it is likely that the open source community and the BeagleBoard.org Foundation will keep supporting Angstrom as they have for the last 5 years or so. Thus, we can assume that these instructions would stay relevant for the longest time that way.

Don't worry if you are not yet familiar with Linux. You will learn the relevant configuration options and maintenance tasks while going through the chapters in this book; as you begin to work with the system in depth, you will soon become proficient enough to maintain the system yourself. It will also be a nice primer into headless systems and remote terminals that are accessed and operated via terminal connections.

Since we will be using our board over the terminal connection from our host, only the initial driver setup, connection initiation, and a couple of minor things will be different for different host operating systems along the way. We will mention them separately at those occasions.

After that, we will (almost) always work directly on the target to keep the same instructions for any type of host.



With embedded systems, often, the computer you use for development is different from the board you are working on, but both of them still contain similar software, configurations, and so on. In these situations, words such as "target" and "host" can be used to easily identify the subject we are talking about. Saying, for example, "Did you check the network configuration on the target?" is not as ambiguous as "Did you check the network configuration?". We will also stick to this form where relevant.

Preparing the host machine

So first, to set up our host, we need to install the drivers for our target.

First, with an Internet browser of your choice, navigate to <http://beagleboard.org/static/beaglebone/latest/Drivers/> and depending on your host machine type, go into the subdirectory to find the necessary drivers.

Windows

We will first have to find out if our system is 32 bit or 64 bit. Microsoft has prepared a helpful page at <http://support.microsoft.com/kb/827218> where you can find instructions on how to find out your system type on any of the recent Windows platforms. If you are running a 64-bit system, download the `BONE_D64.exe` file or the `BONE_DRV.exe` file and install the driver. The BeagleBoard.org Foundation seems to have decided not to go through the process of digitally signing the driver, so Windows will probably ask/complain about this several times; proceed with the installation anyway as it won't cause any harm to your system.

Mac OS X

We need to install `HoRNDIS-rel4.pkg` for networking and `FTDI_Ser.dmg` for serial connections. Go ahead and download both of them, then double-click on each one to install them.

Linux

No particular driver is needed for Beagle in Linux, but there are some udev rules that are helpful for accessing the serial console. Download the script file in the web page and open a **terminal emulator** (from the application menu). Then, add executable permissions to the downloaded script and execute it as a super user:

```
jlumme@simppa:~$ cd Downloads/
jlumme@simppa:~/Downloads$ chmod u+x mkudevrule.sh
jlumme@simppa:~/Downloads$ sudo ./mkudevrule.sh
[sudo] password for jlumme:
jlumme@simppa:~/Downloads$
```

In the preceding console command, you can see that we have used a prefix `sudo` in our call to `./mkudevrule.sh`. This is actually a special command that tells the system that the `mkudevrule.sh` script needs to be executed with root privileges. Root privileges mean administrative rights. These are required as the script is placing the files within a folder that needs those privileges.

Now you should have a new rule file in `/etc/udev/rules.d` called `73-beaglebone.rules`.

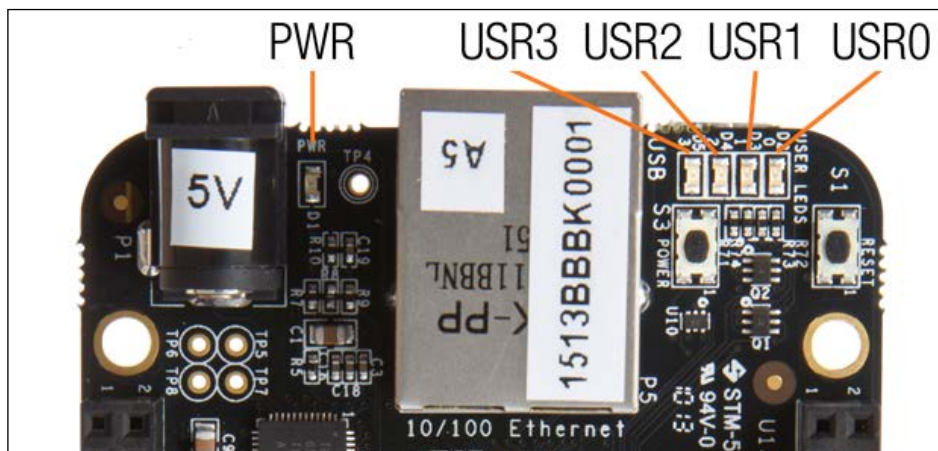
Starting the target board for the first time

Now that the drivers are set up, you can use the microUSB cable that came with the board to plug in the board to the USB port on your host machine, and the board will start booting up.



Keep in mind that the USB 2.0 specification specifies a maximum current of 500 mA, which can be provided by a single port. This might not be enough if you connect many peripherals to the USB port using a USB hub. In those cases, you will need to use a 5-volt DC power supply to power the board.

On the board, there are five LEDs in total; they are all located on the side of the power and LAN connectors, as shown in this picture (many thanks to the BeagleBoard Foundation for allowing us to use this picture):



The default configuration for the user LEDs is:

- **USR0:** This LED is a heartbeat LED; by default, it will keep flashing while the system is powered on
- **USR1:** This LED blinks when the microSD card is being accessed
- **USR2:** This LED will blink during the CPU activity
- **USR3:** This LED will blink when eMMC memory is being accessed

When the system is starting up, it will not be accessible via USB before the USB drivers are initialized. To debug boot time issues, it is possible to see startup debug traces via a serial console connection on Header J1. This is an advanced topic that is described in *Appendix, Security, Debugging, and I2C and SPI*.

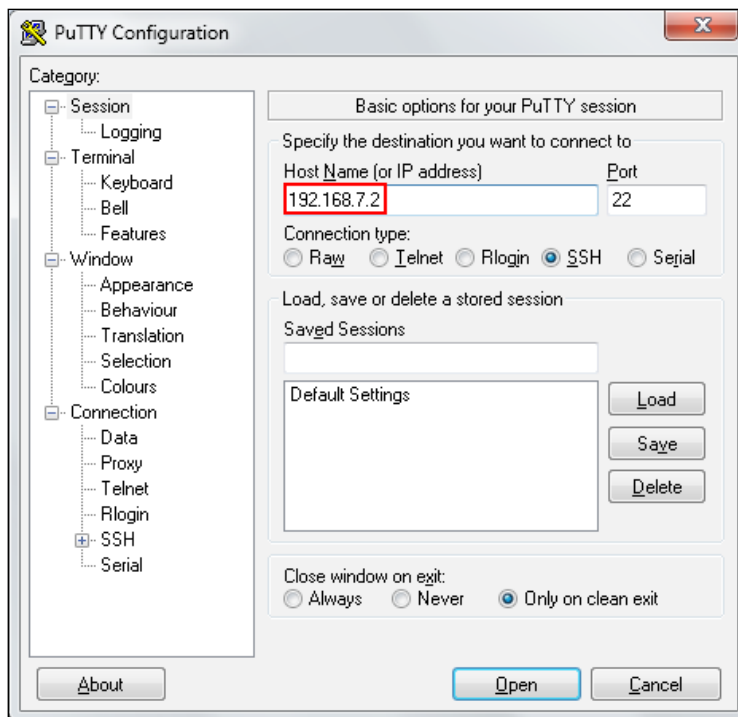
The boot process usually takes around 10 seconds, after which you will be able to connect to the board. There are several ways to connect to the target, and we will talk about two of those. If you are familiar with Linux, you probably already know how console logins work, but if you are not – no worries as we will cover some basics.

Logging in to the system

The login procedure is slightly different in Windows than it is in Mac OS X and Linux, but there's not a great difference between the two.

On Windows, you will need to install a program to create SSH connections. We recommend PuTTY. PuTTY is a free and open source terminal emulator/serial console that we can use to connect to our console login session on the board.

From the **PuTTY Download Page** at <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>, you can download the `putty.exe` file for Windows on Intel x86. It should be the first download link on the page. It's just an executable; you don't need to install anything. Once the download is complete, launch PuTTY, and in the **Host Name (or IP address)** field, enter `192.168.7.2`:



When the system asks for the login name, enter `root` and just press the *Enter* key for the password. You should now be connected to the board. Accept the RSA fingerprint query that BeagleBone uses to identify itself.



The RSA fingerprint that is received from BeagleBone is a public key that can be used to identify whether the remote host is actually the host that you imagine it to be. This topic is out of scope for this book, but it's good to know that in this way, one can identify a trusted host over the network. These SSH connections are also automatically encrypted.

In Linux and Mac OS X, open the console or terminal (you can find it in the application menu); you can just use regular SSH applications to connect to the remote host.

```
jlumme@simppa:~$ ssh root@192.168.7.2
The authenticity of host '192.168.7.2 (192.168.7.2)' can't be
established.
RSA key fingerprint is al:cb:9f:8b:f4:6f:79:52:47:87:83:db:d6:9a:70:6e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.7.2' (RSA) to the list of known hosts.
root@192.168.7.2's password: [Just press enter here]
root@beaglebone:~#
```

Notice how we supplied the username `root` to the SSH application before the IP address. If you don't supply it, the system will try to login with your default username which is not available. Now we're inside our target, and we're ready to start giving commands to it.

Another way to log in to our system is to use a web browser. This is not as convenient, but it can be useful if you can't, for some reason, install the SSH software on the computer. Open the browser window and navigate to `http://192.168.7.2`. The web server on the target should show you a welcome page. Scroll down until you find the **Gate ONE SSH Client** section; this text is a link which starts an SSH session over the browser. It will be slower to use, but you can still use it to operate on the board in situations where you might not be able to install additional software, for example, at a friend's house.



The address (192.168.7.2) that we used to log in to our board is actually the default address for the board when it is connected to the host via a USB connection. If you are facing difficulties in connecting to the IP address, you might want to try "ejecting the USB drive" from your development PC. Older versions of the BeagleBone OS software required this, but the newer software versions do not.

If you want to connect to the board via the router and Ethernet connection, the address will be different. To check the Ethernet interface IP address, you can use the `ifconfig eth0` (when first connected to the board via USB).

Operating Linux from the console

Let's take one step back now, and talk about terminals and consoles for a minute.

If you are not familiar with Linux, maybe you have never used the computer from the command line, but instead have always used the provided **Graphical User Interface (GUI)**. GUIs are great and provide beautiful graphics, animations, and a superior usability experience overall. However, they also demand a lot of resources. And we're not only talking about memory or CPU performance, but peripherals such as keyboards, monitors, and mice. As you noticed, our board has none of these. Of course, you can connect those via the available USB and HDMI ports, but since our target is to build a standalone system, in this book, we will focus on the remote and console operations of our target.

Now, while performing the steps in the previous section, when we logged into the target, we actually started a terminal session. Here, we can control the board with the command line. Using the command line will take some time to get used to; but once you are comfortable with it, it's very fast and efficient to use. Commands can also be chained to be executed sequentially; this is called **scripting**. You will find a lot of scripts on the Internet that perform a certain task in a specified order. Did you notice that in the *Preparing the host machine* section for Linux, we set udev permissions with the `mkudevrule.sh` file? This was also a command-line script. While it's not necessary to become a command-line guru to write and execute programs on our board, we need to go over a few basic things about console operation.



Linux, unlike Windows, does not have a central registry that keeps the information of the system. Everything in Linux is built on top of files, even the hardware under the `/dev/` folder. Or more accurately, almost everything is accessible in Linux through the filesystem. Some of the files might not be the actual files that are permanently saved onto a disk, but you can still access (read) them in pretty much the same way. It's important to become at least somewhat familiar with operating on files in the console.

Basic filesystem operations

When you log into the system, you are placed in the **home folder** of the user you used to log in as, in our case, **root**.



In Linux, **root** is a user that corresponds to an administrator. He can perform all the tasks on the system and has the rights to add other users, perhaps with fewer privileges. For now, we will keep using root even though it is not a good practice in general. You might want to read up on "Linux and user rights" from the Internet.

You can always see the current folder with the command `pwd` (think of it as an acronym for the present working directory):

```
root@beaglebone:~# pwd
/home/root
root@beaglebone:~#
```

You can move between folders with the command `cd` (change directory). The `cd` command takes one input parameter that specifies the directory you want to change to. For example, to go one folder up in the hierarchy, you type `cd ..`. You might have noticed that the current folder structure starts with `/`. This is the root point of the whole filesystem on our Linux OS. You can go and take a look at it:

```
root@beaglebone:~# cd /
root@beaglebone:/# ls
bin dev home lost+found mnt run sys usr
boot etc lib media proc sbin tmp var
```

To list the contents of the current folder, we use the command `ls`. Here, you can see that there are quite a few folders in the filesystem. For example, `/home/folder` contains the home folders of all the users of this system. Take another example where, in the folders `/bin` and `/usr/bin`, you can find programs and commands that can be executed on this system (programs like `ls` and `pwd` are also in the `/bin` folder). Remember how we said that everything is a file? Well, so are the basic commands!



Did you notice that when we navigated away from the `root` user's home folder to the current folder, the last character changed to show the current folder? This is because `~` is a shorthand to indicate the user's home folder. You can always go back to the user's home folder by just typing `cd` and pressing `Enter`.

File permissions

Let's go back to the home folder and look at what we can find there:

```
root@beaglebone:/# cd
root@beaglebone:~# ls -l
total 4
drwxr-xr-x 2 root root 4096 Jan  1 00:01 Desktop
root@beaglebone:~#
```

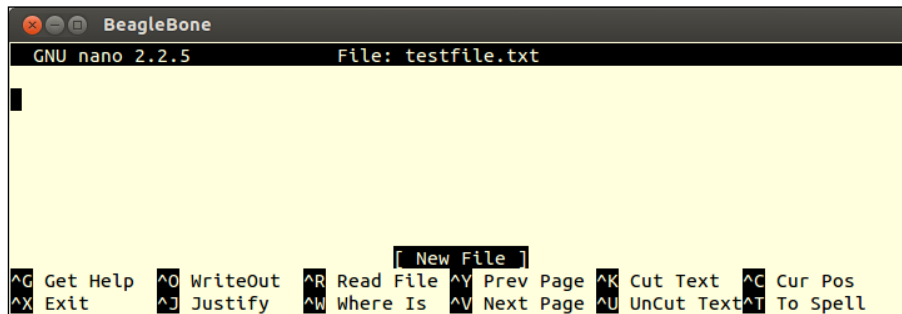
Notice how we first called the change directory command without any parameter? This means that the user wants to change to his or her home folder. In this case, it is `/home/root/`; other users might have different home folders. This time we also added a parameter `-l` to the command. A parameter gives additional instructions to the `ls` command, in this case, to use a long listing format. Now, in the long listing format, there is much more information visible about the files in this directory. The first column is especially important; it defines the access rights to this particular file. There are four basic file-access right properties:

- `d`: This property stands for **directory**; it means that the file is a directory
- `r`: This property stands for **reading rights**
- `w`: This property stands for **writing rights**
- `x`: This property stands for **execution rights**

Why are the properties listed several times then? This is because of the nature of how file-access restrictions work in Linux. File restrictions can be specified separately to three levels: **user**, **group**, and **everyone**. They are listed in that order.

For someone who's new to Linux, this might sound a bit confusing; maybe a practical example can help you understand this. Let's create a new file. We can do this, for example, by launching a text editor that comes with the default installation. There are several editors available; for now, we will use an editor called **nano**. It is easy to understand and will probably feel familiar even to users who have not used command-line editors before.

So, to create a file in the current folder, just type `nano testfile.txt` and an editor should pop up on the screen:



Write any sentence in the window and press `Ctrl + X` to exit. The program will ask you if you wish to save the file; in that case, press `Y` and then `Enter` to confirm the filename, `testfile.txt`. Now, let's list the directory again:

```
root@beaglebone:~# ls -l
total 8
drwxr-xr-x 2 root root 4096 Jan  1 00:01 Desktop
-rw-r--r-- 1 root root   20 Jan  1 01:24 testfile.txt
```

We notice that the freshly created `testfile.txt` file now has the default file access rights set. You can see that it was created by user `root` and belongs to the group `root` (the `root root` part). Only the owner of this file can write to it (the first part `rw-`). Let's try modifying the file permissions and give everyone the rights to write to this file. For this, we can use a command called `chmod` (derived from "change mode"):

```
root@beaglebone:~# chmod a+w testfile.txt
root@beaglebone:~# ls -l
total 8
drwxr-xr-x 2 root root 4096 Jan  1 00:01 Desktop
-rw-rw-rw- 1 root root   4 Jan  1 02:01 testfile.txt
```

Now you can see that we gave everyone the (a) rights to write (+w) to that file, and the file permissions string (-rw-rw-rw-) has changed accordingly.



Logically, if you want to remove a permission, you have to use the minus sign (-) instead of the plus sign (+).

Why it's important to understand the file permissions is because, unlike some systems where only certain file types can be executed, in Linux, any file with an executable bit set is executable. Thus, our Python programs will also require an executable bit to be set. To give executable permissions to a file, we can use the +x flag.



Usually, you don't necessarily want to give everyone execution rights; for example, you can call `chmod u+x`, which means that you only grant the current user the execution rights. As you probably already guessed, for groups you can use `chmod g+x`.

You must be anxious to try and test some code; let's take a look at a basic Hello World example.

Running a Hello World program on BeagleBone

Angstrom Linux already has Python preinstalled, so we can start right away. Let's open our text editor (nano) again, and create a file called `hello.py`. In the editor, just type the following lines:

```
#!/usr/bin/python

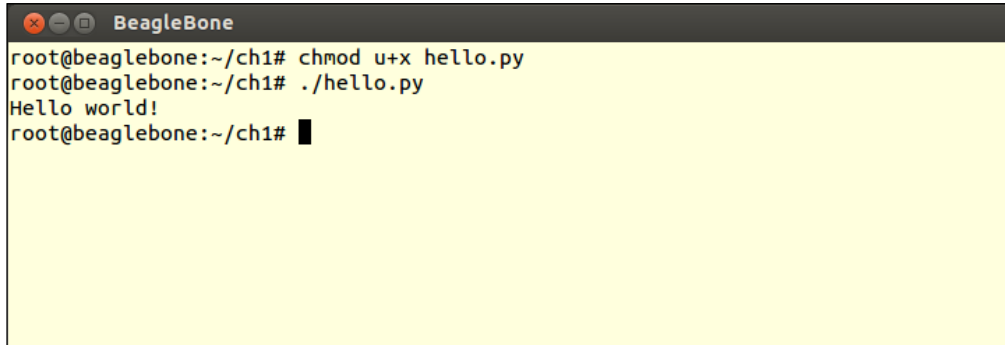
print "Hello world!"
```



Downloading the example code

You can download the example code files for all Packt books you have from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Now, save the file, give it execution permissions, and try running it:

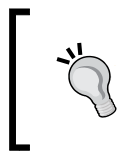
A terminal window titled "BeagleBone" with a yellow background. It shows the following commands and output:

```
root@beaglebone:~/ch1# chmod u+x hello.py
root@beaglebone:~/ch1# ./hello.py
Hello world!
root@beaglebone:~/ch1#
```

Congratulations, you have just created and executed your first program on the board!

Now, you might have noticed a couple of peculiar things. First, what is that `#!/usr/bin/python` line present in our code? This is actually a Unix-specific information known as **shebang**. Shebang is meant for the program loader; it informs the system where to find a suitable interpreter for this file. As mentioned before, any file with execution permission set can be executed. But if the file is not written in native code (for example, compiled C code), the system depends on shebang information to find a suitable command interpreter.

Another thing we did was prepend `./` to the executable file. This is because by default, Linux searches for executables only in the folders that it is told to look from. By default, the root user's home folder is not one of those folders. The dot(`.`) is a shorthand to indicate the current folder. This shorthand is basically the same as if you had typed `/home/root/hello.py`.



The directories that are searched for executables are contained in an environment variable called `PATH`. You can list environment variables with the command `printenv`. Here, you will also see the `PATH` variable.



Summary

This chapter gave us a very short introduction to Linux and our board. We talked a bit about the hardware, how to connect it to your host machine, how to login to the Linux system via SSH, and some very basic commands to perform operations on a Linux system.

You saw how files and folders are represented in the filesystem and how you could modify these permissions. We created our first program and successfully executed it in our target environment. From here on, we will start working on more complicated programs.

In the next chapter, we will connect an LED to our board and control it. Subsequently, chapter by chapter, we will start working through the different areas of our hardware, building up knowledge on how to control different peripheral devices and learning how to communicate with our Beagle remotely over a network.

2

Input and Output

In the previous chapter we executed our very first Hello World program on our target, and in this chapter we will do the embedded equivalent of that. We'll start connecting additional hardware to our target, and see how we can interface it with our software.

We won't spend too much time on principles of electronics here, but we will go over the basics that you have to understand, so you can avoid damaging your precious Beagle. We will talk about the following topics:

- GPIO pins, and how they are used
- LEDs, and how to drive them
- Buttons, and how we can accept input from them
- Pulse width modulation, and how we can use it to trick our eyes to see an LED changing its brightness

Same as with the first chapter, let's take a moment to talk about some features of our Beagle that you need to be familiar with when working on examples in this chapter.

Hardware interfaces

With any kind of electronic components/designs, the manufacturer always provides some kind of specification documentation. Same goes with our Beagle.

BB foundation provides **System Reference Manual (SRM)** (from now on), and you can find it at <http://beagleboard.org/hardware/design>. Just check your board revision (it should be printed on your box), and download the correct one. All instructions in this book are based on the first production release for **BeagleBone Black** called **Rev A5A**. At the time of writing this book, the latest revision stands at **A6**.

You should keep in mind that BeagleBone Black and the original white model don't have the same **General-Purpose Input/Output (GPIO)** pin layout. In this book all the examples are presented on the black model, so please take care if you are using the white model, and follow the guidelines given in this book!

So back to the SRM; this documentation is very useful while planning and designing your own systems, and usually you end up looking through it often, so save it somewhere, since it will become very handy.

We have also listed the full GPIO multiplexing chart in the section *The GPIO mapping of the P8 and P9 headers*, in *Appendix, Security, Debugging, and I2C and SPI*, at the end of this book.

Multiplexing means that a single pin on the processor can serve multiple purposes depending on the current configuration. This is a common way to squeeze more peripherals to a development board, and end-user can then select the peripherals he wants to use. If you look at the GPIO multiplexing chart, you will notice that there are seven modes that a pin can be configured in (though not all the pins have seven different states).

For example, look at the chart for header P8 and pin 13, named **EHRPWM2B**. If the pin is configured to MODE0, it will be used for address line 9 for the **General Purpose Memory Controller (GPMC)** bus and if it is configured in MODE1, it will be used for data line 22 for LCD display, and so forth. The important part here is to realize that using certain pins in certain configurations will block other hardware, so care should be taken when connecting peripherals to the processor.

In the chart we provide in *Appendix, Security, Debugging, and I2C and SPI*, we have some of the pins marked in red for you. By default, these pins are reserved for some purpose already, so you won't be able to use them without reconfiguring some part of the hardware.

General-Purpose Input/Output pins (GPIOs)

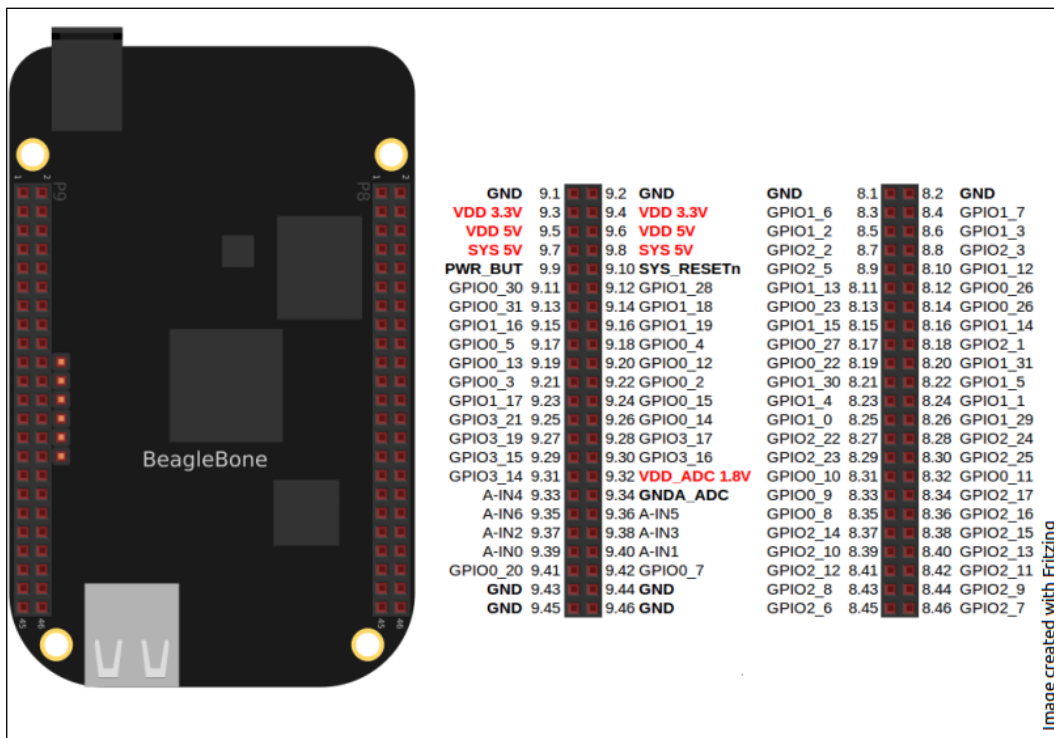
GPIO pins, as the name suggests, are general-purpose pins that are connected to the processor and can be configured to our liking. Thinking broadly, it means that the pin is dynamically configurable to be either an output or an input pin. On top of that, it can serve different functionalities, depending on the configuration.

In the output mode, we (or a peripheral) can control whether voltage is on or off. Respectively, these are called high (also source) or low (sink) states. This is called driving a pin.

In the input mode, the current state of the pin can be read. The read state will be either 1 or 0, depending on the voltage level on that pin. Certain pins also have capability to read input voltage, not just an on or off state.

It is very important to know that GPIO pins in our Beagle are driven with 3.3 volts (with few exceptions, they are marked in the following figure), and in the input mode you cannot apply more voltage to them without risking permanent damage to your board.

Have a look at your board, and turn it so that the **BeagleBone** text is properly aligned. You can see two long 46-pin headers on both sides of the board, and in the following figure, you can see their names on the right:



Throughout this book, we will be showing the schematics of the hardware connected to our target in a very similar fashion.



When you look at the previous figure, you might wonder what the numbers are after the GPIO text. GPIO pins are actually behind separate control chips, and each of these control chips controls 32 pins. Sometimes (for example, the Linux kernel) the GPIO pins are only referred to as a number, for example, 88. You can get the actual chip and pin number by the formula $[\text{bank} * 32 + \text{pin}]$. In this case it is $2 * 32 + 24$ (so it would be P8.28 or GPIO2_24).

We already mentioned this once, but we repeat it once again here, always remember to re-check the voltage level when you hook up live wires to your Beagle. GPIO pins can only take 3.3 volts and analog pins even less; analog pins operate in 0-1.8 volt range. Analog pins are the **VDD_ADC**, **GNDA_ADC**, and **A-INx** pins.

It is a nasty surprise when you start smelling something burning, and a puff of smoke comes out from some components on your board. With the board full of tiny SMD components, it's usually time to order another board, as finding out exactly how many components have been damaged takes a lot of time and experience.

On-board LEDs

We have four users LEDs on our board, and we can control them if we wish.

Remember how in the first chapter we talked about everything being represented with files in the Linux kernel? Well, so is the external hardware as well!

Log in to your target board, and navigate to the following folders:

```
root@beaglebone:~# cd /sys/class/leds/
root@beaglebone:/sys/class/leds# ls
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
```

The `/sys` folder is a special folder that holds quite a bit of interesting information, and you might find it interesting to explore that further with the help of Google. But for now, let's concentrate on the `/sys/class/leds` folder. In this folder, the kernel has exposed to us all the four LEDs that are on the board. Let's go into the `usr0` (heartbeat LED) folder and look at its contents:

```
root@beaglebone:/sys/class/leds/beaglebone:green:usr0# ls -la
total 0
drwxr-xr-x 3 root root    0 Jan  1 00:00 .
drwxr-xr-x 6 root root    0 Jan  1 00:00 ..
```

```

-rw-r--r-- 1 root root 4096 Jan  1 06:28 brightness
lrwxrwxrwx 1 root root    0 Jan  1 06:32 device -> ../../../../gpio-
    leds.7
-r--r--r-- 1 root root 4096 Jan  1 06:32 max_brightness
drwxr-xr-x 2 root root    0 Jan  1 06:32 power
lrwxrwxrwx 1 root root    0 Jan  1 06:32 subsystem ->
    ../../../../class/leds
-rw-r--r-- 1 root root 4096 Jan  1 06:32 trigger
-rw-r--r-- 1 root root 4096 Jan  1 00:00 uevent

```

Here we see several interesting files. The file called `brightness` controls the brightness of the LED. It takes values between 0-255. However, actually in this case a value 0 means that the LED is off, and anything else means that the LED is on.

The `trigger` file contains some predefined triggers that can cause the LED to blink. You can list the content of the file to the console with the command `cat`; this takes the target file as a parameter and will list the contents of that file.

```

root@beaglebone:/sys/class/leds/beaglebone:green:usr0# cat trigger
none nand-disk mmc0 mmc1 timer oneshot [heartbeat] backlight gpio
cpu0 default-on transient

```

So currently the trigger has been set to `heartbeat`. You can stop this by writing `none` to the file. Since these are special files, it's easier to push data to these files, for example, with a command called `echo`. So, let's try stopping the LED heartbeat:

```

root@beaglebone:/sys/class/leds/beaglebone:green:usr0# echo none >
    trigger
root@beaglebone:/sys/class/leds/beaglebone:green:usr0# cat trigger
[none] nand-disk mmc0 mmc1 timer oneshot heartbeat backlight gpio
cpu0 default-on transient

```

We can now observe that our LED's heartbeat has stopped. Now you can control the LED as you wish by writing either 0 or (for example) 1 in the `brightness` folder. Go ahead, try it.



The contents of this special folder are actually defined by the trigger. For example, if you defined a timed trigger (`echo timer > trigger`), you will notice that the contents of this folder will change to include the files called `delay_on` and `delay_off` with what you can define some delays—how long your LED will be on or off.

You can restore the heartbeat operation by just echoing the text `heartbeat` to the file `trigger`.

You must already be thinking how we can start working with these files through programming. Let's do just that!

Go to your home folder, and let's create a new program called `led_fun.py`. We can control the LEDs very much the same way as we did through command line, via files.

First, we need to open file handlers to the brightness and trigger files of each LED:

```
led0 = file("/sys/class/leds/beaglebone:green:usr0/trigger", "w")
led0_ctrl =
    file("/sys/class/leds/beaglebone:green:usr0/brightness", "w")
```

We do that for each LED we wish to control, and then write the keyword `none` to the trigger:

```
led0.write(str("none\n"))
```

After this line, the trigger for each LED is stopped, and we can control our LEDs freely by ourselves.

To write a new value to the control file (brightness) in a similar way, we write:

```
led0_ctrl.write("0")
led0_ctrl.flush()
```

Because file access is buffered in Python, we have to always force a buffer flush after we have written to the file.



If you use an **IDE (Integrated Development Environment)** or some text editor on your host system for writing the code, you will need a mechanism to transfer the file to the board. In Linux and Mac OS X, you can use `scp` (secure copy); the command is very similar to the `ssh` command we used to log in to the system:

```
jlumme@simppa:~/ide_folder$
scp led_fun.py root@192.168.7.2:/home/root/
root@192.168.7.2's password: [press enter here]
led_fun.py 100% 325B 202.3KB/s 00:01
jlumme@simppa:~/ide_folder$
```

On Windows you will need to use some program such as WinSCP (<http://winscp.net>) or other file-copying program that copies over SSH.

These are the basic functions you need to achieve the same functionality in Python that we did via the command line. Let's add a couple of `wait` functions and `for` loops to have some flashing going on.

Full listing of `led_fun.py` is shown in the following code:

```
#!/usr/bin/python

import time #For our time.wait() calls

'''
Simple function that takes an LED number, and writes
the desired string to the control file of that LED.
'''
def write_to_led(led, string):
    if led == 0:
        led0_ctrl.seek(0) #Go to beginning of the file
        led0_ctrl.write(string)
        led0_ctrl.flush() # Force buffer flush

    elif led == 1:
        led1_ctrl.seek(0)
        led1_ctrl.write(string)
        led1_ctrl.flush() # For the write

# File handles to the kernel data structures
led0 = file("/sys/class/leds/beaglebone:green:usr0/trigger", "w")
led0_ctrl =
    file("/sys/class/leds/beaglebone:green:usr0/brightness", "w")

led1 = file("/sys/class/leds/beaglebone:green:usr1/trigger", "w")
led1_ctrl =
    file("/sys/class/leds/beaglebone:green:usr1/brightness", "w")

# Let's turn off the default operations on the LEDs
led0.write(str("none\n"))
led1.write(str("none\n"))

led0.flush() # To force emptying of the file buffer
led1.flush()

while True:
    for i in range(0,2):
```

```
write_to_led(i, "1")
time.sleep(0.05)
for u in range(0, 2):
    write_to_led(u, "0")
    time.sleep(0.05)
```

Executing the code will make the LEDs `usr0` and `usr1` flash in order with some sleep time. You can stop the execution at any time by pressing the keys *Ctrl* + *C* (this is an escape sequence in Linux, any program should exit when you press this combination).

The program is very basic and also quite crude. There is no input verification and it doesn't restore the system state after you exit. But it gives you an idea about the files that actually represent the hardware in the Linux kernel. We're sure you already know how to restore the LED states if you want to, so we can move ahead with more hardware examples.



If you ever forget what was originally in those control files, you can just reboot your board since these are not permanent files but data structures that are created by the kernel during boot time.

GPIO library for Python

From now on, we will start working with GPIOs directly from the code (meaning, without mapping to the files the kernel provides via the `/sys/class/gpio/` folder), and for that purpose, we can use a very handy library that has been developed by Adafruit, called **Adafruit-BBIO**.



Usually when you control any type of low-level hardware, you need to access it via memory-mapped hardware registers. This control is strictly specified in the specifications from the manufacturer and it usually takes quite a bit of studying and understanding how each peripheral works. The Adafruit-BBIO is a very helpful library that hides all this complexity from the end user, and all the functionality is accessible via simple function calls.

To download the library, you need Internet connectivity; so go ahead and connect a network cable to your board, and wait a moment for it to receive an IP address.



To check the IP configuration, type `ipconfig` in the console.

Now we can use the Angstrom's package manager to download the necessary packages. First we need to update the time on our board, otherwise we will have SSL certificate problems.

Setting the proper time

To update the time, type the following code:

```
root@beaglebone:~# ntpdate -b -s -u pool.ntp.org
root@beaglebone:~# date
Wed Nov 13 19:39:47 JST 2013
root@beaglebone:~#
```

However it's better to make time autoupdate itself, otherwise you will lose the time during a reboot. To do this, you need to first install **Network Time Protocol (NTP)** service:

```
root@beaglebone:~# opkg update
root@beaglebone:~# opkg install ntp
```

After this, you should check the address of the server physically close to your region. Go to <http://www.pool.ntp.org/> and from the right, choose the appropriate zone, and then a country you reside in. For example, for Japan, we get:

```
server 0.jp.pool.ntp.org
server 1.jp.pool.ntp.org
server 2.jp.pool.ntp.org
server 3.jp.pool.ntp.org
```

Insert this information to your NTP config file at `/etc/ntp.conf`. You should also comment out any other "server" or "fudge" lines currently existing in the configuration file. Next we will have to set up the correct time zone. In the folder `/usr/share/zoneinfo/`, you can find all the regions and countries listed as a file; you should locate the file that is appropriate for your location. In the `/etc` folder, there is a file called `localtime`. This is a **symbolic link** pointing to the correct time zone:

```
root@beaglebone:~# ls -la /etc/localtime
lrwxrwxrwx 1 root root 30 Nov 24 2013 localtime ->
/usr/share/zoneinfo/Asia/Tokyo
```



Symbolic links, or symlinks as they are often called, are links to a file. These links can be operated in the same way as files, and operations performed at them will be performed against the file they point to (with some exceptions, such as deleting a symlink).

If the `localtime` file is not pointing to the correct file, delete the current link, and create a new one pointing to the appropriate location:

```
root@beaglebone:/etc# rm localtime
root@beaglebone:/etc# ln -s /usr/share/zoneinfo/Asia/Tokyo /etc/
localtime
root@beaglebone:/etc# ls -la localtime
lrwxrwxrwx 1 root root 30 Nov 24 2013 localtime ->
/usr/share/zoneinfo/Asia/Tokyo
```

Now that we have configured our `ntp`, start the services:

```
root@beaglebone:/etc# systemctl enable ntpdate.service
root@beaglebone:/etc# systemctl enable ntpd.service
```

And then you have to make two more changes in the `ntpdate` service:

```
root@beaglebone:~# nano /lib/systemd/system/ntpdate.service
```

Modify the `ExecStart` variable to look as following:

```
[Service]
Type=oneshot
ExecStart=/usr/bin/ntpd -q -g -x
ExecStart=/sbin/hwclock --systohc
RemainAfterExit=yes
```

After this, the time should be kept even after a reboot. Try it, you can reset the board by entering the command `reboot` in console.

After this, we should be able to install the required packages. First install is `pip`, which is a handy Python package manager:

```
root@beaglebone:~# opkg update && opkg install python-pip python-
setuptools python-smbus
```

This will take a minute or so, and once it finishes, you can install the Adafruit library:

```
root@beaglebone:~# pip install Adafruit_BBIO
```

Now we have the necessary library installed, and we can access it from the Python code.



If you are not familiar with package management in Linux, you could think of it as a kind of app store for your Linux distribution. It is a repository that holds precompiled programs that are compatible with the system libraries your Linux is running on. Instead of downloading programs from all over Internet (which you still can do!), you ask the package manager to install them for you. The great benefit of this is that your package manager will always update to the newest version of the software automatically when it becomes available. Your smartphone does exactly the same thing.

External output

Before we start attaching our own LEDs to the GPIO pins and driving them, we should mention a few things about current and voltages.


LEDs are light emitting diodes, meaning that they pass current one way, and when they do, they light up. They also have a limit of current they can pass through before they break (they will light up very brightly for a moment and burn out). You can usually assume that a medium-size LED can withstand 20 mA of current before burning out, but if you have a specification at hand, you can easily verify this for your LED.

An LED has a forward voltage. This is the level after which the LED will start conducting and lights up. There will also be a voltage drop across the LED when it's conducting. Usually we can assume this to be around 1.7V, the forward voltage varies in LEDs and certain parameters such as the color of light it emits affect that.

Armed with these two bits of information, we can think about our circuit. According to specification, our board can withstand 4, 6, or 8 mA of current in the GPIO pins, depending on the pin. So, to be on the safe side, let's use 4 mA as our current target and calculate the suitable resistor value from the *Ohms law* $R = V/I$:

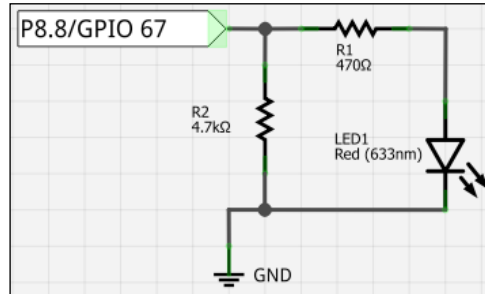
$$R = (3.3 - 1.7) / 0.004$$
$$R = 400$$

We get a value of 400, so we can pick a resistor of 470 Ohm. You can always choose a bigger resistor; this will just mean that the LED will shine with less brightness, and if it's too dim, start decreasing the resistance step-by-step.

 There are several online resistor value calculators on the Internet, so you can always verify your calculations. For example: <http://www.ohmslawcalculator.com> (easy to remember)

Now, let's design our circuit. In this example, as our source pin, we'll use **GPIO 67**, which is located in header P8 pin8.

Our schematic will be like the following figure:

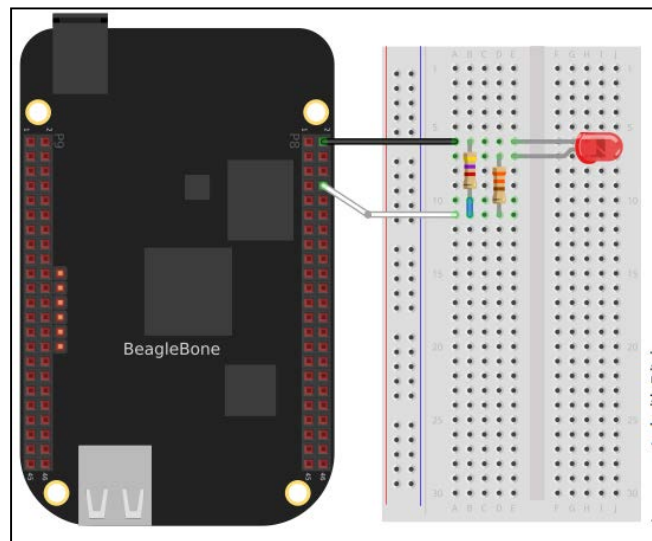


Connect a wire from that pin to the resistor and from the resistor connect to our diode's anode pin (the longer one), and then connect the cathode to the ground.



Besides having a different length of legs, round LEDs also tend not to be perfectly circular when looking at them from top or below. The cathode side of the LED will have a flat, as if shaven off, side.

Our actual wiring should look as shown in the following figure:



After this, we should write a program to control our GPIO. Open up your editor, and let's start writing GPIO control code. We will call this program `ext_led_fun.py`.

First, we need to import our newly installed GPIO library; we do that with the following code:

```
import Adafruit_BBIO.GPIO as GPIO
```

Then, we need to set up our GPIO as output. With the help of this library, it's extremely easy:

```
GPIO.setup("P8_8", GPIO.OUT)
```

To trigger our GPIO, you use:

```
GPIO.output("P8_8", GPIO.HIGH) #P8.8, aka GPIO 67 aka GPIO2_3
```

This sets the GPIO 67 pin to the HIGH state and lights up our LED. Now we can add a while loop again and a couple of calls to sleep to get a nice flashing red LED.

Full listing of `ext_led_fun.py` is as follows:

```
#!/usr/bin/python

import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P8_8", GPIO.OUT) #Set P8.8 as output pin

while True:
    GPIO.output("P8_8", GPIO.HIGH) #P8.8, aka GPIO 67 aka GPIO2_3
    time.sleep(0.5)
    GPIO.output("P8_8", GPIO.LOW)
    time.sleep(0.5)
```

Now we have a nice blinking external LED, and on the basis of this, we could add several different color LEDs, for example, to indicate the current status of our program.



In our `ext_led_fun` schematic, can you guess why we have the resistor R2 in place? Go ahead and remove it. Can you notice anything different? This resistor is what is called a **pull down resistor**. This resistor is needed when we are not driving the circuit (LED). When nobody drives a GPIO pin, it will **float** (its voltage value will be somewhere between low and high). Now depending on the LED characteristics, it might actually be dimly lit, if there is no resistor **pulling the circuit to ground**. When we apply current to the circuit (drive GPIO high), the combined R1 and LED resistance is less than R2, and thus the current will flow through the LED as we expect it to.

External hardware input

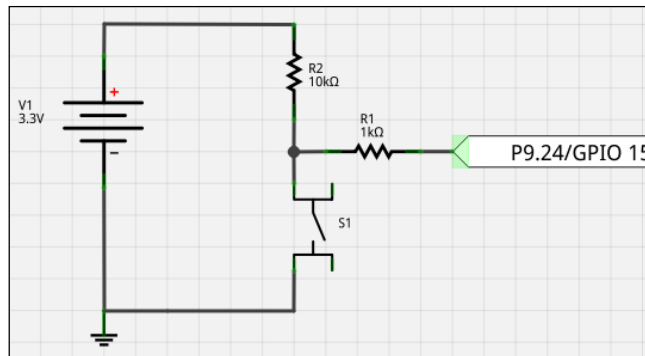
So now that we have an idea on how to operate GPIO in a way that we control its output, let's get going with input.

In general, input for a standalone system usually means some kind of trigger to perform some actions. In all its simplicity, an input trigger is just a voltage level that can go from low to high or from high to low. In this example, we will add an external button to our system, and it will perform duties of a trigger for us, and we will perform some action based on that.

As you know with electronic circuits, they have to be complete to operate. In case of a button, we are bringing a new element to our design, where one branch will lead to an open circuit when the button is not pressed.


In this kind of situation, it's important to tie that input to either high or low state, so that GPIO can still properly read the state. If we don't do that, the pin is said to be floating, and it will give us erratic results, thus it's programming will be impossible.

In this example, we will use a **pull-up resistor**. This way when the button is pressed, the input will read low (1K resistance from R1 is infinitely higher than conducting S1 to ground), and high when the button is not pressed. The schematic will look like the following figure:



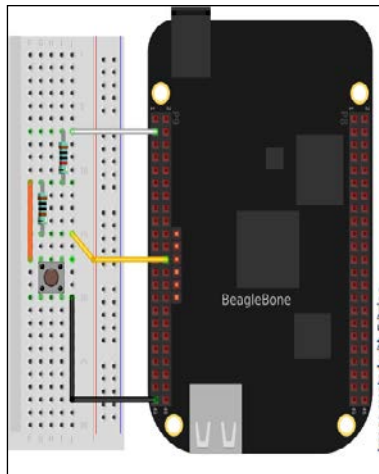
You can clearly see that when the button is not pressed, the pin is driven high (the actual amount of current is not relevant, only the voltage is). And when the button is pressed, a logical 0 will be read.

Our button in this case will be just a regular four-leg button. This specific button is wired in a way that all the four legs will conduct when the button is pressed.

[ You can usually take a look at *belly* of the button to get an idea how it's wired.]

Now to the actual wiring. This time we will use P9 on the left side, mostly for aesthetic reasons since we need 3.3V output for this circuit and it is on P9. Please be careful not to mix the resistors, and you should be good to go.

The wiring will look as shown in the following figure:



Next we will write the program to operate this button. Let's call it `first_button.py`.

As before, we need to include our GPIO library in the beginning:

```
import Adafruit_BBIO.GPIO as GPIO
```

This time we will be setting our GPIO as input, so our `GPIO.setup` call is slightly different:

```
GPIO.setup("P9_24", GPIO.IN)
```

We will need a flag to help us to distinguish from the previous state (transition from high to low):

```
previous_state = 1 #High is the default
```

And then in the same way, we'll have a eternal while loop that just waits for the transition from high to low, and prints a message. The full listing of `first_button.py` is shown in the following code:

```
#!/usr/bin/python

import Adafruit_BBIO.GPIO as GPIO
import time

GPIO.setup("P9_24", GPIO.IN)

previous_state = 1 #High is the default

while True:
    new_state = GPIO.input("P9_24")
    if new_state == 0 and previous_state == 1 :
        print('Button pressed!')
        previous_state = new_state
```

Run it, and see how it works. Now our button should react to the button press, by writing `Button pressed!` to the console window.

Did you notice anything particular? Try running it a bit more.

Sometimes we get the printout message twice, don't we? This is because of the physical characteristics of the button. Internally, it has a lead that touches another lead when you press the button. But because of the physics involved, the leads will sometimes bounce slightly off each other and just enough for our program to register a difference in states.

To avoid this, we can add a small delay to the function after the state change is recognized, so that the while loop doesn't read the next state immediately. Add a tiny (0.1s) delay to the if statement, and that should fix it!

Now that we have a working output LED and an input button, there are already quite a few interesting programs we could make. Do you think you can make the button to drive the LED on and off? How about a simple access control with several buttons that you have to press in correct order for entry? It's quite amazing how much you can achieve with simple 3.3 voltage and a few input and output pins.

Pulse width modulator

As you know by now, we can drive our GPIOs to high or low state, and if we have an LED attached in our circuit, it will either light up or not. We can control the brightness by adjusting the resistance in the circuit, but this is a static way to do it, and once our circuit is complete, we can't change the resistor easily.

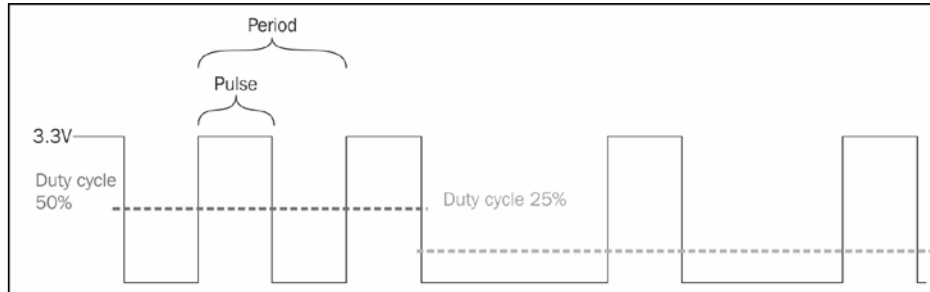
So, how can the LEDs on your wrist watch, mobile phone and remote control then fade in and out so perfectly? This is where pulse modulation comes in.

PWM allows us to drive a certain pin to on and off at desired frequency and duty cycle. This way we can pulse our LEDs much faster than our eyes can react, and while we only see a dimmer or a brighter LED, if one would look at it with a high speed camera, one would see that the LED still only turns on or off. Our eyes just perceive this differently.

There are three basic terms you need to understand about PWM:

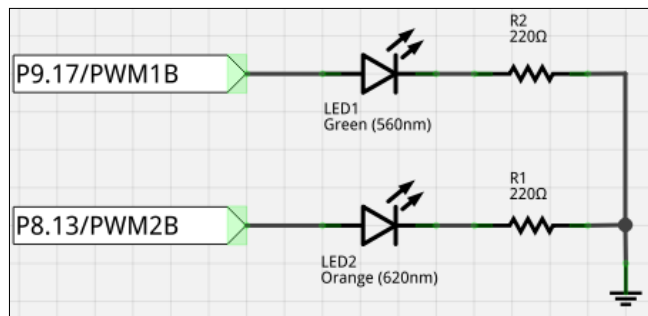
- **Frequency:** This defines how many full on/off "cycles" are generated in a second
- **Period:** This defines how long one complete pulse cycle takes
- **Duty cycle:** This specifies the time the signal is high during one full period

Think about the following figure:

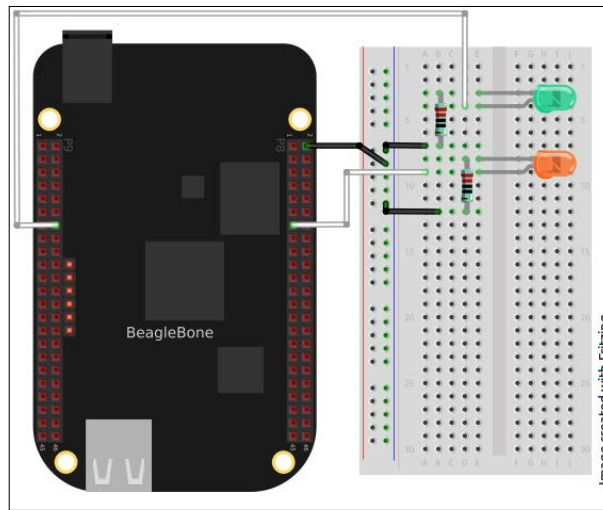


In the preceding figure, we have PWM generating first a 50 percent duty cycle and then dropping to 25 percent. The effective time the LED spends as on and off can be controlled with very high precision, and this way we can achieve smooth brightness fluctuations

Let's try doing just that. First, we will design a schematic with two LEDs that are connected to two different PWMs. Nothing fancy here either really, we have a current limiting resistor after the LEDs and that's it.



This time we will be using input from both headers, as PWM1 and PWM2 are located on P9 and P8, respectively.



Our third and last program in this chapter will be called `racing_PWMs.py`. As usual, we need to include the Adafruit library here as well, this time the PWM part of it:

```
import Adafruit_BBIO.PWM as PWM
```

When you initialize a PWM, you can initialize it with two to four parameters listed as follows:

- Channel (header pin)
- Duty (as in percent, 0-100)
- Frequency (periods in a second, default is 2000)
- And polarity (0 or 1. With this you can invert the duty cycle, default is 0)

So, we will initialize both our channels:

```
PWM.start("P9_14", 50, 100) #50% duty and 100hz cycle
PWM.start("P8_13", 50, 100)
```

At this point, the LEDs will light up, and now we can start changing our duty cycle in a loop to adjust the average voltage.

Full listing of `racing_PWMs.py` is as follows:

```
#!/usr/bin/python

import time
import Adafruit_BBIO.PWM as PWM
```

```
sleep_time=0.005 #The lower the value the faster the activity

PWM.start("P9_14", 50, 100) #50% duty and 100hz cycle
PWM.start("P8_13", 50, 100)

while True:

    for i in range(100,1, -1):
        PWM.set_duty_cycle("P9_14", i) # Dimming
        PWM.set_duty_cycle("P8_13", abs(i-100)+1) # Getting brighter
        time.sleep(sleep_time)

    for i in range(1, 100):
        PWM.set_duty_cycle("P9_14", i)
        PWM.set_duty_cycle("P8_13", abs(i-100)+1 )
        time.sleep(sleep_time)
```

When you run the program, you should see both LEDs racing (blinking) in opposite phases.

As you see, the Adafruit BBIO library is extremely useful and easy to use. And so far we have only used two functionalities it provides. Actually, the library also supports easy access to SPI and I2C communication and **Analog to Digital Converter (ADC)** as well. ADC use will be demonstrated in *Chapter 4, Extending Server Capabilities*, and in *Appendix, Security, Debugging, and I2C and SPI*, we will show the usage of the data bus communication modules.

Summary

In this chapter we went through foundations of input and output on a very basic level. We talked about the general purpose I/O pins, and how they can be used to control external components, such as LEDs or buttons. As you must have gathered already, with proper application of voltage and care, we can operate many of the basic electronic components. You should now feel comfortable with the basic building blocks that can help you build some external inputs and outputs to your Beagle programs.

As you have noticed, it's not really all that difficult, and we're sure you have already built more complicated applications with these elements that we introduced here. Did you get that access control program working? We added a buzzer to ours, and it was fun seeing people trying to guess the code, and keep getting the fail beep!

Next we will bring some remote elements to our programs and get our host computer talking with our Beagle over the network!

3

Creating the Client and Server Applications

So far, we have been working directly on the board with the use of SSH to log in to the system. However, this is not feasible in many scenarios, and we don't want to limit the interface to this type of console-based access.

In this chapter, we will leave lower-level electronics aside for a moment, and start talking a little bit about network programming. Since our target is basically a headless system (in this case, a system with no direct output device), we will need some way of interfacing with it.

Perhaps we also don't want to force ourselves too tightly into a single platform, but would like to have a means to communicate with different types of computing platforms. So, how does one go about creating a platform-independent remote access?

For this, we will use sockets, and we will implement ours on top of the **TCP/IP** stack so that we will be able to establish them over a network. We will cover the following topics:

- Discussing TCP/IP socket principles
- Creating a socket application to retrieve a web page
- Implementing server and client applications that can talk to one another via a TCP/IP socket

Sockets

If you are not familiar with sockets, we will cover the most important concepts in this section, and you will get a good sense of how they work. If sockets are already familiar to you from another programming language and/or environment, you can skip right ahead to the *Example socket application* section.

You can think of sockets as bidirectional communication pipes. Sockets are the main method of communication between two software modules, especially over a network. On an operating system level, there are other, more efficient methods of communication between processes (inter-process communication), but when communication needs to be established between different machines, perhaps even different platforms, sockets are pretty much the only option.

If the communication method is agreed between programs, there is no reason why different architectures can't talk to each other, even when the code base is different. This means that a C++ program that is running on a server machine can talk to a Python program on Beagle without any extra effort.

When you open up your web browser and connect, for example, to `http://beagleboard.org`, actually a socket connection is opened to a server that is running on the `beagleboard.org` host. Once this socket is established, the data transfer between your web browser and the server can start, and the information about the HTML page on that host is transmitted to you. The same thing happens when you start a Skype session with your parents, or start Snap chatting with your friends.

Sockets can be divided into two parts, a **client socket** and a **server socket**. The main differences are as follows:

- A client is basically an endpoint of communication.
- A server exists to serve information to clients.
- The client initiates connections towards the server, and the server serves these connections.
- A server can host many clients simultaneously if support for this is programmed, but a client only talks to one server socket at the time. Of course, nothing stops you from starting multiple client connections to the same or different servers, but it's still a one-to-one communication from a client point of view.



In this book, we will focus on stream sockets on top of the TCP (and IPv4) stack, but sockets can also be established over UDP. Those are called datagram sockets. There exists a third type of socket called **raw socket** that gives access to lower-level socket stream methods, and can be used, for example, to create a custom communication protocol.

Stream sockets, often called **Internet sockets**, are very user friendly and easy to understand once you just start thinking of them as read and write buffers. All the magic that makes the bytes move is taken care of for you in the background.

With these basics, let's take a look at implementing few sockets on our own.

An example socket application

Let's take the web browser use case, which we talked about, as an example, and create a very simple web page fetcher. The purpose of this program will be just to show how a socket data can be handled so that you can become familiar with working with stream data. So, go ahead and create a program called `get_webpage.py`.

First, we include the socket library in our program:

```
import socket    #For access to socket communication
import sys       #To terminate with error code
```

Then, to initialize our client socket, we create the stream `INET` socket:

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

The first parameter of the `socket()` function call is the address and protocol family that we want to use, in this case, Internet Protocol. The second parameter is the type of socket that we are going to create.



If you are interested, you can take a look at the socket documentation at <http://docs.python.org/2/library/socket.html> for other types of sockets that Python Version 2 supports.

Next, we will need an IP address to connect to the web server. For this purpose, the socket library has a method called `gethostbyname()`. Resolve the IP address with it using the following code:

```
host = "www.beagleboard.org" #The address we will connect to
port = 80 #The default port for web-server
print "Connecting to %s" % host
```

```
# Use socket library to retrieve ip address for www.beagleboard.org
try:
    target_ip = socket.gethostbyname( host )
```

After this, we are ready to connect to the server:

```
client_socket.connect((target_ip , 80))
```

After the connection is created, we will send a message to the server requesting the default page with the HTTP GET method:

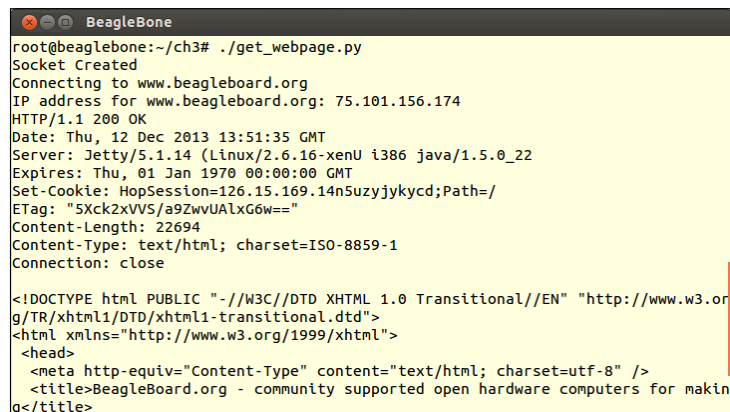
```
# Send a simple HTTP-GET message
msg = "GET / HTTP/1.0\r\n\r\n"

client_socket.sendall(msg)
except Exception, err:
    print "Socket connection has failed:"
    print err
    sys.exit(2)
```

After the message has been sent, we read the reply (or in this case, part of it) using the following code:

```
# Ask the socket to retrieve the first 1024 bytes
reply = client_socket.recv(1024)
print reply
client_socket.close() # Close the socket afterwards
```

Basically, these are the only lines of code that you need when reading a web page from the Internet. Before you execute this program, make sure that the Ethernet cable is connected to the board, and that you have connectivity to the Internet. As you can see, the socket library is extremely helpful in hiding a lot of complexity for us. When running the program, you can see an output similar to the following screenshot



```
root@beaglebone:~/ch3# ./get_webpage.py
Socket Created
Connecting to www.beagleboard.org
IP address for www.beagleboard.org: 75.101.156.174
HTTP/1.1 200 OK
Date: Thu, 12 Dec 2013 13:51:35 GMT
Server: Jetty/5.1.14 (Linux/2.6.16-xenU i386 java/1.5.0_22
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: HopSession=126.15.169.14n5uzyjkykcd;Path=/
ETag: "5Xck2xVVS/a9ZwvUAlxG6w=="
Content-Length: 22694
Content-Type: text/html; charset=ISO-8859-1
Connection: close

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>BeagleBoard.org - community supported open hardware computers for makin
g</title>
```

When you run this program, you will see the partial reply from the beagleboard.org (because we only read the first 1024 bytes), but you could easily change the program to fetch the whole page, create a file from it, and your web browser could render the page correctly.

Next, let's start working on something permanent for us.

Echo server

Now that you have a basic understanding of how a socket can work, we will create our own custom server and another client so that we have full control over both parts of the transaction.

All the examples that we have created so far have been more or less demonstrative in nature. Let's start building something larger and more functional now, and also re-use some code from *Chapter 2, Input and Output*, as well. We will need to modularize some of our previous code so that they can be easily re-used and extended.

Create a file called `server_socket.py`. This utility class won't do much by itself; its sole purpose will be to set up and manage the incoming socket connections. First, let's define a method called `initialize_server`. In it, we create and bind a socket in much the same way as we did in the client:

```
def initialize_server(host, port):

    print "Starting server on %s:%s" % (host,port)
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    #Set the socket reuse to 1, so that socket terminates quicker
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    try:
        srv.bind((host, port))
    except socket.error , msg:
        print "Bind failed. Error: " + str(msg[0]) + "\nMessage: " +
            msg[1]
        sys.exit(-1)

    srv.listen(1) #Currently only one client at the time
    return srv
```


After the `initialize_server` function is called, a socket is bound, and a handle to the created socket is returned to the calling function. It will be the caller's responsibility to handle the operation and termination of this socket.



We set an option `SO_REUSEADDR` for the socket to terminate the bind quickly once the application exists. This is not necessary, but it's a good-to-know feature when you are developing your application and possibly restart the server often. Otherwise, the bind will still stay in the OS until it times out. There are other socket options that you might want to look into as well. These are Linux specific; so, if you're running your server on a Windows machine, keep in mind that they will be different there. Flag definitions and explanations can be found at: <http://man7.org/linux/man-pages/man7/socket.7.html>

We also need a method to start waiting for incoming client connections. It's very simple, as we only need to call `socket.accept()` function:

```
def wait_for_client(server_socket):  
  
    (client, c_address) = server_socket.accept() #blocking wait for a  
    client  
    print "Received a connection from [%s]" % c_address[0]  
  
    return client
```

For now, we don't need other functionality in our `server_socket` class. Next, we will create our server logic. Create another class called `echo_server.py`. Because we will import our `server_socket` class, it needs to be in the same folder as the `server_socket.py` file.

In the `echo_server` class, we will create, handle, and terminate all communication with the clients. So, we first import our `server_socket` class:

```
#!/usr/bin/python  
  
import server_socket  
print "Starting echo server.."
```

Then, we initialize our socket:

```
srv = server_socket.initialize_server("", 7777)
```

Notice how we set the server address to `"`. This is to indicate that we will be using our `localhost`, and accept connections from outside networks as well. If we would set `localhost` here directly, the server would actually only ever accept connections from other programs running on this particular host.



The port, 7777, was arbitrarily chosen. Actually, you can enter any port number except those that are already being used by the system to listen to other communication methods. Some ports have been clearly defined for certain protocols. For example, FTP connections are expected in port 21 and SSH in 22. HTTP protocol expects you to use port 80. You can find a list of these on Wikipedia: http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers.

In general, choosing anything over 1023 is considered safe. If your OS complains that the socket is already taken, just choose another one.

After we have initialized our socket, we have to start considering client handling. This is something that we will come back to later as well, but for now, our `echo_server` will just use an eternal `while` loop to always welcome new friends for a chat:

```
while True:
    print "Waiting for someone to chat with..."

    # This will be a blocking wait
    client = server_socket.wait_for_client(srv)

    client.send("This is an echo server. Let me know if you want to
        quit with 'quit' or 'exit'")
```

When the `wait_for_client` function returns, we have a connected client, and we send him a welcome message letting him know the rules of the chat. Next, we will go into another loop, where we react to the input from client:

```
exit_requested = 0

#Now client has connected, and let's start our messaging loop
while not exit_requested:
    data = client.recv(1024) # Always attempt reading 1024 bytes

    print "received [%s] from client" % data
    #check the message
    if "quit" in data or "exit" in data:
        exit_requested = 1
```

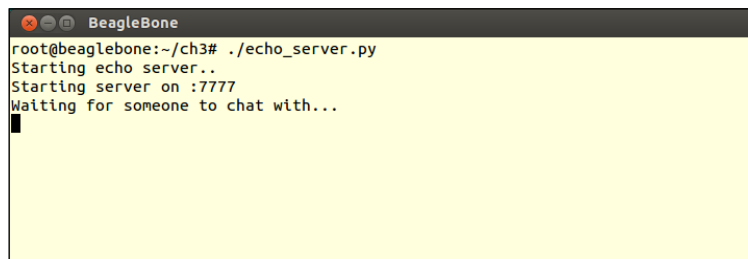
```
else:
    print "replying..."
    msg = "You wrote [%s]" % data
    client.sendall(msg)
    # Disconnect the client when we exit the while loop
    client.send("thanks for coming!")
    client.close()
```

You can see that in our while loop, we always try to read messages from the client, and once we receive one, we check whether the user wants to end the connection with the server. If not, we send his message back, and start waiting on the next one.



Notice how the server is blocked by our `recv()` function call. The server always waits for a message from its client until it receives one. Often this is OK, but, for example, if our server is supposed to serve multiple clients, we might want to use non-blocking style, and timeout idling clients. You can implement non-blocking reads by setting the `setblocking(0)` option to the server socket. You will need to catch the exceptions, `socket.EWOULDBLOCK` and `socket.EAGAIN` when there is no new data available from the `recv()` function.

Now that our server is ready, we can start it as shown in the following screenshot:



```
BeagleBone
root@beaglebone:~/ch3# ./echo_server.py
Starting echo server..
Starting server on :7777
Waiting for someone to chat with...
```

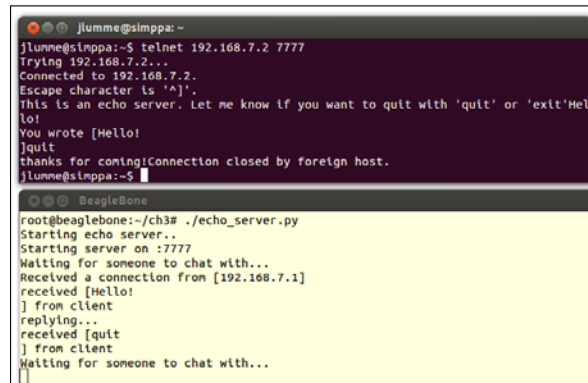
Our server is now running on Beagle, waiting for some friends to chat with.

Echo client

In this section, we will create a Python client for our echo server, but actually, before we go ahead and do that, it might be interesting to see that our echo server can already talk to different types of client programs as well.

Our server just expects a socket to communicate with. It neither knows nor cares what kind of client connects to it. This means that actually we can use a regular telnet to have a chat with our server.

On Windows, start your Putty client, select the raw mode, and make a connection to 192.168.7.2 and port 7777. On Linux and Mac OS X, just type a command, `telnet 192.168.7.2 7777` in the terminal, as shown in the following screenshot:



```

j1umme@slmpps: ~
j1umme@slmpps:~$ telnet 192.168.7.2 7777
Trying 192.168.7.2...
Connected to 192.168.7.2.
Escape character is '^['.
This is an echo server. Let me know if you want to quit with 'quit' or 'exit'Hello!
You wrote [Hello!
]quit
thanks for coming!Connection closed by foreign host.
j1umme@slmpps:~$

root@beaglebone:~/ch3# ./echo_server.py
Starting echo server...
Starting server on :7777
Waiting for someone to chat with...
Received a connection from [192.168.7.1]
received [Hello!
] from client
replying...
received [quit
] from client
Waiting for someone to chat with...

```

As you saw, our echo server was communicating with a telnet client without problems (well, a couple of formatting issues aside). This is indeed the power of sockets. With some care, it's not difficult to use them to create cross-platform communication.

Now, let's get back to Python and create our `echo_client`. At this point, we suggest that you create the application on your host machine. This is not mandatory, but it's more fun that way, and you will be able to see the network aspect of the communication better.

Most Linux and Mac OS X distributions come with Python pre-installed in them in much the same way as on your target board. However, on Windows, you will need to install a Python interpreter. We will not go over the Python installation in detail, but it is quite easy. Just download the correct Windows package from <http://www.python.org/getit/>, and install it. We are using Python 2, not 3, in all our examples. There are some differences in some libraries; so, we suggest that you install version 2 as well.



With Windows Python installation, you will need to add the installation directory to the PATH variable. After that, you can run Python programs using command line, which is shown as follows:

```

C:\Python27\BBB>python test_python.py
Python is working!

C:\Python27\BBB>

```

Our echo client will also be a rather straightforward program. It will just connect to the server, read input from the user, and send that message to the server. Let's create our `echo_client.py`.

First, create a new client socket, and connect to 192.168.7.2 using the code shown in the following code snippet:

```
#!/usr/bin/python

import socket, time

# Initialize the socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(("192.168.7.2" , 7777))
```

After the connection is established, we will read the initial welcome message and then start expecting input from the keyboard to be sent to the server:

```
print "Connection established!"

reply = client_socket.recv(1024)
print reply

keep_connection = 1
while keep_connection:

    msg = raw_input('Enter your input:')
    if msg == "quit" or msg == "exit":
        keep_connection = 0

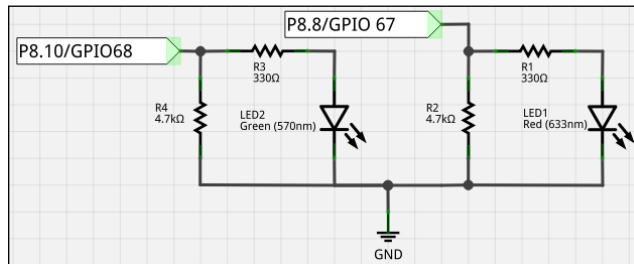
    client_socket.sendall(msg)
    time.sleep(0.5)
    reply = client_socket.recv(1024)
    print "Server replied: %s" % reply

print "Connection terminated"
```

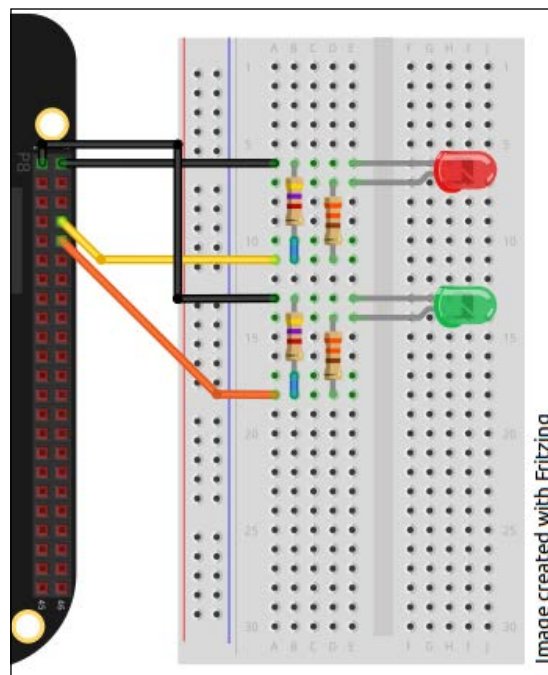
Now, you can start the client on your PC, and watch it connect to our Beagle. The client will terminate once you type `exit` or `quit`, but the server will just drop the connection and start waiting for another client.

As our echo server currently cannot handle multiple clients connecting to it, we don't really know whether the server is currently busy serving a client. But maybe there is some other way which could indicate if the server is busy. You guessed it, an LED!

Our `ext_led_fun.py` program in *Chapter 2, Input and Output*, was a stand-alone program; we can use it as our baseline for creating a new utility class that can be used easily by other programs to control LEDs that are connected to GPIO pins. The circuit can remain largely the same as the one that we used in *Chapter 2, Input and Output* besides adding another LED to the circuit, as shown in the following figure:



The wiring diagram will look similar to the following figure:



Create a new class called `led_control.py`. In this utility class, we will create the functions, `control_led_green(value)` and `control_led_red(value)` that will drive their respective LEDs. There is one thing to keep in mind though; we need a function to initialize the respective GPIOs since we can't drive them otherwise. So, we need an initialization function:

```
# A global flag, that we will use to force initialization
gpio_initialized = 0

def initialize_gpio_leds():
    global gpio_initialized

    GPIO.setup("P8_8", GPIO.OUT)

    GPIO.setup("P8_10", GPIO.OUT)

    print "led_control/initialize_gpio_leds(): GPIO 67 initialized as
        output"

    print "led_control/initialize_gpio_leds(): GPIO 68 initialized as
        output"

    gpio_initialized = 1
```

Then, create functions to control both the green and red LEDs:

```
def control_led_green(value):
    global gpio_initialized

    if not gpio_initialized:
        raise Exception("GPIO uninitialized!")

    if value:
        GPIO.output("P8_10", GPIO.HIGH)
    else:
        GPIO.output("P8_10", GPIO.LOW)

def control_led_red(value):
    global gpio_initialized

    if not gpio_initialized:
        raise Exception("GPIO uninitialized!")
```

```
if value:
    GPIO.output("P8_8", GPIO.HIGH)
else:
    GPIO.output("P8_8", GPIO.LOW)
```

Now, in our echo server, we can import `led_control` (remember that the `led_control.py` file has to be in the same folder), and after calling the initialization function, trigger the red and green LEDs to indicate the server status.

Summary

In all their simplicity, stream sockets are not too difficult as you see. There are some other matters that demand attention if full cross-platform compatibility is desired, but for now, this should give you a good understanding of how you can create your own client-server communication.

We also created some utility classes here that we will be able to import and use in the future chapters as well.

As a challenge, you could also extend our LED control class to include the PWM control with user-defined pulse times (in seconds), and don't forget to include the proper LED triggering to our `echo_server.py`!

In the next chapter, we will introduce two different environmental sensors and show how to operate them with the use of analog-to-digital converters on Beagle. We will also start building our home automation server with capabilities to send miscellaneous data to our clients.

4

Extending Server Capabilities

Now that we have a simple networked client and server architecture working, it's time to start doing something a bit more serious with it. Currently, the server is not very useful in the end and cannot really do any complex tasks or handle any real input.

We shall start extending server capabilities to make it more useful for real-life tasks. We will show you how to build a foundation so that the server can easily be extended even further for any type of remote activity.

In this chapter, we will also introduce some new external and internal hardware and learn how to work with them. We will cover the following topics:

- Learning how to use temperature and light sensors
- Using the onboard ADC
- Creating a more robust client server model, one that can support data transfer
- Talking about transistors and how they allow us to be free of voltage and current limitations our target board sets

Environmental sensors

Up to this point, we have only been working with digital input to our board. In the button example covered in *Chapter 2, Input and Output*, the external hardware input only gave us the state of the button, a high or a low state. In this chapter, we will talk about analog input; we will measure the voltage on a pin and observe its change over time. For this purpose, our board has several pins that are connected to ADC. ADC converts analog voltage to a digital value, instead of just indicating a high or a low state.

Environmental sensors, such as temperature or light sensors, change their output voltages depending on the surrounding conditions. This way, for example, a temperature of 25 degrees will result in a different output voltage than 15 degrees. For our home automation server, there are a couple of interesting sensors.

Light sensor

These sensors, also known as photocells/photo resistors or **Light Dependent Resistors (LDR)**, are resistors whose value changes depending on the amount of light received by the sensor. The resistance of the resistor decreases as the light intensity increases.

They are generally not very accurate and cannot really be used to measure precise candela or lux values. But they are still accurate enough, for example, to determine whether it's day or night and the general brightness. In real life, for example, street lights might contain this type of circuit so that they are activated only during night time.

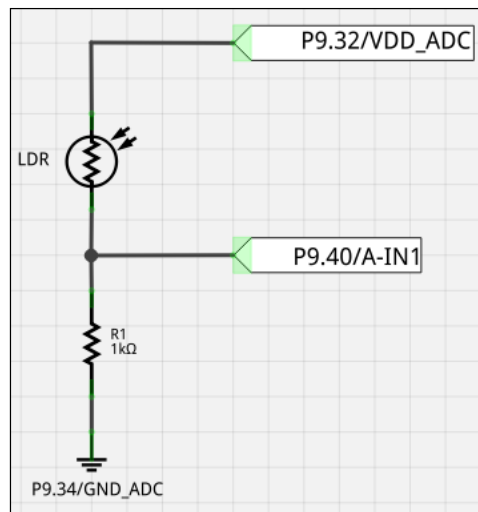
The resistance values will vary depending on the specification, and you should refer to the specification sheet of the resistor you have; however, in general, you can assume that we are talking about a range of couple hundreds to 1000 Ohm when the light is on, and in the magnitudes of 10 KOhm when the light is off.

Our ADC can only read the difference in voltage and not in current, so we will need to form a potential divider circuit between LDR and a pull-down resistor. When we add another resistor in series with LDR to the ground, the voltage drop across our pull-down resistor will be affected by the resistance in LDR.

It's also good to know that LDRs in general are not too fast, so they are not suitable for measurements where the reaction speed is critical. They might quickly react to light being turned on, but when the light is turned off, they tend to take some time to settle back. However, we are still talking about some tens or a hundred millisecond range, so this slowness is also relative.

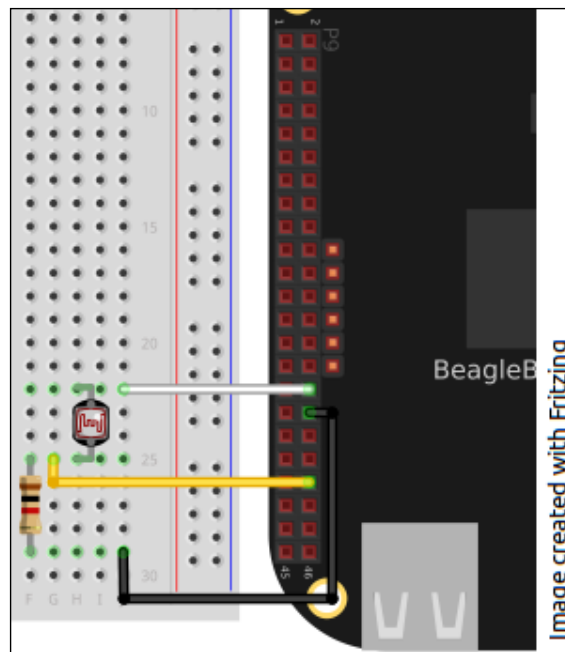
If you need time-critical responsiveness (let's say, for example, reacting to a laser pointer), you should look into photo diodes. They can have response times that are measured in nanoseconds.

So, let's design our light measurement circuit for the sensor. If you don't have specifications of your LDR at hand, you can try a resistor of around 1 to 2 KOhm for the pull-down resistor. Because we will connect our LDR resistor to the 1.8 V ADC output in the header **P9.32**, we don't have to worry about overloading our analog input pin **P9.40**.



Our circuit should now be pulled close to the ground when there is no light on the LDR, since 1 KOhm resistance will be smaller than the (assumed) 10 KOhm of the LDR. When light is detected by the LDR, the resistance should drop, and we should start seeing voltage increase in **A-IN1 (P9.40)**.

The wiring of the circuit looks as shown in the following figure:



Now that we have our wiring complete, we can start creating a program that will read these values. Let's first create a library class to read ADC values so that we can also access information easily from other programs. Create a class called `adc_control.py`. In it we will temporarily create two functions to initialize the ADC control and to read a raw value from a specified pin as shown in the following code snippet:

```
#!/usr/bin/python

import Adafruit_BBIO.ADC as ADC
import time

init_done=0
'''
Function init_adc

Initializes the ADC using the Adafruit_BBIO library
'''
def init_adc():
    global init_done

    print "Initializing ADC"
    ADC.setup()
    init_done = 1

'''
Function read_raw_analog_input

Reads the raw value from requested analog input pin.
'''
def read_raw_analog_input(pin_no):
    global init_done

    if not init_done:
        init_adc()

    reading = ADC.read(pin_no)
    return reading
```

The idea is to extend this class later so that we can encapsulate different methods inside it (such as reading the temperature and so on). This is so that the caller won't have to worry about transformations and so on.

As you can see, again thanks to Adafruit library, using ADC is just as simple as working with GPIOs. Only initialization is needed, and then values from A-IN pins can be read.

Let's take a look at how to read values from our light sensor. Create a class, for example, `reading_light.py`, and add the following code there:

```
#!/usr/bin/python

import led_control, adc_control
import time

print "Reading pin 9_40!"
while True:
    val = adc_control.read_raw_analog_input("P9_40")
    print "raw: %f | d: %d" (val, val*100)
    time.sleep(0.5)
```

The value (`val`) that is returned will be a floating point value somewhere between 0 and 1. To get the voltage read, you can multiply this by 1.8 if needed. In our example, we will multiply this value by a hundred to get the constant value, which could be easily compared. Run the program, and see how the value changes as you cover the sensor. The output will be as follows:

```
root@beaglebone:~/ch4# ./reading_light.py
Reading pin 9_40!
Initializing ADC
raw: 0.713889 | d: 71
raw: 0.463889 | d: 46
raw: 0.465556 | d: 46
raw: 0.465000 | d: 46
raw: 0.192222 | d: 19 #sensor covered here
raw: 0.050556 | d: 5
raw: 0.048889 | d: 4
raw: 0.048333 | d: 4
raw: 0.045556 | d: 4
raw: 0.108889 | d: 10 # sensor uncovered here
raw: 0.464444 | d: 46
raw: 0.462778 | d: 46
```

Play around with the code a bit and test different sleep values so that you can see the responsiveness of your sensor, and see how different light levels affect the result in your environment.



With this knowledge, you could easily create, for example, a light control system that turns on the lights on the outside once it starts getting dark. Have a go and implement a prototype of that system with the help of our `led_control` library we created in the previous chapter!

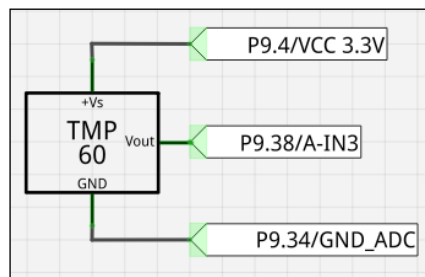
Now that we can measure light, let's look into measuring the current temperature.

Temperature sensor

These sensors are also not difficult to use, and actually from the programming point of view, the measurement method is exactly the same. We only need to read the voltage with one of our analog input pins, and then only some math is needed to get the temperature in Celsius (or Fahrenheit).

As our hardware component of choice, we will use a temperature sensor LM60 (<http://www.ti.com/product/lm60>) from Texas Instruments (you might also still find it under the National Semiconductor brand). LM60 is actually a small IC (**integrated circuit**) with three legs, and you have to take care to wire it correctly.

From the data sheet, we can see that the sensor works on voltage between 2.7 V and 10 V. So we can use our 3.3 V in P9.4 as a power source for it. The same sheet also tells us that the maximum output of this part is ~1200 mV, so we can safely use it with our analog input pins. Let's use **A-IN3** in **P9.38** as our analog input, and connect the LM60 ground pin to **P9.34**. Our schematic will look as shown in the following figure:

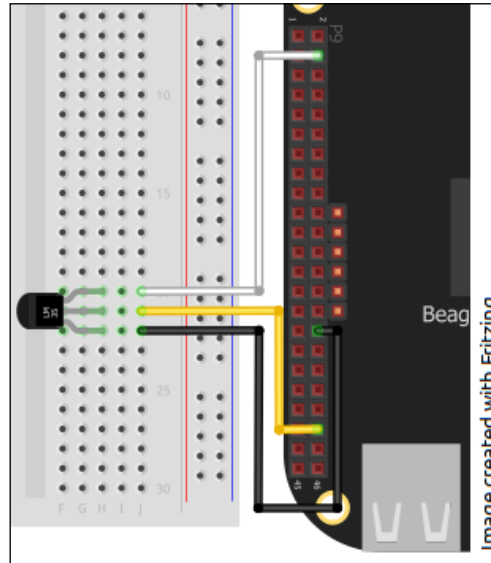


The output from **Vout** depends on the surrounding temperature, and it can be calculated from the formula: $V_{out} = (6.25 \text{ mV} * C) + 424$. So, if C is 25 degrees Celsius, the output would be roughly around 580 mV.



The offset of 424 mV is not arbitrary; it is there for our convenience so that we can also measure negative temperatures easily.

Let's do our wiring as shown in the following figure:



Our example program will be very much similar to `reading_light.py`. Create a new class called `reading_temp.py` with the following changes for the temperature reading:

```
#!/usr/bin/python

import adc_control
import time

while True:
    val = adc_control.read_raw_analog_input("P9_38")
    mv = val*1.8*1000
    celsius = (mv - 424) / 6.25 #for LM60
    print "%dmV ~%dC" %(mv, celsius)
    time.sleep(0.5)
```

Go ahead and run the program, and play around, for example, with a hair dryer or some ice cubes (but be careful about the ice melting on the circuit!).

After these two exercises, we have some interesting data readily available. But wouldn't it be nice if we had this data readable remotely? This is exactly what we will do next, when we start extending our server and client applications!

Advanced server

In the last chapter, our chat server was fairly simple and couldn't really do anything genuinely useful, but it served as a nice primer into socket communications.

Now we will start getting more serious about socket communication and server functionality. We shall implement a client server framework that transmits real-time information from the server and also creates a base for future extensions of supported functionalities.

The first thing when more advanced communications are expected between clients and servers, a protocol needs definition. There are different approaches to writing a communication protocol between programs. You can roughly divide them in two categories: human readable protocols and binary protocols. Human readable protocols are easy to understand and debug, but they are perhaps somewhat wasteful in resources as they use more space and have more data to transmit. In this book, we will stick to a human readable one. As a learning exercise, human readable protocol is also unbeatable.



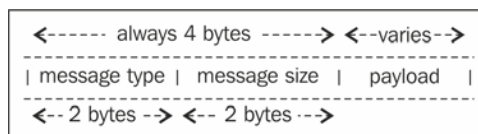
When maximum efficiency is required from a protocol, it is desirable to minimize different overheads and, for example, use bit masking to convey data. You can take a look at how, for example, TCP/IP protocol is implemented at the protocol level to learn further.

Defining our Beagle protocol

So, when defining a protocol, the most important thing to consider is "What are we planning to achieve with this?"

In our case, we want to transfer data from the server to a client and perhaps give orders remotely. So, let's define the first version of our protocol to accommodate those features.

Our protocol shall look as follows:



We will not implement any kind of error-checking or packet-termination sequences. The protocol will solely rely on the fact that our data packet will always have the initial four bytes that will let the receiver know about the optional trailing data. You will see that even now this is actually quite sufficient, since we can already rely on TCP/IP for error correction, and in our server, we do not need complicated state information inside the transfers.

Basically, we divide our protocol into two main types. They will be requests (`MT_REQUEST`) and replies (`MT_REPLY`). Both of them will have a minimum payload of two bytes, but it can be more if necessary.

Create a file `beagle_protocol.py`. First we will add the following definition for our message types:

```
MT_REPLY_SHORT          = 10 #Payload limited to 2 bytes
MT_REQUEST              = 13
```

For requests, we also need to define actions; for now, we have the following two actions:

```
ACTION_READ_TEMPERATURE = 1
ACTION_READ_LIGHT_LEVEL = 2
```

There are also a couple of special messages and variables defined in our protocol as follows:

```
MT_INFO_INITIAL_HELLO    = 15
MT_DISCONNECT            = 0

# Fixed payload size for a single command/reply
PAYLOAD_FIXED_SINGLE_VALUE = 2
PROTOCOL_VERSION = 1
SUPPORTED_ACTIONS =
    (ACTION_READ_TEMPERATURE, ACTION_READ_LIGHT_LEVEL) #for client
```

You must be wondering why we chose values that are not sequential. This was on purpose, as in the next chapter, we will add more message types, and they will line up at that time.



Notice how we also defined our protocol version there; this can be useful if in future, for example, we add new features to our protocol and either the server or client gains new features. This way we can leave the door open for working with older client versions that now do understand how to take advantage of new features.

The new server code

While the basic logic behind transmissions remains the same, data transfer is quite a bit more complicated compared to `echo_server` we created previously. So we will create a new application called `beagle_server.py`.

But first, you should extend your `adc_control.py` so that it will have methods to read the current temperature and light level. We will use these methods in our server to read the current data when requested. Don't forget to import this interface to the main server once you're done.

After you have finished creating the new methods in `adc_control_extended.py`, let's create our server application called `beagle_server.py`, add the include statements, and define our main method as follows:

```
import server_socket
import beagle_protocol as BP
import adc_control_extended as ACE

from struct import *

if __name__ == "__main__":
    print "Starting beagle server.."
    srv = server_socket.initialize_server("", 7777)

    # Eternal while loop that just waits for clients
    while True:
        print "Waiting for a user to connect."
        client = server_socket.wait_for_client(srv) # blocking!

        # Inform our protocol version in the hello message
        data_packet = create_data_packet(BP.MT_INFO_INITIAL_HELLO, "",
            BP.PAYLOAD_FIXED_SINGLE_VALUE)
        client.sendall(data_packet)

        #Now we start waiting for commands from client
        keep_alive = 1
        while keep_alive:
            keep_alive = handle_client_request(client)

        # If we're here, it means we should let the client go
        client.close()

    srv.close()
```

First, the server sets up the serving socket (imported from our `server_socket.py` class, which we created in the *Echo server* section in *Chapter 3, Creating the Client and Server*) and starts waiting for a client to connect. When it detects a connection, the server sends a "welcome message" to the client, informing of the current protocol level—we will take a look at creating a data packet shortly. After the server has sent the welcome message, it goes into a service loop where it will wait for commands and serve them in the `handle_client_requests` method. Let's take a look at that method (define it above our main method):

```
def handle_client_request(cs):
    try:
        msg = cs.recv(4) # Retrieve the header, blocking call
        header = unpack("!HH", msg) # decode the mandatory headers
```

Always, when handling a packet from the client, the server initially reads four bytes from the stream and decodes them. For decoding, the server uses the `unpack` method from the `package struct`. We inform the `unpack` method that we want to decode two shorts (`HH`) from `msg` in the big endian format (`!`) and place the result into a header tuple. As we know one short is of two bytes, we have our mandatory first four bytes decoded.



Endianness is a term used to define the way bits and bytes are stored in computer memory. For example, consider our `MT_REQUEST` header. It is two bytes, and these bytes are stored in memory. Big endian format means that the most significant byte is stored in the smallest (first) memory address. In little endian, this is the opposite.

Each platform has its "native way" of storing data, so when sending data over a socket, you need to consider that will the receiving end interpret the bytes in same order as you, and if not, one end will have to do conversion.

Fortunately for us, in *Chapter 6, Creating an Android Client*, when we create an Android client, Java uses same big endian format by default as Python, so this time we do not have to worry about this. You can find Python documentation about packing at <http://docs.python.org/2/library/struct.html>.

Now we proceed to identify the incoming message with the following code:

```
# Identify the message type
if header[0] == BP.MT_REQUEST:
    remaining = int(header[1])
    msg = cs.recv(remaining)
    header = unpack("!H", msg) #We know this is 2 bytes in MT_
REQUEST
    request = int(header[0])
```

Because we know that `MT_REQUEST` type is always six bytes, we don't necessarily have to check the remaining bytes from `header[1]`, but we do it here for consistency. After that, we can identify the message type and act on it as follows:

```
if request == BP.ACTION_READ_TEMPERATURE:
    curr_temp = get_temperature()

    data_packet =
        create_data_packet(BP.ACTION_READ_TEMPERATURE,
                           curr_temp, BP.PAYLOAD_FIXED_SINGLE_VALUE)
    if not data_packet == None:
        cs.sendall(data_packet)

elif request == BP.ACTION_READ_LIGHT_LEVEL:
    curr_light = get_lightlevel()

    data_packet =
        create_data_packet(BP.ACTION_READ_LIGHT_LEVEL,
                           curr_light, BP.PAYLOAD_FIXED_SINGLE_VALUE)
    if not data_packet == None:
        cs.sendall(data_packet)
```

Our server identifies the action requested by the client. It then first prepares the data (by calling the appropriate functions in this class, which will use the methods that you created in your `adc_control_extended.py` file) and then creates a reply message using the `create_data_packet()` function. Once the data packet is created, it sends the data to the open client socket.

If the message was not of the `MT_REQUEST` type, we of course check for other actions and handle the case where stream has been abruptly broken.

```
elif header[0] == BP.MT_DISCONNECT:
    print "User requested disconnection, closing connection"
    return 0

# By default, we continue for as long as client wants
return 1
except:
    print "Rude disconnect from user"
    return 0
```

Lastly, on the server side, we take a look at our `create_data_packet()` function (add it above the `handle_client_request()` function).

```
def create_data_packet(msg_type, data, data_length):
    packet = ''
    try:
        if msg_type == BP.MT_INFO_INITIAL_HELLO:
            packet = pack("!HHH", BP.MT_INFO_INITIAL_HELLO,
                          BP.PAYLOAD_FIXED_SINGLE_VALUE, BP.PROTOCOL_VERSION)
        elif msg_type == BP.ACTION_READ_LIGHT_LEVEL:
            packet = pack("!HHH", BP.MT_REPLY_SHORT,
                          BP.PAYLOAD_FIXED_SINGLE_VALUE, data)
        elif msg_type == BP.ACTION_READ_TEMPERATURE:
            packet = pack("!HHH", BP.MT_REPLY_SHORT,
                          BP.PAYLOAD_FIXED_SINGLE_VALUE, data)
        else:
            print "unrecognized packet type, ignoring"
            return None
    except Exception, ex:
        print "Something went wrong with packing"
        print ex
    return packet
```

Here we encapsulate our data with the help of the `pack` function. It works much the same way as `unpack` in just the opposite way. We tell it how to pack data and the data to be packed. For now, we only support three types of messages, and each of them is exactly of six bytes (three short values) in length.

This is how the server will operate. Next we will look at a client that can take advantage of these new fancy services the server is providing.

The new client code

Let's start creating our client named `beagle_client.py`. The main function of our client starts with establishing a connection to the server and reading four bytes from the welcome message. After this, we check that we are communicating as expected (we understand our server) before moving forward. Once we have identified that we indeed received the initial hello message, we read the next two bytes to check the protocol version of the server using the following code snippet:

```
if __name__ == "__main__":
    if not len(sys.argv) == 2:
        print "please specify target address"
```

```
sys.exit(2)

target_ip = sys.argv[1]
print "Connecting to %s" % target_ip

# Initialize the socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((target_ip , 7777))

print "Connection established!"

# Read the initial welcome message, to initialize our protocol
reply = client_socket.recv(4) # Retrieve the initial size
msg = unpack("!HH", reply)

if msg[0] == BP.MT_INFO_INITIAL_HELLO:
    remaining_size = int(msg[1]) # This is actually fixed
    reply = client_socket.recv(remaining_size)
    msg = unpack("!H", reply)
    print "Server protocol version: %s" % msg
    if not BP.PROTOCOL_VERSION >= int(msg[0]):
        print "WARNING> The server protocol is newer than ours."
else:
    print "We received message we can't understand, abort"
    sys.exit(2)
```



Notice that in this example, the client reads the target IP address from the argument list of the program. So when you start the program, you need to specify the IP address for the target board. In our case it is 192.168.7.2 if the board is connected directly to the development machine. Otherwise, please check the eth0 address on the target board with the command `ifconfig eth0`. Of course, for this to work, we need to have received an IP address from our router.

After the initial checks and handshake is complete, we can enter our action loop. The loop will be somewhat similar to the chat client example. The client will expect an input from the user, verify that the input is valid (and print help text if it's not), and then call the `send_message()` function to send a message to the server as follows:

```
while True:
    command = raw_input(':')

    #Valid input
    if not command == "" and int(command) in BP.SUPPORTED_ACTIONS:
```

```

        print "Selected %s" % command
        send_message(int(command), client_socket)
        read_reply_from_server(client_socket)

    #Disconnect
    elif not command == "" and int(command) == BP.MT_DISCONNECT:
        print "Disconnect requested"
        send_message(int(command), client_socket)
        break

    #Unrecognized input
    else:
        print_options()
    print "Connection closed"

```

As you can see, basically in a very similar fashion as the server, we have two functions that take care of communication with the server: one for sending a request and another to receive a reply. Let's first take a look at the `send_message()` message function.

In the following code snippet, we identify what type of message a client wants to send to the server, and then again, we pack our data and send it down the socket. The disconnect request is only of four bytes; other messages have a payload of two bytes.

```

def send_message(message_type, socket):

    packet = ''

    if message_type == BP.ACTION_READ_TEMPERATURE:
        print "Requesting for temperature"
        packet = pack("!HHH", BP.MT_REQUEST,
            BP.PAYLOAD_FIXED_SINGLE_VALUE, BP.ACTION_READ_TEMPERATURE)
        socket.sendall(packet)

    elif message_type == BP.ACTION_READ_LIGHT_LEVEL:
        print "Requesting for light level"
        packet = pack("!HHH", BP.MT_REQUEST,
            BP.PAYLOAD_FIXED_SINGLE_VALUE, BP.ACTION_READ_LIGHT_LEVEL)
        socket.sendall(packet)

    elif message_type == BP.MT_DISCONNECT:
        packet = pack("!HH", BP.MT_DISCONNECT, 0)
        socket.sendall(packet)

```

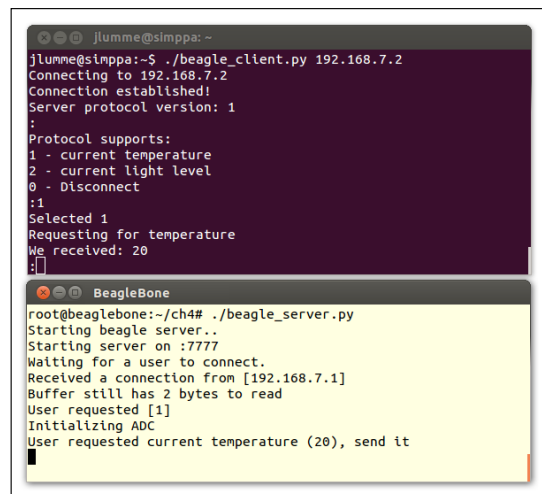

Once the message has been sent, we start expecting a reply from the server. The function that reads reply will currently support only short replies (two-byte payload). We will extend it later to also support larger payloads.

```
def read_reply_from_server(socket):
    try:
        reply = socket.recv(4) # Retrieve the initial size
        msg = unpack("!HH", reply)
        if msg[0] == BP.MT_REPLY_SHORT:
            remaining_size = int(msg[1]) # This is actually fixed
            reply = socket.recv(remaining_size)
            value = unpack("!H", reply)
            print "We received: %s" % value
        else:
            print "Received unexpected reply -> ignore"
    except Exception, err:
        print "Problem when reading from socket"
        print err
```

We're left with the code of the help message for the client, which just basically prints our definitions from `beagle_protocol.py`. The code is as follows:

```
def print_options():
    print "Protocol supports:"
    print "%d - current temperature" % BP.ACTION_READ_TEMPERATURE
    print "%d - current light level" % BP.ACTION_READ_LIGHT_LEVEL
    print "%d - Disconnect" % BP.MT_DISCONNECT
```

Now you should be ready to run the client code and connect to the server. The output should be as shown in the following screenshot:



The screenshot shows two terminal windows. The top window, titled 'jlumme@simppa: ~', shows the execution of `./beagle_client.py 192.168.7.2`. It displays the connection process, the server protocol version (1), and the protocol supports list: 1 - current temperature, 2 - current light level, 0 - Disconnect. The user selects 1, requests the temperature, and receives the value 20. The bottom window, titled 'BeagleBone', shows the execution of `./beagle_server.py`. It displays the server starting on port 7777, waiting for a connection, receiving a connection from [192.168.7.1], and responding to the user request for the current temperature (20).

Our client and server now happily work together. We're sure you already have many ideas how to extend that from here. For example, you could have a go at adding a functionality to turn the lights on and off remotely and read the current state of the lights.

Transistors

You might have already wondered what can we do if we would like to control devices that require higher voltages or current levels than those that are available from our headers.

For example, you might want to engage a 12 V motor that moves a rig holding your SLR camera for a time lapse movie, or you might want to use a high-power LED that requires much more current than the usual 4-6 mA available in our GPIO pins. And there are plenty of other use cases you might want to consider.

For this kind of application, transistors are ideal. Without going too deep into the theory on how they operate, you can think that transistors can be used as switches of a certain kind that you can operate with a low-power input signal (they are also used as amplifiers, but this is not really in the scope of this book).

Transistors mostly fall into two main types: **field effect transistors (FETs)** and **bipolar junction transistors (BJTs)**. We will focus on BJTs and their main categories: NPN and PNP. Both of these are a type of BJTs. They both share the same basic structure of having three legs. The legs are as follows:

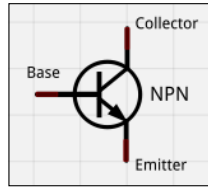
- The **base** is the lead that activates the transistor
- The **collector** is the positive lead
- The **emitter** is the negative lead

Both of them also operate in the same fashion. By applying voltage to the base pin, you can control the flow of electricity through the transistor.

The difference is that while you increase the current more and more to the base pin of the NPN transistor, it will start conducting more and more. PNP, on the other hand, works exactly in the opposite way; the more voltage you apply to the base pin, the less it conducts.

In real life, NPN transistors are much more popular, partly thanks to their electrical characteristics. We will also focus on NPN transistors in this section.

The schematic picture for an NPN transistor looks as shown in the following figure:



Using NPN transistors, you can easily control higher current loads to external hardware as compared to, for example, the maximum 4 mA (6 mA on some pins) supplied by GPIO pins on our Beagle.



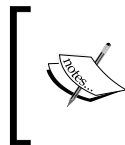
You have to be careful, however, that your transistor can conduct the current you are planning to pass through it. This value should be checked from the schematic of your transistor.

We mentioned earlier that by increasing the current and voltage to the base of an NPN transistor, it starts conducting more and more current across its emitter and collector. There is a ratio, called **DC Current gain (hFE)**, that describes this current conductivity ratio. This ratio varies depending on the input current and voltage at the base of the transistor.

Now let's take a look at an example of how one could use an NPN transistor to conduct roughly 20 times higher current in a circuit than before. This will be a somewhat theoretical example, as it's not very practical to run such high currents using normal "household batteries". But it will demonstrate the use of NPN transistors in the context that we are already familiar with.

We will re-use our old code from `led_control.py` created in the *Echo client* section in *Chapter 3, Creating the Client and Server*, to add a new control target for our LED library.

This LED will be powerful enough to be used in a dark room, instead of being just an indicator light. We will be using a Luxeon Rebel high-power LED from Phillips that we will drive with ~100 mA of current from a pair of 9 V external batteries connected in series.



The LXM3-PW61 LED has the following characteristics:

- Typical forward voltage of 3 V at around 4000 K
- Maximum forward current of 700 mA

Now we will have to do some more calculations for currents and resistances that we will design for this circuit.

Since we are using an 18 V battery and we want to drive our LED at 100 mA, we can calculate a suitable resistor for the LED from the following formula:

$$R2 = (18V - 3V) / 0.1A$$

$$R1 = 150 \text{ Ohm}$$

As we need to keep in mind the maximum current limits our GPIO can stand, we will choose a 1 K resistor for the GPIO line, which will be connected to the base of our transistor. Our load on the GPIO will thus be:

$$I_{\text{gpio}} = (3.3V - 0.6V) / 1000$$

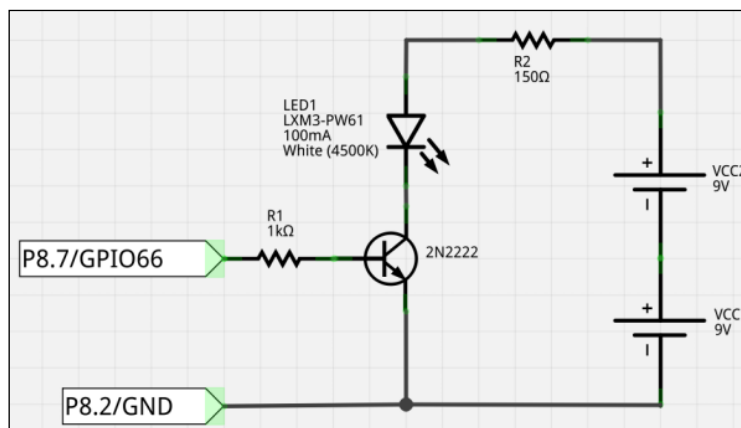
$$I_{\text{gpio}} = 2.7 \text{ mA}$$

💡 Typically the "on state" voltages of BJTs made from silicon are around 0.6 to 0.7 Volts.

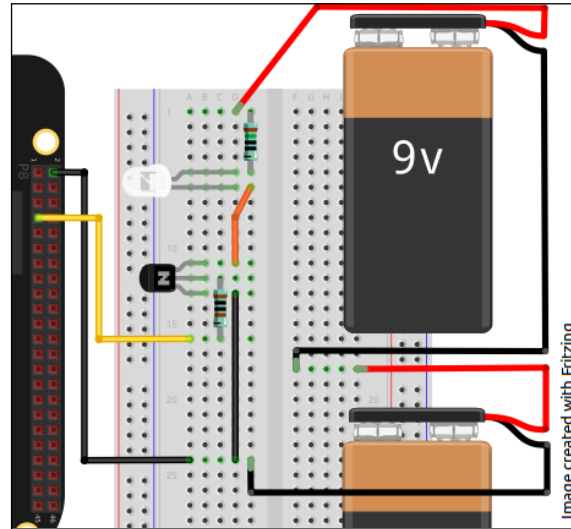
2.7 mA is nicely below the 4 mA our GPIO line can handle. We will be using a fairly common **2N2222** transistor, which has a gain (**hFE**) of 50, when the current is at least 1 mA. We can get maximum current at these levels from the following equation:

$$I_{\text{npn}} = 2.7 \text{ mA} * 50 = 135 \text{ mA}$$

The math looks good, so the whole schematic will thus look as it appears in the following figure:



The setup of wiring for the schematic is as follows:



Now there is one more caveat you have to consider here before turning on the circuit. The R2 that is limiting the current to the LED has to be capable of handling high power. In this circuit it will be driven with the power of:

$$\begin{aligned} P &= I * U \\ P &= 0.1A * (18V - 3V) \\ P &= 1.5W \end{aligned}$$

Once you're done adding the new code to control GPIO66 to `led_control.py`, you should be able to turn on the new extremely bright LED. It's really, really bright, isn't it?

The efficiency of the circuit can also be questioned, and whether the NPN approach is the best in this case. But it's a fairly simple circuit that should give you an understanding of how NPN transistors can be used to drive much higher voltages and currents that would not normally touch our target board.

Summary

We browsed through many subjects in this chapter; we extended our Beagle in many directions all at once, so it's become much more capable after this chapter, hasn't it? You now have a good understanding of how one can efficiently use network programming to transfer arbitrary data between different machines, and how to create custom transfer protocols. For the first time, we also added some new exciting environmental sensors that can gather real-time data so that our Beagle can, for the first time, sense its environment. And all of this data is now reachable to you remotely over the TCP/IP protocol. Exciting! Isn't it?

Next, we will take the last limitations and chains off the server, so you will be completely free to implement any kind of data frameworks you like, and of course, we will again add some, yet even more exciting, hardware components to our repertory.

5

Implementing Periodic Tasks

Our server is now capable of serving our clients with some environmental data, but it has no memory, and there is no way to alter its behavior without rewriting code. We will extend our server capabilities further in this chapter, and it will finally become a fully standalone server that will live its own life without the need for manual operation. We will also enhance our server-client interface so that the server can be operated and runtime configuration can be changed remotely with our client.

Of course, as always, we will also introduce new exciting hardware. In short, we will cover the following topics:

- Adding a save/load framework for arbitrary data to our server
- Creating configurable periodic tasks to our server
- Adding interfaces to access data on the server to our client
- Implementing remote reconfigurability to our server
- New hardware: infrared motion sensor
- Talk about "cape" extensions and integrate one to our Beagle
- Adding support for the camera interface to our server

Implementing a save/load framework

Now that our target will be having a server constantly running on our board, we will need some kind of configuration file for it so that the server knows how it is supposed to operate. For this purpose, we will create a file called `server.conf` (or more exactly, we will define a variable that will hold the name of this file) that will always hold the current runtime configuration of our server.

First, we should define a structure for it. A very simple one can be a configuration like `keyword=value`. It's simple to read, and also parsing it with code is easy.

Then, we should think about what type of configuration data we need in our server. This will, of course, depend on the features of the server. What kind of features do we currently have in our code? That's right, we have two functions, temperature and light sensor measurement. Let's make the delay between readings a configurable value. Create a file called `server.conf` and enter the following lines in it:

```
delay_between_temperature_measurements=30
delay_between_lightsensor_measurements=30
```

For now, that is enough.



When you start deploying your server, you might want to consider where you want to place this kind of configuration file, and of course the code itself as well.

One location we recommend is under `/opt/your_app_name/`. This is quite a common location to store these applications and the data that is Linux distribution independent.

First, let's update our protocol class `beagle_protocol.py` so that we have the necessary new messages ready:

```
ACTION_READ_CONFIG_SETTINGS          = 3
ACTION_CONFIG_CHANGE                  = 4

CONFIG_ITEM_TEMPERATURE_READ_DELAY   =
    "delay_between_temperature_measurements"
CONFIG_ITEM_LIGHTSENSOR_READ_DELAY   =
    "delay_between_lightsensor_measurements"

REPLY_CONFIGURATION_CHANGE_OK         = 41
REPLY_CONFIGURATION_CHANGE_FAILED     = 42

PROTOCOL_VERSION = 2
SUPPORTED_ACTIONS =
    (ACTION_READ_TEMPERATURE, ACTION_READ_LIGHT_LEVEL, ACTION_READ_
     CONFIG_SETTINGS, ACTION_CONFIG_CHANGE)
```

We defined two new actions for our protocol, and we also defined two new reply messages from the server.

Let's also update our protocol version string and supported actions list. We will also add names for our configurable settings.

Now that we have our configuration file ready and protocol updated, we should extend our server to take advantage of these new features. Open up our `beagle_server.py` class and add the following new variables to our server:

```
config_delay_temp_meas=0
config_delay_light_meas=0

supported_configuration_items = [
    BP.CONFIG_ITEM_TEMPERATURE_READ_DELAY,
    BP.CONFIG_ITEM_LIGHTSENSOR_READ_DELAY,]

filename_server_conf="server.conf"
```

And then, let's add a new method to load those values from our config file:

```
def reload_configuration():
    print "Reloading configuration"
    try:
        cf = open(filename_server_conf, "r").read().splitlines()
        for line in cf:
            if line.startswith("delay_between_temperature_measurements"):
                config_delay_temp_meas = int(line[line.index("=") + 1:])
                print "Configured: Temperature measurement every %d
                    minutes" % config_delay_temp_meas
            if line.startswith("delay_between_lightsensor_measurements"):
                config_delay_light_meas = int(line[line.index("=") + 1:])
                print "Configured: Light measurement every %d minutes" %
                    config_delay_light_meas

    except Exception, err:
        print "FATAL: Failed to read config file"
        print err
        sys.exit(2)
```

As you can see, no fancy code here. We use the function `open` to read the input file line by line, and after that we set our internal parameters. We should always call this function when the server is started (and also when a client updates any of these values).

Next, let's implement a function that will allow us to update a parameter in this configuration file:

```
def update_configuration(param_name, param_value):
    print "Configuration change requested [%s] to [%s]" %
        (param_name, str(param_value))
    if param_name in supported_configuration_items:
        curr_file = open(filename_server_conf, "r").read().splitlines()
        new_file = open("conf.tmp", "w")

        for line in curr_file:
            if line.startswith(param_name):
                print "we should modify [%s] to %s" % (line,
                    str(param_value))
                new_file.write(param_name + "=" + str(param_value) + "\n")
            else:
                new_file.write(line + "\n")
        new_file.close()
        os.rename("conf.tmp", filename_server_conf)
        return BP.REPLY_CONFIGURATION_CHANGE_OK

    else:
        print "Illegal configuration option specified"
        return BP.REPLY_CONFIGURATION_CHANGE_FAILED
```

As input, the function takes parameter to change and the new value. It will first check if the requested parameter is in the list of configurable parameters, and if it is, it will open the server configuration file (the name is defined in variable `filename_server_conf`) for reading and look for the necessary line that needs changing. It writes the old config plus the modification to a temporary file, `conf.tmp`. Lastly, that will be renamed as the new server configuration once the file has been fully processed.

Now that we have functions to reload and modify the runtime configuration, we need one more helper function. We need a method to easily read and process the configuration to a presentable (actually, transferable) format. Add one more function to our server:

```
def get_current_configuration():
    reply = ""
    try:
        cf = open(filename_server_conf, "r").read().splitlines()
        for line in cf:
            reply = reply + line + "|"
```

```

    return reply
except Exception, err:
    print "FATAL: Failed to read config file"
    return ""

```

Now that our backend code is ready, let's first jump over to the client side and write the code that will use these new functionalities.

Retrieving and changing permanent settings

As you must have already guessed, we will need to add some new message-handling sequences on both the client and the server. Since our client is always the transaction initiator, we start there.

The client side

As you remember, our client takes input from the user, and available input can be listed via the `print_help()` function. So we update that first:

```

def print_options():
    print "Protocol supports:"
    print "%d - current temperature" % BP.ACTION_READ_TEMPERATURE
    print "%d - current light level" % BP.ACTION_READ_LIGHT_LEVEL
    print "%d - Read current config from server" %
        BP.ACTION_READ_CONFIG_SETTINGS
    print "%d - Change a config parameter on the server" %
        BP.ACTION_CONFIG_CHANGE
    print "%d - Disconnect" % BP.MT_DISCONNECT

```

The first function to get called after user input is received and validated, is `send_message()`. This function will need two new supported message types:

```

def send_message(message_type, socket):
    ...
    elif message_type == BP.MT_DISCONNECT:
        packet = pack("!HH", BP.MT_DISCONNECT, 0)
        socket.sendall(packet)
    elif message_type == BP.ACTION_READ_CONFIG_SETTINGS:
        packet = pack("!HHH", BP.MT_REQUEST,
            BP.PAYLOAD_FIXED_SINGLE_VALUE, BP.ACTION_READ_CONFIG_SETTINGS)
        socket.sendall(packet)

```

```
elif message_type == BP.ACTION_CONFIG_CHANGE:
    pname = raw_input("Parameter name to change:")
    pval = raw_input("New value:")

    message = pname + "|" + pval
    mlen = len(message)
    pack_string = "!HH%ds" % mlen
    packet = pack(pack_string, BP.MT_REQUEST_PL_CONFIG_CHANGE, mlen,
                  message)
    socket.sendall(packet)
```

The first new message type requests our server to send the current configuration settings to the client. The second one is used to request a configuration change. We will keep this functionality simple, and we require manual input of the parameter name and value.

The other part of the message transaction is always the function `read_reply_from_server()`; previously, we only handled the `MT_REPLY_SHORT` type, but now we have two new reply types that we process. The first one will be `MT_REPLY_CURRENT_CONFIG`:

```
def read_reply_from_server(socket):
    ...
    elif msg[0] == BP.MT_REPLY_PL_CURRENT_CONFIG:
        print "Receiving a payload with configuration settings"
        remaining_size = int(msg[1])

        reply = socket.recv(remaining_size)
        unpack_string = "!%ds" % remaining_size
        value = unpack(unpack_string, reply)

        #Check how many config entries to parse
        item_count = value[0].count("|")
        idx = 0

        print "Currently configured:"
        for i in range(0,item_count):
            print "%s" % value[0][idx:value[0].index("|", idx +1)]
            idx = value[0].index("|", idx +1) +1
```

Once we have identified that the incoming message type is the current configuration listing, we read the reply to a string, and then we do some parsing on it to identify individual parts (the server separates each config item with a `|` character) and print them on the screen.

The second new incoming message type on the client side is reading a reply to our configuration change request:

```
elif msg[0] == BP.MT_REPLY_CONFIG_CHANGE_RESULT:
    print "Received config change result"
    # Length of this message type is always same
    remaining_size = int(msg[1]) # This is actually fixed
    reply = socket.recv(remaining_size)
    value = unpack("!H", reply)

    if value[0] == BP.REPLY_CONFIGURATION_CHANGE_OK:
        print "Configuration successfully updated"
    else:
        print "Configuration change was rejected"
```

As seen before, this is the same type of message handling, and we just read the server reply and showed it to the user. These are all the required changes on the client side.

The server side

We have already added all the new activity functions to our server, but we still need to add the handling code for sending and receiving the new messages. The client we just updated can send two new message types, so let's look at handling functionality for them. In `handle_client_request()`:

```
if header[0] == BP.MT_REQUEST:
    ...
    elif request == BP.ACTION_READ_CONFIG_SETTINGS:
        print "User has requested current runtime settings"
        reply = get_current_configuration()
        length = len(reply)

        data_packet =
            create_data_packet(BP.MT_REPLY_PL_CURRENT_CONFIG, reply,
                               length)
        if not data_packet == None:
            cs.sendall(data_packet)
```

The message requesting the current configuration is a regular request without payload (extra data besides the initial two protocol headers), and thus it is handled inside the `BP.MT_REQUEST` branch. We will talk about creating the appropriate reply for this message in a moment. When a user requests for a configuration change, it requires a bit more processing (the configurable item the users wants to change is specified in the payload).

This message is not handled inside `BP.MT_REQUEST`.

```
elif header[0] == BP.MT_REQUEST_PL_CONFIG_CHANGE:
    print "User requested configuration change"
    remaining_size = int(header[1])
    reply = cs.recv(remaining_size)

    unpack_string = "!%ds" % remaining_size
    incoming_string = unpack(unpack_string, reply)

    #Parse the read data from client
    pname = incoming_string[0][:incoming_string[0].index("|")]
    pval = incoming_string[0][incoming_string[0].index("|")+1:]

    #Try to change the config:
    result = update_configuration(pname,pval)

    data_packet =
        create_data_packet(BP.MT_REPLY_CONFIG_CHANGE_RESULT, result,
            BP.PAYLOAD_FIXED_SINGLE_VALUE )
    if not data_packet == None:
        cs.sendall(data_packet)
```

As you can see, only payload processing requires a bit more code around it. The client has supplied the parameter it wishes to change and the new value separated by `|`. The server doesn't do any input verification here; it is done in the `update_configuration()` function we described earlier. It immediately sends the result of that function call back to the client.

We added the handling code for two new incoming messages, and now we also need support for sending replies to them. Add the following functionality to the `create_data_packet()` function:

```
def create_data_packet(msg_type,data, data_length)
...
elif msg_type == BP.MT_REPLY_SHORT:
    packet = pack("!HHH", BP.MT_REPLY_SHORT, data_length, data)

elif msg_type == BP.MT_REPLY_PL_CURRENT_CONFIG:
    pack_string = "!HH%ds" % data_length
    packet = pack(pack_string, BP.MT_REPLY_PL_CURRENT_CONFIG,
        data_length, data)
```

```
elif msg_type == BP.MT_REPLY_CONFIG_CHANGE_RESULT:
    packet = pack("!HHH", BP.MT_REPLY_CONFIG_CHANGE_RESULT,
                   data_length, data)
```

We only have to take care of the proper packing of the payload message when sending the current configuration information to the client.

Now the functionality is complete in both the client and the server. The client can now read the current configuration on the server and change it if required. As you surely have noticed, this is currently just a basic functionality to read and change the configuration file contents over the network. Next, we will implement some actual functionality around these parameters.

Periodic tasks on the server

Currently, our server has the capability to read temperature and light levels on request, but this data is not collected periodically, nor saved anywhere. In this section, we will implement both functionalities with the help of our new configuration options.

First we will need some new imports in our server:

```
import sys, os, time
from time import gmtime, strftime #For formatted time
from multiprocessing import Process #For multiprocessing
```

Then we will need process handles to our child processes so that the parent process can terminate and restart them when necessary:

```
# Our process handles
periodic_temp_measurement = Process()
periodic_light_measurement = Process()
```

Currently, we have passed just a placeholder for the `Process` class; they will be properly initialized later. We also need to define two new variables holding the filenames for the files that will hold the measurement data:

```
filename_measurements_light=".server_data_lightlevel.txt"
filename_measurements_temperature=".server_data_temperature.txt"
```


Next, we will define the function that will collect a data sample and save it to a logfile with the timestamp of the acquired sample. First is the function for temperature collection:

```
def periodic_temperature_collector(frequency):

    print "Temperature collector initialization"
    print "At %s, frequency every %d minute(s)" % (strftime("%Y-%m-%d %H:%M:%S", gmtime()), frequency)

    outputfile = open (filename_measurements_temperature, "a")

    # Parent takes care of the life cycle of this process
    while True:
        #Read temperature and current time
        val = get_temperature()
        timenow = strftime("%Y-%m-%d %H:%M:%S", gmtime())

        outputfile.write(timenow + "|" + str(val) + "\n")
        outputfile.flush() #Force flush

        # Sleeping the defined amount
        time.sleep(frequency * 60)
```

The call to `strftime()` with the `gmtime()` function's parameter is used to make a timestamp in a desired format. Also, note that we have opened our output file in **append** mode (the file will be opened as it is, and writing will continue from the end of the file) so as not to clear the previous entries. Also define a similar function, called `periodic_lightlevel_collector()`, for light-level measurement and make it save a similar logfile with a name from the variable `filename_measurements_light`.

Next, we will need some mechanism to configure the threads and start them. That's right, we already have a perfect place for them in our `reload_configuration()` function! In case the child processes are already running, we need to terminate them first:

```
def reload_configuration():

    global periodic_temp_measurement
    global periodic_light_measurement

    print "Reloading configuration"

    #First terminate the previous threads
    if periodic_temp_measurement.is_alive():
```

```

periodic_temp_measurement.terminate()
if periodic_light_measurement.is_alive():
    periodic_light_measurement.terminate()

```

Then, after the configuration has been read from the config file, we can start new instances of our processes:

```

print "Restarting periodic tasks"
# Initialize new Processes with fresh data
periodic_temp_measurement =
    Process(target=periodic_temperature_collector,
            args=(config_delay_temp_meas,))

periodic_light_measurement =
    Process(target=periodic_lightlevel_collector,
            args=(config_delay_light_meas,))

#Start our periodic tasks
periodic_temp_measurement.start()
periodic_light_measurement.start()

```

Here you can see how a new process is created. Basically, you make a call to `Process` and pass the starting function for the new process and give the possible input parameters for that function. The call to the `Process.start()` function starts the execution.



When the server is running, you can check the measurement logfiles in the same directory your server code is in. Since the first character of the logfiles is a dot (`.`), the logfiles are not visible in normal listing. You need to use the command `ls -la` to see them. To follow the writing to those files in real time, use `tail -f filename`.

Now we have our periodic tasks running, and they are automatically restarted with new values if a client changes the server configuration.

We have been talking a lot about software in this chapter so far, so we're sure you're ready to get your hands dirty with some hardware!

Movement-detection alarm system

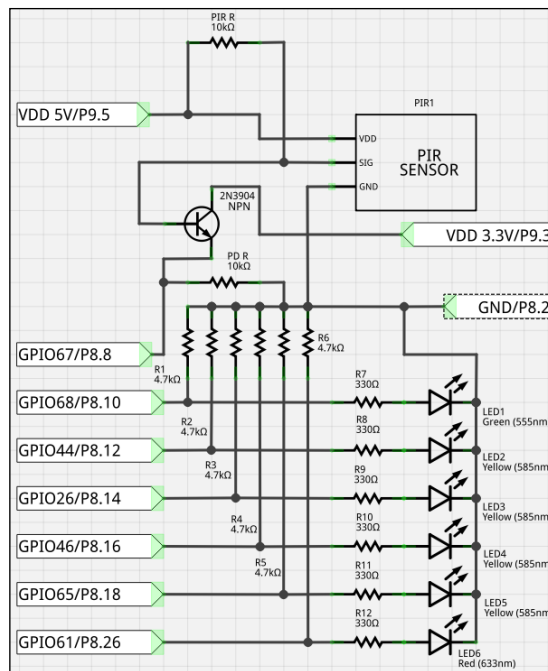
Throughout the previous chapters, we have learned a major part of the basic components that most electronic systems are created from, and through examples you have learned to master many of these. Now we feel it's time we took our lessons to practice, and create something cool and fun! This example will be like it's straight out of the casino robbery movie, *Oceans Fourteen*. Okay, okay, maybe we won't have lasers and smoke and stuff, but we will create an alarm system that will monitor its environment for any kind of movement, and raise an alarm when it realizes someone is trying to enter the room.

In this example, we will be using a movement-detection sensor that operates on infrared light, so it will "see" in the dark too. Specifically, we will use a **passive infrared (PIR)** sensor. It's a sensor that monitors its environment for changes in infrared light radiation patterns. You have most likely seen this type of sensor in real life; maybe, for example, your garage has one? They are usually used to turn on the lights in a dark room when movement is detected.

Our hardware part will be a quite common PIR sensor, which can be found in many places. You can order one, for example, from *SparkFun*, with part number **SEN-08630**. If you search through Amazon, or your local electronic parts retailer, we're sure you'll find one nearby.

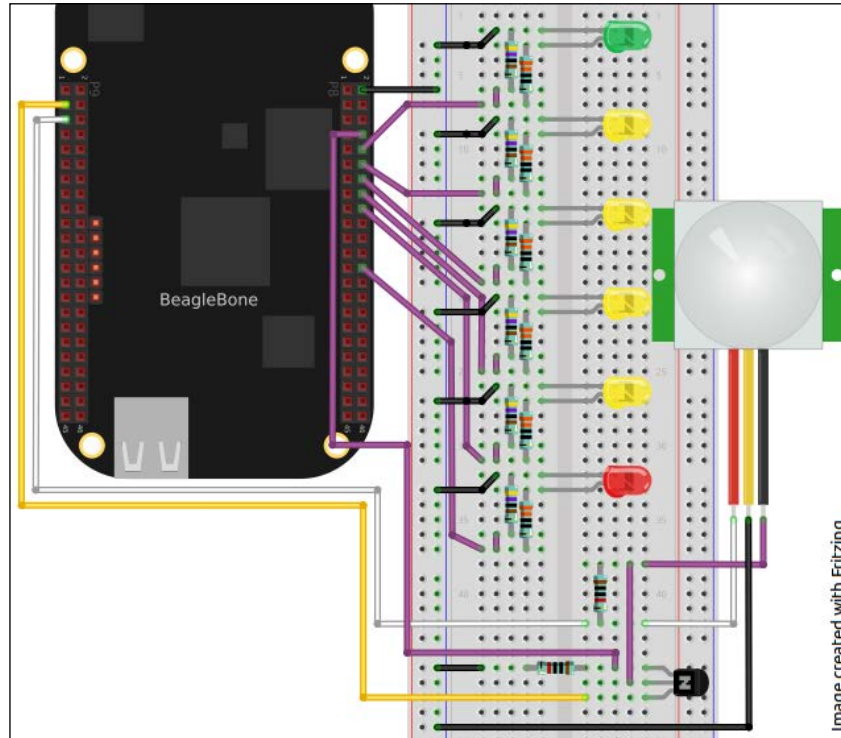
So let's start designing our electrical diagram for this example. The PIR sensor we are using works with 5 volts, so you need to have an input of 5V; the 5V rail will not have any voltage if the board is powered using the USB, and as you remember we cannot read voltages that are that high in our GPIO input pins; so we will be using a transistor to keep the high voltages out of our GPIO lines. We also connect a series of LEDs to our diagram, which will represent different stages of alertness. This way, we will be able to configure the sensor sensitivity to our liking.

First, the schematic for our circuit is as follows:




At first, the schematic might look a bit confusing, but actually there isn't anything too complicated in it. LEDs 1 to 6 are connected in the same fashion as in the example `ext_led_fun` in *Chapter 2, Input and Output*. The only input GPIO (P8.8) is fed from a 3.3V line through an NPN transistor, which is switched by the PIR sensor's signal pin. There is a 10K pull-down resistor (**PD R**) connected from the emitter pin of the transistor to the ground, so that our input GPIO is never floating, and the 10K resistor between the signal line and 5V works as a pull-up resistor. For us, 10K worked best, but you can give 4.7K a try if you have issues.

The wiring diagram of the previous schematic is as follows:



You can notice here that the PIR sensor wiring is actually different from the schematic; this is how the wiring is in the SEN-08630 sensor. The colors of the wires vary, so when you look at the sensor from the top, you will find that the wire on the left-hand side is +5 and the right-hand side one is the alarm line.

[ Keep in mind that when the PIR sensor gets power, it will take some time to stabilize. Our sensor sometimes took as long as 10 seconds to start giving reliable results. Googling around revealed that we weren't the only ones experiencing this.]

Now that we have our hardware wired up, let's take a look at some sample code that can operate this. Let's create a library class to control our movement sensor, called `PIR_ctrl.py`.

First we will initialize our GPIOs:

```
#!/usr/bin/python
import Adafruit_BBIO.GPIO as GPIO
import time
from time import gmtime, strftime #For formatted time

ALARM_LEVEL = 6

def init_gpio():
    #Initialize GPIOs
    GPIO.setup("P8_8", GPIO.IN)

    GPIO.setup("P8_10", GPIO.OUT)
    GPIO.setup("P8_12", GPIO.OUT)
    GPIO.setup("P8_14", GPIO.OUT)
    GPIO.setup("P8_16", GPIO.OUT)
    GPIO.setup("P8_18", GPIO.OUT)
    GPIO.setup("P8_26", GPIO.OUT)
```

Then let's create a simple function that will light up the proper number of LEDs depending on the alarm level:

```
def indicate_alert(alarm_level):
    if alarm_level == 0:
        GPIO.output("P8_10", GPIO.LOW)
        GPIO.output("P8_12", GPIO.LOW)
        GPIO.output("P8_14", GPIO.LOW)
        GPIO.output("P8_16", GPIO.LOW)
        GPIO.output("P8_18", GPIO.LOW)
        GPIO.output("P8_26", GPIO.LOW)

    if alarm_level >= 1:
        GPIO.output("P8_10", GPIO.HIGH)
    if alarm_level >= 2:
        GPIO.output("P8_12", GPIO.HIGH)
    if alarm_level >= 3:
        GPIO.output("P8_14", GPIO.HIGH)
    if alarm_level >= 4:
        GPIO.output("P8_16", GPIO.HIGH)
    if alarm_level >= 5:
        GPIO.output("P8_18", GPIO.HIGH)
    if alarm_level >= 6:
        GPIO.output("P8_26", GPIO.HIGH)
```

And then create a function `start_monitoring()`. This function will be called by the main function to start the operation of this class. The function will look like this:

```
def start_monitoring():
    alarm_buffer = 0
    init_gpio()
    while True:
        new_state = GPIO.input("P8_8")
        if new_state == 0:
            alarm_buffer += 1
            indicate_alert(alarm_buffer)
        if alarm_buffer > ALARM_TRIGGER:
            alarm_buffer = 0
            alarm_status = "@%s ALARM" % time.strftime("%Y-%m-%d
                           %H:%M:%S")
            print alarm_status
            for i in range(0,5): #Flash red LED
                GPIO.output("P8_26", GPIO.HIGH)
                time.sleep(0.3)
                GPIO.output("P8_26", GPIO.LOW)
                time.sleep(0.3)
        else:
            alarm_buffer = 0
            indicate_alert(alarm_buffer)
        time.sleep(0.1) #Sleep
```

As you can see, we first initialize the GPIOs, and then start monitoring the line P8_8 to go low. When this happens, we increment the "alarm level". We chose to check the sensor every 100ms, and if the alarm is triggered for six consecutive measurements, the alarm will be set off.



Some of the PIR sensors have a minimum time they stay triggered, so you might need to adjust your sleep periods, and what the suitable trigger level for your sensor will be. Play around with different settings until you feel that the sensor will reliably notice someone entering the room.

Now give it a go, and lay a bet with your sister to sneak into your room and take something from your table without our Beagle noticing her.

You have now created your very own alarm system with the board, so how about trying to extend our server with this new functionality? Maybe our server can record all the alarms that are triggered throughout the day, and create logfiles from those security breaches? And what about extending the functionality even further, wherein you can remotely check these alarm logs?

The security system we have created here currently only uses visual data, but actually you can also consider extending the system using magnetic switches (called **reed switches**) to monitor, for example, the safe!



For this functionality, some extensions will be needed for `beagle_protocol`, such as a new periodic task to read the security state and create logs and, of course, extensions to our client and server code.

Hardware extensions

We have so far been creating all our examples ourselves on breadboards, but there are many readymade extensions available. These extensions are called **cap**es in the Beagle world, and you can think of them like plug-and-play hardware modules. They are much more complicated and usually contain autonomous computing chips on them that handle the necessary data processing for that particular cape (for example, calculating GPS coordinates). They are meant to be sandwiched on top of the header pins. You can place several of them on top of each other as long as you configure the SW1 jumper on each board correctly (you can find more information about this in the reference manual). Of course, you have to make sure two capes don't block each other.

You can find many different types of capes around, starting from simple breadboards that are placed on top of your Beagle to very sophisticated HD quality cameras or LCD capes.

As you probably guessed, some of these can get fairly expensive (for example, the LCD cape for our Beagle costs three times as much as our Beagle!), but these can be extremely useful when prototyping your own design and so on.



Please keep in mind that because the BeagleBone Black and White don't share the same header and GPIO configuration, you have to be sure that the cape you are ordering will work with your board.

Naturally, each board needs some control software so that the Linux kernel can operate with the cape properly, and this is provided by each cape's manufacturer. Let's now take a look at one example cape.

BeagleBone HD camera cape

In this section we will introduce the HD camera cape from Radiumboards (don't confuse it with the older camera cape originally meant for the white board), which integrates the camera to our Beagle, and shows you how to use it. The cape we are talking about in this section can be found at http://www.radiumboards.com/HD_Camera_Cape_for_BeagleBone_Black.php. We do realize that the camera cape costs as much as the Beagle itself, so this might not be a relevant section for every reader, but it does have advantages for someone integrating a camera to his design (such as extremely low power requirements and tiny size). In this section we will also enable general support for using a normal USB webcam for those readers who are not interested in ordering another piece of hardware just for hacking around. In the next section we will talk about the necessary preparation to use the camera cape.



There are some downsides to using the camera cape instead of a USB solution. As we mentioned before, the cape uses GPMC to talk with the main board processor, and thus it reserves quite a lot of pins to its use—28 in fact. This of course means that there are less GPIO pins available, and you need to consider this during the design phase. This cape actually reserves a lot of pins in the P8 header, and none of our examples from this chapter would work without pin reassignment.

Changing the boot media

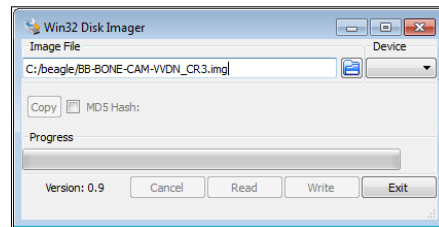
The first thing to realize is that we cannot use eMMC to boot our Beagle, since the cape uses the **General Purpose Memory Controller (GPMC)** interface to transfer data to the main board. The CD-ROM provided with the camera contains a pre-prepared Angstrom image that has already been applied with the necessary patches so that the camera will work.



The CD also contains the necessary patches, so you can integrate them to another distribution if you have chosen to run some other flavor or Linux on your board, or have already taken some other pre-prepared software image that came with another cape. Compiling and integrating the driver patch is an advanced topic and is out of the scope of this book. But we're sure the BeagleBoard community can help you with this through its forums.

So we will need at least a 4GB microSD card for the Angstrom image (its contents will be erased, so please take care of that first). The compressed BB-BONE-CAM-VVDN_CR3.img.xz image is located on the CD in the folder /bin/image.

Windows users will also need a program that can create a bootable microSD card from the IMG file. For extracting the .xz files, you can use 7-Zip. If you don't already have it, you can download it from <http://www.7-zip.org/>. Once you have extracted the image file, download the image creator program from <https://wiki.ubuntu.com/Win32DiskImager>. It looks as shown in the following screenshot:



Just choose the extracted file (**Image File**) as the source to be used and the correct drive as the target (**Device**), and you can write the image to the memory card.

In Linux, users can use the pre-installed console program called `xz` for extracting the IMG file, and on Mac OS you can do this with **Unarchiver** (you should be able to find it in the Mac App store for free). On both the systems you can use the program called `dd` (it can be used for byte-by-byte copying irrespective of the filesystem used) for writing to the memory card. But before using it, you will need to know the appropriate drive for the memory card. When you plug your microSD card into the card reader, your system should recognize the card, and you should be able to see the appropriate drive by using the command `dmesg` (it prints the log messages from the kernel in chronological order). If you run the command right after insertion, you will see something similar to the following screenshot (only the last few messages are shown here):

```
jlumme@simppa: ~
I: 0
[19378.826168] sd 8:0:0:0: Attached scsi generic sg1 type 0
[19379.457914] sd 8:0:0:0: [sdb] 31326208 512-byte logical blocks: (16.0 GB/14.9 GiB)
[19379.458777] sd 8:0:0:0: [sdb] Write Protect is off
[19379.458782] sd 8:0:0:0: [sdb] Mode Sense: 03 00 00 00
[19379.459591] sd 8:0:0:0: [sdb] No Caching mode page found
[19379.459596] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[19379.462317] sd 8:0:0:0: [sdb] No Caching mode page found
[19379.462321] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[19379.463450] sdb: sdb1
[19379.465761] sd 8:0:0:0: [sdb] No Caching mode page found
[19379.465766] sd 8:0:0:0: [sdb] Assuming drive cache: write through
[19379.465770] sd 8:0:0:0: [sdb] Attached SCSI removable disk
[19383.814285] systemd-hostnamed[4057]: Warning: nss-myhostname is not installed. Changing the local hostname might make it unresolvable. Please install nss-myhostname!
jlumme@simppa:~$
```

In the previous screenshot, you see that on this particular PC the memory card is recognized as `sdb` (the drives are usually in the form `/dev/sd*`). If the system has automatically already mounted your memory card, you can call `umount /dev/sd*1` and all the other numbers that the system might have found. The system might otherwise refuse to write the image if the filesystem has already been mounted for use.



Word of warning! You have to be absolutely sure about the drive letter you will use as a target when you run the next command. It will overwrite the contents of that drive, so if you supply your normal hard disk drive as the target, you can lose a lot of data. But, since you know the size of your memory card, you can usually identify the drive easily. Here we used a 16GB card, and normal hard disks are hundreds of gigabytes, if not more.

Also, don't forget that if you do this several times, the drive letter might change (for example, if you connect your smartphone to the PC between operations, it might take one of those positions as well), so always check.

On Mac you can see the available list of disks with `diskutil`; you should find something similar to `/dev/disk*`. Then call `diskutil unmountDisk /dev/disk*`, and you should be ready to write the image.

When you know the proper drive letter, you can go ahead and create the image with the following command (replace the star with the appropriate letter):

```
xz -cs BB-BONE-CAM-VVDN_CR3.img.xz > BB-BONE-CAM-VVDN_CR3.img
dd if=BB-BONE-CAM-VVDN_CR3.img of=/dev/sd* bs=1M
```

The `xz` command first extracts the image to another file, and then we use `DD` to write the file to `/dev/sd*` one MB at a time (block size). On Mac, you should use `/dev/disk*` instead of `/dev/sd*`.

The copying of the almost 4GB-sized file will take a while, so while you're waiting, you can turn off your Beagle (enter the command `halt` in the console), disconnect all cables from it, and put the cape on top of the Beagle. Be sure to place it the proper way, otherwise you might damage your board! The round edges should match on both boards at the bottom side (see the figure on page 8 of *RB_HD_Camera_Cape_for_BeagleBone_Black_System_Reference_Manual_A0-01*).

After the copying has finished, you can remove the microSD card from your computer, insert it into the Beagle, and connect the 5V plug for it to start booting up. Connecting the external power supply is necessary for the camera hardware to operate, so powering the board through the USB will not be sufficient.



We noticed that at least the Angstrom image that came with the revision **A1** of the cape did not enable SSH connectivity over USB. If this happens with your image as well, you will have to connect to the board via either serial cable (described in detail in *Appendix, Security, Debugging, and I2C and SPI*) or connect your board to the router, and use SSH over TCP/IP. You will need to check the IP address from the router management console with your web browser.

When the board is booting, you will see new messages that indicate that the camera board has been successfully loaded:

```
[ 0.246779] bone-capemgr bone_capemgr.8: Baseboard:
'A335BNLT,0A5C,2713BBBK7397'
[ 0.246818] bone-capemgr bone_capemgr.8: compatible-
baseboard=ti,beaglebone-black
[ 0.277118] bone-capemgr bone_capemgr.8: slot #0: No cape found
[ 0.314225] bone-capemgr bone_capemgr.8: slot #1: No cape found
[ 0.344484] bone-capemgr bone_capemgr.8: slot #2: 'BeagleBone
1.2MP CAMERA CAPE,00A0,RadiumBoards,BB-BONE-CAM-VVDN'
[ 0.374772] bone-capemgr bone_capemgr.8: slot #3: No cape found
```

Next, let's talk about the setup that we will need to enable the operation of `/dev/video0` in our Python programs so that we can start taking some pictures with our cameras.

Controlling cameras with Python

The first thing we need to verify is if our video hardware is recognized. Enter the following commands to the console (if you are using a USB webcam at this step, make sure it's connected):

```
BeagleBone
root@beaglebone:/dev# ls -la video*
crw-rw---- 1 root video 81, 0 Jan  1 00:28 video0
root@beaglebone:/dev#
```

If you do not see `video0` in the listing, it most likely means that we have kernel drivers missing in our image. To rectify this, install the missing driver from the package manager:

```
root@beaglebone:~# opkg install kernel-module-uvccvideo
```

After this, reboot the board and the `/dev/video0` device should appear.

Next, we will need to add a few more development libraries to our system so that we will be able to compile a Python library called `pygame`; this can be used to access our camera hardware. This time we will install a whole bunch of files:

```
root@beaglebone:~# opkg install libSDL-image-1.2-0 libSDL-image-1.2-dev libSDL-mixer-1.2-0 libSDL-mixer-1.2-dev libSDL-ttf-2.0-0 libSDL-ttf-2.0-dev libavformat-dev libavcodec-dev v4l-utils libv4l libSDL-1.2-dev python-distutils python-compile
```

If, at the time of your reading, the exact versions can no longer be found in the package manager, you can search for the currently supported version with the following command:

```
root@beaglebone:~# opkg list libSDL-image*
```

At the time of writing this book, there was a missing header file that prevented the successful compilation of `pygame`. The following symbolic link was also necessary (on your system, this might no longer be needed; check first):

```
root@beaglebone:~# cd /usr/include/linux
ln -s ../libv4l-videodev.h videodev.h
root@beaglebone:~#
```

Now we are ready to compile `pygame`. We will first need to retrieve the package from the Internet. Go to <http://www.pygame.org/download.shtml> and get a link to the latest source code (1.9.1 at the time of writing), and download, extract, and try to build it:

```
root@beaglebone:~# cd /tmp
root@beaglebone:/tmp # wget http://www.pygame.org/ftp/pygame-1.9.1release.zip
root@beaglebone:/tmp # unzip pygame-1.9.1release.zip
root@beaglebone:/tmp # cd pygame-1.9.1release
root@beaglebone:/tmp/pygame-1.9.1release # python setup.py build
```

The build will take several minutes, and once it has completed without any errors, you can install the library:

```
root@beaglebone:/tmp/pygame-1.9.1release # python setup.py install
```

Now we should be ready to start using our camera hardware in Python. Let's create a new library class for the camera, called `camera_ctrl.py`. Let's define a couple of new imports first, and a handle for our camera:

```
#!/usr/bin/python
import pygame
import pygame.camera
camera = None
```

We will need a function to initialize the camera:

```
def init_camera():
    global camera
    try:
        pygame.init()
        pygame.camera.init()
        # Retrieve instance of camera
        camera = pygame.camera.Camera("/dev/video0", (1280, 720))
    except Exception, err:
        camera = None
        print "Problem initializing camera"
        print err
```

You can see that we first initialize the pygame library and then we initialize the camera. After that, we retrieve the camera object from the library and we set it up to take pictures in 720p resolution from `/dev/video0`. After initialization, we are ready to define a method to take a picture, and define a simple main method if we call this class directly:

```
def take_picture(filename):
    global camera
    if not camera == None:
        try:
            camera.start()
            img = cam.get_image()
            img = cam.get_image()
            pygame.image.save(img, filename)
            camera.stop() # Close the camera
            return 0
        except Exception, err:
            print "Pygame threw exception:"
            print err
            return 1
    else:
        print "Unable to take picture, camera not initialized?"
```

```
        return 1 #Failed taking the picture
    if __name__ == "__main__":
        init_camera()
        take_picture("selftest.jpg")
```

In the `take_picture` function, we start the camera and take a picture. Actually, two pictures. This is on purpose, since we noticed that if we only take one picture, the automatic white balance algorithm on our camera did not have enough time to adjust to the environment, and the pictures turned out too dark. Go ahead, comment out one of those `get_image()` function calls, and check how your camera performs. Now if you execute this class, it will create a picture `selftest.jpg` in the same folder.

This is all that is needed to take a picture with the camera module on the cape (or from a camera connected to the USB port). Even though this was a very brief introduction to (the camera) capes, you can see that using one of these capes is not that difficult. The reference manual of the cape will give all the necessary instructions on how to use the cape, and all you have to do is plug it in correctly and set up the needed software support. If you have several capes, or you have decided to use some other Linux distribution on your Beagle, it will require some more advanced skills to set up a proper compiling environment. But if you have decided to take that road, don't worry, there are plenty of instructions around the Internet on how to do it.

Summary

In this chapter we did quite a bit of coding to extend our server and client programs, and they have become quite useful now, haven't they? Hopefully, we have shown you enough here to ensure you have a good understanding how to extend the software for your own needs. We now also have a working setup to access camera hardware on (or connected to) our board. Considering the previous movement sensor example, this must have given you plenty of ideas already.

The hardware example we went through in this chapter was actually the last complete hardware design that we will be going through in this book. We believe that since you have come this far already, you now have a good, general understanding how different types of components can be integrated to our target board. And even if you don't readily know how to add some components, we're sure you have a good basis to start studying and experimenting on your own.

In the next chapter, we will jump a bit to a different environment, and start creating an Android application that you can use on your smartphone. Yes, you guessed it—our target will be to connect to our server from outside of our home network so that you will be able to operate features on your server from virtually anywhere in the world! Just think of the possibilities!

6

Creating an Android Client

So far, we have been using a Python program to talk to another Python program. But as we mentioned in *Chapter 3, Creating the Client and Server Applications*, socket streams are a very easy way to create platform-agnostic interfaces. In this chapter we will demonstrate a concrete example of the same. Our target is to build an Android application that will connect to our server and implement the most important interfaces that the command-line Python client has, and also add a couple of new ones.

We will not go too deeply into Android programming here, and some previous experience in Java and/or Android programming will be very helpful when moving forward, though not necessary.

Setting up our Android project

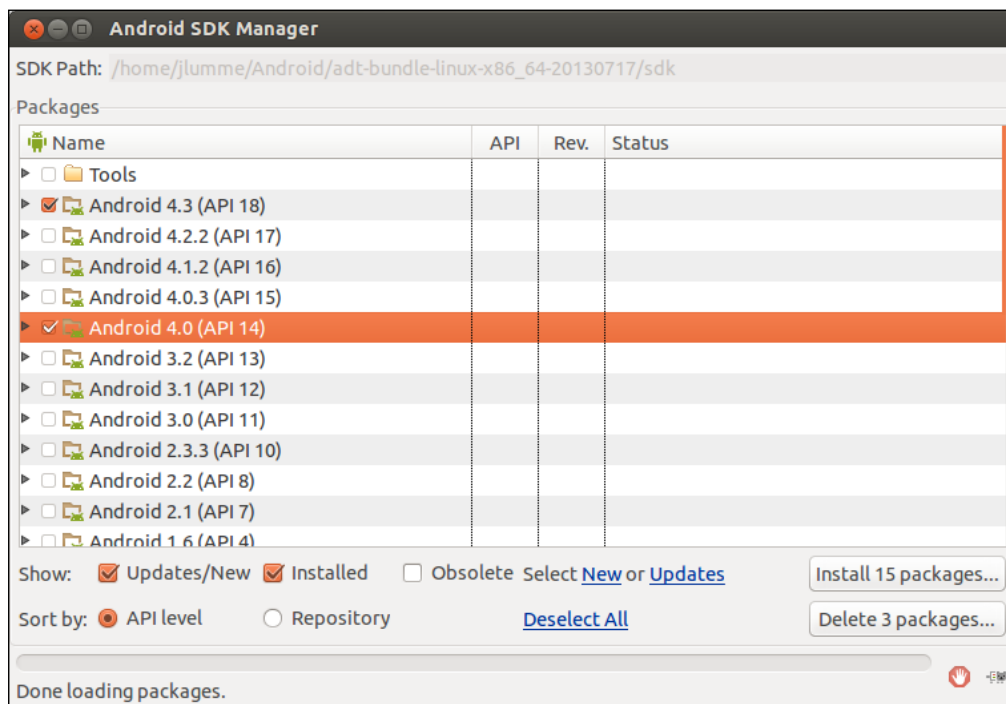
On your browser, navigate to <http://developer.android.com/sdk/index.html>; the website should present you with a download link to the appropriate **Android Development Tools (ADT)** bundle. The web page automatically provides a proper version for your working platform, and thus downloading and installing it is a fairly straightforward process.



If you already have an existing Eclipse installation on your machine, you can also save some disk space and download only the ADT part from the same page. The installation process is slightly more complicated in this case, but the instructions on the web page can help you with that.

Open the ADT bundle (or the Eclipse environment) that you've just installed. Before we start a new project, you will most likely have to download an SDK that is suitable for your particular device. In our example, we will be using Android 4.0 (code-named "Ice Cream Sandwich"). However, we will avoid using any recent APIs; so if you have an existing installation, it is not necessary to update the SDK. If you are just setting up the environment for the first time, these instructions can be useful. If you wish to execute the application on your own device, you need to check the Android version running on your phone; just navigate to the **Settings** page and confirm your Android version.

To download an SDK in Eclipse, from the menu bar (all the menu bar locations provided here are from the Linux SDK; these may vary slightly on other platforms), select **Window | Android SDK Manager**. The tool will automatically contact Google servers and display the options for different SDK downloads. By default, the newest SDK will be installed. In the screenshot that follows, you can see that we have additionally chosen **Android 4.0 (API 14)** SDK for installation:



Click on **Install xx packages...**, and in the next window choose **Accept license** and click on **Install**. Depending on your connection speed, this will take several minutes; so, go grab a cup of coffee, and let's talk about the emulator for a moment.

For the purpose of development, an emulator is invaluable. The emulator that comes with the SDK allows us to basically run the exact same application on your computer screen without having to always deploy your application to a real device. This not only saves time, but also helps with debugging. Eventually, of course, we will want to use our own device. For this purpose, depending on the platform, you will need some additional preparation so that your ADT can talk to your device without any issues.

For Windows, you will need a suitable USB driver for your device. This will most likely be found on the manufacturer's website. You can also try searching for "USB driver Android *phone model*".

In Linux, it will be enough to give appropriate udev privileges to your USB device. You can check your manufacturer's USB ID from the following list: <http://developer.android.com/tools/device.html#VendorIds>. Once you have identified your ID, create a new rule file for Android as follows:

```
/etc/udev/rules.d$ sudo nano 99-android.rules
```

In the rule file, enter the following string:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", MODE="0666", GROUP="plugdev"
```

Remember to change "04e8" with the manufacturer ID of your device.



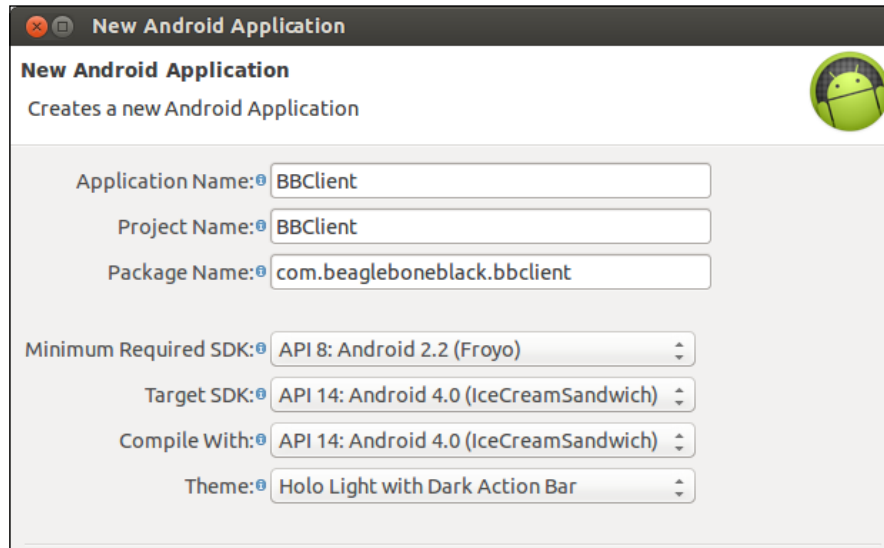
In case your manufacturer ID was not listed in the Android developer page, you can use the command `lsusb`, and try to locate your device ID. It will be the "XXXX" part of the character ID, like this:

```
jlumme@simppa:~$ lsusb
Bus 003 Device 006: ID 13fe:4100 Kingston Technology
    Company Inc.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0
    root hub
...
Bus 003 Device 007: ID 04e8:6866 Samsung Electronics
    Co., Ltd
```

On Mac OS X, there are no changes required for the SDK to be able to connect to your device.

Hopefully the SDK is already installed by now, so we can create a new project. Navigate to **File | New | Android Application Project**.

You are presented with the **New Android application** window. Let's call our application `BBClient`. If you wish to run your application on your phone, it will be important at this point to choose the correct Android version as our **Target SDK**.



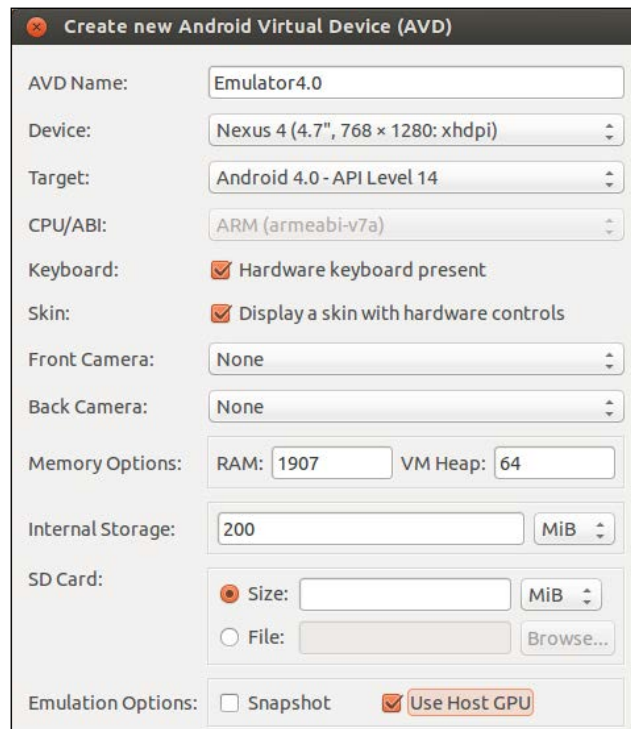
After you have named the application and chosen the appropriate SDK versions, you can use the default values (select **Next**) for the rest of the options, and click **Finish** in the last dialog. This will create a working project, and we are ready to start programming.

Before we start writing any additional code, let's verify whether we can run our project in the emulator.

Creating an emulator

We should first create a new emulator for ourselves. From the menu bar, navigate to **Window | Android Virtual Device Manager**; there should be no emulators defined yet. Click on the **Manager** button; it should open up a new window where we can create new emulator targets.

Click on **New** to create a new emulator. For the name, you can enter, for example, `Emulator4.0` and choose some device skin for it (**Device**). We chose **Nexus 4**, and then from **Target** we selected the SDK we downloaded. We enabled the **Use host GPU** checkbox, and we left the rest of the options in the default state. The selection should look like this:



Now that we have created an emulator, we can close the window. Next, we should create a new run configuration. From the menu bar, select **Run | Run configurations**. Right-click on the **Android Application** tab to create a new configuration, and name it, for example, `4.0 Emulator`. Enter `BBCClient` as our project to launch, and then go to the **Target** tab. Make sure that **Always prompt to pick a device** is selected; you should see our freshly created `Emulator 4.0` there. We can now click on **Run** and another window should pop up asking where to launch our application. If you don't currently have your device connected, only the **Launch a new Android virtual device** option should be available, check it and click on **OK**.

The emulator should start now, post which the APK is installed and the program starts.



If you are having problems starting the emulator, you might want to try disabling the **Use host GPU** option, as this is known to sometimes cause problems on some hardware configurations. You could also try a newer SDK (of course, it might make your application look different than it does on your actual device). Disabling GPU acceleration will make the refresh rate on the emulator much slower, but for this type of application, it's not so important.

You should see a window like the following screenshot:

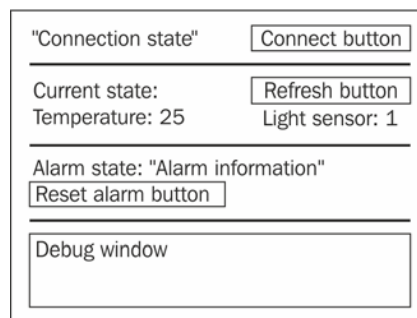


The socket client on Android

In a similar fashion as we did for our Python client application, we will create a socket client for our server in Android.

Defining the UI components

First, let's create the UI for our application. We will design a very basic-looking UI that will be functional, above all. Of course, once you have the complete application logic, nothing will stop you from rewriting the UI with fancier graphics. For the main layout, we will use `RelativeLayout`, as it is easy to understand. In the **Package Explorer** pane on the left, navigate to **res | layout**, and double-click on `activity_main.xml` so that it opens in the editor pane. Our target will be to define a layout that looks similar to this:




We will only need the `TextView`, `Button`, and `View` components in our UI. You can see that ADT already created a layout for us, where a `TextView` component has been added. We can use this view as our **Debug window**, so let's leave it there for now, and add the other components in the top part of our UI:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >
<RelativeLayout
```

```
        android:id="@+id/top_section"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/connection_state"
            android:layout_width="200dp"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:text="@string/connection_state_str" />
        <Button
            android:id="@+id/connect_button"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_toRightOf="@id/connection_state"
            android:text="@string/connect_button_str"
            android:onClick="uiEventHandler" />
    </RelativeLayout>
```

After the *main* `RelativeLayout` definition, we have defined another `RelativeLayout` definition that only contains the top two elements. We will call this sub layout as "top_section".

 Defining layouts inside one another is an easy way to compartmentalize the UI elements; you can then mix and match layouts in different parts of the screen.

To force some rendering consistency, we specify the first `TextView` (`connection_state`) to be 200dp in width, and set the "fill_parent" keyword for the `Button`. Also note the property for our button "onClick". This parameter is important as it defines the handler for this button. In our case, clicking on this button will always call the function "uiEventHandler" – we will implement it later. After this, we will add the first separator as shown:

```
<View
    android:id="@+id/first_separator"
    android:layout_width="fill_parent"
    android:layout_height="1dp"
    android:layout_below="@id/top_section"
    android:background="@android:color/darker_gray"/>
```

The `View` element serves as a separator by being a 1px line that extends the whole window.

You might have noticed that we defined quite a few `"android:text"` elements. These need to correspond to elements found in the file `"strings.xml"` under **res | values**. We set them up like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">BBCClient</string>
  <string name="action_settings">Settings</string>
  <string name="connection_state_str">Not connected</string>
  <string name="connect_button_str">Connect</string>
</resources>
```

To see how the changes have affected our app, save the XML files and launch the application. Notice how without a single line of actual Java code, we already created roughly 30 percent of our application UI.

Let's continue defining the next section of the UI in the `activity_main.xml` file.

```
<RelativeLayout
  android:id="@+id/refresh_state_section"
  android:layout_alignParentLeft="true"
  android:layout_below="@id/first_separator"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal" >
  <TextView
    android:id="@+id/current_state_label"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:layout_centerVertical="true"
    android:text="@string/current_state_label_str" />
  <Button
    android:id="@+id/refresh_button"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/current_state_label"
    android:text="@string/refresh_state_str"
    android:enabled="false"
    android:onClick="uiEventHandler" />
</RelativeLayout>
```


We call this section "refresh_state_section", and it only holds one `TextView` with a `Button` element to refresh the state. Here, notice the new element in our "refresh_button" section called `enabled="false"`. As you have already guessed, this defines that our `Button` element will be disabled by default.

Next, we will define the section where the data from the temperature and light sensors is shown. Since this element has four components, we will use `LinearLayout` for it; with this layout type it will be easy to set them all with equal shares of the screen's real estate. Add the following section to the `activity_main.xml` file:

```
<LinearLayout
    android:id="@+id/actual_state_section"
    android:layout_alignParentLeft="true"
    android:layout_below="@id/refresh_state_section"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <TextView
        android:id="@+id/temperature_label"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="@string/temperature_label_str" />
    <TextView
        android:id="@+id/temperature_value"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="@string/temperature_value_str" />
    <TextView
        android:id="@+id/lightlevel_label"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="@string/lightlevel_label_str" />
    <TextView
        android:id="@+id/lightlevel_value"
        android:layout_width="0dp"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:text="@string/lightlevel_value_str" />
</LinearLayout>
```

With `LinearLayout`, we have set each `layout_width` component to `0dp`, and set `layout_weight` to `"1"`. This will make the layout manager choose an equal size for the components and spread them across the window with equal sizes.

Lastly, let's add the final alarm section, and the debug window.

```
<View
    android:id="@+id/second_separator"
    android:layout_width="wrap_content"
    android:layout_height="1dp" android:layout_marginTop="4dp"
    android:layout_below="@id/actual_state_section"
    android:background="@android:color/darker_gray" />
<RelativeLayout
    android:id="@+id/alarm_section"
    android:layout_alignParentLeft="true"
    android:layout_below="@id/second_separator"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <TextView
        android:id="@+id/alarm_label"
        android:layout_width="wrap_content"
        android:layout_height="30dp" android:layout_marginLeft="4dp"
        android:text="@string/alarm_label_str" />
    <TextView
        android:id="@+id/alarm_state"
        android:layout_width="wrap_content"
        android:layout_height="30dp"
        android:layout_toRightOf="@id/alarm_label"
        android:text="@string/alarm_state_str" />
    <Button
        android:id="@+id/alarm_reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/alarm_label" android:enabled="false"
        android:onClick="uiEventHandler"
        android:text="@string/alarm_reset_button_str"/>
</RelativeLayout>
<TextView
    android:id="@+id/debug_view"
    android:layout_width="fill_parent"
```

```
        android:layout_height="fill_parent"
        android:layout_below="@id/alarm_section"
        android:scrollbars = "vertical" android:maxLines = "500"
        android:focusable="true"
        android:focusableInTouchMode="true"
        android:background="@drawable/debug_view_border"
        android:text="@string/debug_window_str" />
    </RelativeLayout>
```

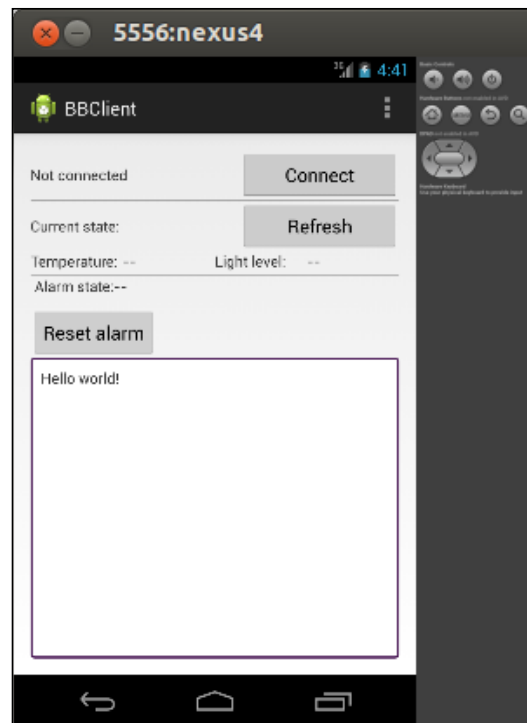
There is nothing new here, except in the last component "debug_view". We have used a new XML property defining a background for the TextView component. The "@drawable/debug_view_border" component actually points to the files inside the folder at **res | drawable** (there is one for each resolution). This property defines the background for the view. Add a new XML file called "debug_view_border.xml" to all the different "drawable" folders with following definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#FFFFFF"/> <corners android:radius="3dp" />
    <padding android:left="10dp" android:top="10dp"
        android:right="10dp" android:bottom="10dp" />
    <stroke android:color="#4B1A57" android:width="2dp" />
</shape>
```

There are still some strings that we need to define for our UI, those that we have referenced in our activity_main.xml file. Open strings.xml by navigating to **res | layout**, and add the following strings there:

```
<string name="current_state_label_str">Current state:</string>
<string name="refresh_state_str">Refresh</string>
<string name="temperature_label_str">Temperature:</string>
<string name="temperature_value_str">--</string>
<string name="lightlevel_label_str">Light level:</string>
<string name="lightlevel_value_str">--</string>
<string name="alarm_label_str">Alarm state:</string>
<string name="alarm_state_str">--</string>
<string name="alarm_reset_button_str">Reset alarm</string>
<string name="debug_window_str"></string>
```

Now our UI's main functionality is complete. When you launch this application, it should look like the following screenshot:



There is one more XML definition that we will have to define for this application. Remember how in the end of the previous chapter, we added support for a camera to Beagle? Well, in this chapter, you will be sending images from Beagle to your smartphone; so we need to create a new window for this purpose. Add a new layout resource (in the **Package Explorer** pane, right-click on the **res | layout** folder, and navigate to **New | Android XML**) called `downloaded_image_dialog.xml`, and add the following lines of code in it:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/dialog_root" android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp">
    <ImageView android:id="@+id/actual_image"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Now that our UI definition is ready, we can start implementing the actual beef around our application's bones.

Application permissions

Android can impose quite a fine-grained security control if desired, and these restrictions can be defined on a per-application basis. During development, each application has a file that defines which types of services it will request from the platform. The file that holds these permissions (and some other information) is called `AndroidManifest.xml`. It is in the base of your project directory structure.

Since our application uses socket connectivity, we need access to the Internet. For this purpose, you need to add the following two permissions to the manifest file (add the highlighted part):

```
...
<uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="14" />

<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE" />

<application
    android:allowBackup="true"
    ...
```

The support classes

As you remember, the first thing that we need to define is a communication protocol so that our programs can understand each other. From the toolbar, navigate to **New** | **Class** to create a new class named `BeagleProtocol.java`. Make sure that you create it in the same package as `MainActivity.java`. This class will contain exactly the same constants as before.

```
public final static short MT_REPLY_SHORT           = 10;
public final static short MT_REPLY_PL_CURRENT_CONFIG = 11;
public final static short MT_REPLY_CONFIG_CHANGE_RESULT = 12;
public final static short MT_REQUEST               = 13;
public final static short MT_REQUEST_PL_CONFIG_CHANGE = 14;

public final static short MT_INFO_INITIAL_HELLO     = 15;
public final static short MT_INFORM_ALARM_STATE     = 16;
public final static short MT_CAMERA_IMAGE_DATA      = 17;
public final static short MT_DISCONNECT             = 0;

public final static short ACTION_READ_TEMPERATURE  = 1;
public final static short ACTION_READ_LIGHT_LEVEL  = 2;
```

```

public final static short ACTION_READ_CONFIG_SETTINGS    = 3;
public final static short ACTION_CONFIG_CHANGE           = 4;
public final static short ACTION_READ_ALARM_STATE        = 5;
public final static short ACTION_RESET_ALARM             = 6;

public final static String CONFIG_ITEM_TEMPERATURE_READ_DELAY =
    "delay_between_temperature_measurements";
public final static String CONFIG_ITEM_LIGHTSENSOR_READ_DELAY =
    "delay_between_lightsensor_measurements";

public final static short REPLY_CONFIGURATION_CHANGE_OK    = 41;
public final static short REPLY_CONFIGURATION_CHANGE_FAILED = 42;
public final static short REPLY_ALARM_RESET_OK            = 43;

public final static short PAYLOAD_FIXED_SINGLE_VALUE = 2;
public final static short PROTOCOL_VERSION = 3;

//Android UI specific constants
public final static short SERVER_CONNECTED = 60;
public final static short PROTOCOL_READ    = 61;
public final static short RESET_ALARM      = 63;
public final static short RETRIEVE_IMAGE   = 64;
public final static short IMAGE_RETRIEVED  = 65;

```

Before we can start writing code that communicates with the server, we need to create one more helper class to communicate between the UI and network threads.



Android has a thread-safe UI framework, which is somewhat rare actually. This should (in theory) provide an always responsive UI; however, it does present some peculiar challenges and limitations. One of those is network communication. They must all be done in another thread, and they cannot touch any UI elements. There are a couple of mechanisms provided to still receive some type of feedback from a networking thread; for example, you can read about the `Timer`, `AsyncTask`, and `Handler` functionalities. In our program, we will be using a `Handler` functionality to pass messages "back to the UI".

Define yet another class called `Parcel.java`, and add the following lines to it:

```

public class Parcel {
    public String alarmState    = "";
    public String temperature   = "";
    public String lightlevel    = "";
    public short protocollevel = 0;
    public byte[] imagePointer;
}

```

The purpose of this class will be only to hold messages and data from the network thread that we want to pass to the UI via the `Handler` class. You will see how we can use it soon.

The main UI

Now that we have the required helper classes and UI definitions in place, let's start working on the main classes by adding some variable definitions to `MainActivity.java`.

```
private String SERVER_ADDR = "192.168.7.2"; //USB connected!
private int SERVER_PORT    = 7777;

// Variables that are used for UI
private int systemProtocolVer    = 0;
private String systemTemperature = "--";
private String systemLightLevel  = "--";
private String systemAlarmState  = "--";
private ImageView alarmImage      = null;

// Internal application state and helper variables
private String messageFromNetworkThread = "";
private boolean connectedToServer       = false;
private int lineHeight;

//Main context
Context mainContext;

// Handler that will receive messages from NetworkTask
final Handler uiHandler = new Handler();
// UI elements
private TextView debugWindow, connectionState, temperatureState,
    lightState, alarmState;
private Button connectButton, refreshButton, resetAlarmButton;
//Network thread
private NetworkTask nt;
```

Note here that we have *hardcoded* our server address to `192.168.7.2`. This means the emulator is going to try to connect to the Beagle *that is attached to the development PC via USB*. For example, on your phone, of course this address is not available, so when you want to connect to the Beagle with your actual phone, you will have to insert network cable to the board, and check the IP address (using `ifconfig eth0`). For now, you will have to have your phone also connected to your local network (Wi-Fi), since we are not using public IP addresses here. We will talk about connecting over your cell phone operator's network to your public IP address at the end of this chapter.

Next, we will define a debug output function that will print information to the debug view:

```
private void debug(String message) {
    Log.v("jlumme", "main/" + message);
    debugWindow.append(message + "\n");

    // Check if we should scroll
    if (debugWindow.getHeight() < (debugWindow.getLineCount()
        * lineHeight) ) {
        debugWindow.scrollTo(0,
            (debugWindow.getLineCount() * lineHeight ) -
            debugWindow.getHeight() + lineHeight);
    }
}
```

The debug function will print information to Android's **LogCat** (the logging mechanism in Android), and also to our debug window on the device. Notice that here we defined a keyword "jumme", which we use to filter out the output in LogCat; you should define your keyword, and add a filter to LogCat. We also use features of the TextView component to decide when we need to scroll our output window.

Next we will take a look at the onCreate method, which is the first method to be called when any activity is created. As this method is the onCreate method of our MainActivity.java file, it is also the entry point to our program.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mContext = this; //Save our context

    debugWindow = (TextView) findViewById(R.id.debug_view);
    debugWindow.setMovementMethod(new ScrollingMovementMethod());
    lineHeight = debugWindow.getLineHeight();
    connectionState = (TextView)
        findViewById(R.id.connection_state);
    temperatureState = (TextView)
        findViewById(R.id.temperature_value);
    lightState = (TextView) findViewById(R.id.lightlevel_value);
    alarmState = (TextView) findViewById(R.id.alarm_state);

    connectButton = (Button) findViewById(R.id.connect_button);
    refreshButton = (Button) findViewById(R.id.refresh_button);
    resetAlarmButton = (Button) findViewById(R.id.alarm_reset_button);
```



```
        debug("Application loaded");
    }
```

We can use the `findViewById` function to get handles to the UI elements that we have defined in the activity's XML files. We do this for future use and for convenience.

Next, let's add a function that will be responsible for showing the image sent by the server:

```
private void showDownloadedImage() {

    ImageView tempImageView = alarmImage;

    //Inflate a layout from our resource files
    AlertDialog.Builder imageDialog = new
        AlertDialog.Builder(mainContext);
    LayoutInflater inflater = LayoutInflater.from(mainContext);
    View layout = inflater.inflate(R.layout.downloaded_image_dialog,
        (ViewGroup) findViewById(R.id.dialog_root));
```

Most importantly, in this function, we set `tempImageView` to point to the variable `alarmImage`. This image has already been downloaded by `NetworkThread`, and the image data passed to the UI thread via the handler (we will show the code soon). After this, the function creates a new `AlertDialog`, and inflates the layout we defined in `downloaded_image_dialog.xml`. Lastly, we create a new `View` from the layout, and show the loaded image as follows:

```
    ImageView image = (ImageView)
        layout.findViewById(R.id.actual_image);
    image.setImageDrawable(tempImageView.getDrawable());
    imageDialog.setView(layout);
    imageDialog.setPositiveButton("OK" , new
        DialogInterface.OnClickListener() {

        //Cleanup will happen here
        public void onClick(DialogInterface dialog, int which) {
            dialog.dismiss();
            alarmImage = null;
        }
    });

    imageDialog.create();
    imageDialog.show();
} //showDownloadedImage()
```

After passing the image data to `ImageView`, we define a simple click listener for `DialogInterface` that will dispose of the view (and also the downloaded image). Lastly, show the view with the image to the user.

At this point, we have only defined the main control logic of the UI. Let's now define the function `uiEventHandler` that we mentioned during the definition of our UI layout XML file.

```
public void uiEventHandler(View v) {

    if (v == connectButton && !connectedToServer) {
        debug("Connecting to " + SERVER_ADDR + ":" + SERVER_PORT);
        new Thread(new Runnable() {
            public void run() {
                nt = new NetworkTask(SERVER_ADDR, SERVER_PORT, handler);
                nt.run();
            }
        }).start();
    }
    else if (v == connectButton && connectedToServer) {
        debug("Disconnect button pressed");
        nt.alive = false;
        nt = null;

        connectedToServer = false;
        connectionState.setText("Not connected");
        connectButton.setText("Connect");

        refreshButton.setEnabled(false);
        resetAlarmButton.setEnabled(false);

    } else if (v == refreshButton) {
        debug("refresh button pressed");
        nt.messageQueued = BeagleProtocol.REFRESH_DATA;
    } else if (v == resetAlarmButton) {
        debug("Alarm reset requested");
        nt.messageQueued = BeagleProtocol.RESET_ALARM;
    }
}
```

This function controls all of the application input from the user. It takes care of starting an instance of `NetworkTask` (in a separate thread), passing execution orders to it when appropriate, and also handles the connection and disconnection procedures. This is the function that you would touch if you add any new UI items, or change the execution flow of the application. Note also how we pass a "handler" pointer to the `NetworkTask` class. This handler will be defined next, and it is the way through which our network thread will be able to reach our UI thread.

Now we're left to define `Handler` and the support code that takes care of updating the UI. First, we will define the handler.

```
@SuppressWarnings("HandlerLeak")
final Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        Log.v("jlumme", "handler/message received");
        Parcel p;
        switch (msg.what) {
            case BeagleProtocol.PROTOCOL_READ:
                p = (Parcel) msg.obj;
                systemProtocolVer = p.protocollevel;
                messageFromNetworkThread = "Server protocol version: " +
                    Integer.toString(p.protocollevel);

                connectedToServer = true;
                break;

            case BeagleProtocol.REFRESH_DATA:
                Log.v("jlumme", "handler/Temperature read");
                p = (Parcel) msg.obj;
                systemAlarmState = p.alarmState;
                systemLightLevel = p.lightlevel;
                systemTemperature = p.temperature;
                messageFromNetworkThread = "Data refreshed";
                break;

            case BeagleProtocol.IMAGE_RETRIEVED:
                p = (Parcel) msg.obj;

                //Retrieve the image from the parcel
                Bitmap bmp=BitmapFactory.decodeByteArray
                    (p.imagePointer,0,p.imagePointer.length);
                alarmImage = new ImageView(mainContext);
```

```

        alarmImage.setImageBitmap(bmp);
        Log.v("jlumme", "Image loaded");
        break;

    default:
        break;
    }

    //Schedule UI update
    uiHandler.post(uiHandlerRunnable);
}
};

```

This code is called by `NetworkTask` to inform the UI that there is now some new information it needs to react to. This function receives a pointer to a `Message` class from the system that can be used to identify the different types of events that happened via the `Message.what` integer. Did you notice how we are using our `Parcel` class here? The `Message` class also has a variable `obj` that acts as a pointer to an arbitrary object; here we use it for the `Parcel` class. This pointer is set in `NetworkTask`, and since we know which data is available at which update (by checking `msg.what`), we can safely access the available information.

This way we are free to send any type of data back, as long as we take care to properly access those objects. After we have updated our internal states and variables, we schedule an actual UI update to take place sometime later using `uiHandlerRunnable`.



It might be difficult to immediately see the relation between the UI thread and `NetworkTask` (after all, we started it in a new thread!), but just keep in mind that the UI thread doesn't want to depend on any other thread. It only has some "trusted friends" (such as `uiHandlerRunnable`) who can pass its information when convenient, but strangers are not welcome.

In the previous code, we used `uiHandler` to schedule a UI update, and now we will add the necessary `Runnable` that will perform it.

```

final Runnable uiHandlerRunnable = new Runnable() {
    public void run() {
        Log.v("jlumme", "Updating the UI");
        if (messageFromNetworkThread != "") {
            debug(messageFromNetworkThread);
            messageFromNetworkThread = "";
        }
    }
}

```

```
if (connectedToServer) {
    connectionState.setText("Connected");
    connectButton.setText("Disconnect");
    refreshButton.setEnabled(true);
    resetAlarmButton.setEnabled(true);
}

temperatureState.setText(systemTemperature);
lightState.setText(systemLightLevel);
alarmState.setText(systemAlarmState);
```

The `run()` method here will be executed at some point after we have scheduled it via the handler, and the UI components are updated with new data. When the user requests a refresh of the current server state (temperature, and so on), the alarm state is retrieved from the server. Now, we will perform some additional processing for an alarm triggered on the server.

```
//Check if image retrieval should be scheduled
if( !systemAlarmState.toLowerCase().equals("all ok") &&
    !systemAlarmState.equals("--") && alarmImage == null) {

    //Build a temporary dialog
    new AlertDialog.Builder(mainContext)
        .setTitle("Alert occurred")
        .setMessage("Would you like to retrieve the image captured
            during the alert?")
        .setPositiveButton("Yes", new
            DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                    //Schedule image retrieval from the server
                    nt.messageQueued = BeagleProtocol.RETRIEVE_IMAGE;
                }
            }).setNegativeButton("No", new
            DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int
                    which) {}
            }).show();
    }

    //If image has been loaded
    if (alarmImage != null ) {
        showDownloadedImage();
    }
} //run()
}; //Internal Runnable
} //MainActivity.java
```

If an alarm has been triggered, an `AlertDialog` is created to ask the user whether he would like to retrieve the image that was taken when the alarm was triggered. The handler code is also defined there to initiate an `RETRIEVE_IMAGE` handling in `NetworkTask`, if the user chooses to do so. Finally, there is a small piece of code that checks whether an image has been loaded, and we should show it to the user.

That was it for our `MainActivity.java` class.

The network thread

Now we still need to define `NetworkTask`; it handles the actual communication with the server. Let's get right to it; create a new class called `NetworkTask.java`:

```
public class NetworkTask implements Runnable {

    private boolean datastreamDebug = true;
    public boolean alive             = true;

    //Holds the image that is loaded from the server
    private byte[] alarmImage;

    //Connection related variables
    private Socket socket;
    private DataInputStream is;
    private DataOutputStream os;
    private String  serverAddress = "";
    private int serverPort        = 0;

    // Used to send messages for UI thread
    private Handler parent;

    // UI sets this variable to request network activity
    public int messageQueued = 0;

    // Simple output function to LogCat
    private void debug(String message) {
        Log.v("jlumme", "nt/" + message);
    }

    // Function to get a short from 2 consecutive bytes in stream
    public static short getSingleShort(byte[] arr, int off) {
        return (short) (arr[off]<<8 &0xFF00 | arr[off+1]&0xFF);
    }
}
```

```
// Constructor
NetworkTask(String address, int port, Handler handler) {
    serverAddress = address;
    serverPort    = port;
    parent        = handler;
    alive = true; //Set to false to end the thread
}
```

NetworkTask extends Runnable, so that it can live separately from our main thread, as network activity is not permitted in the main application thread in Android. The constructor takes the target node information for the socket connection. The helper function `getSingleShort()` is useful when working with stream data, as we need to combine two consecutive bytes to get a Short type. You will see it in use quite often when we parse data from the server.

Now let's add the functionality to connect to our server.

```
@Override
public void run() {
    try {
        debug("Connecting to " + serverAddress);
        socket = new Socket(serverAddress, serverPort);

        debug("Socket established, attempt reading data");
        is = new DataInputStream(socket.getInputStream());
        os = new DataOutputStream(socket.getOutputStream());
        debug("Streams are ready");

        //Read the initial welcome message
        byte[] readArray = new byte[6];
        int howmany = is.read(readArray, 0, 6);

        if (howmany == -1) {
            debug("Stream has closed. Something wrong with login ?");
            is.close();
            os.close();
            socket.close();
            return;
        }
        debug("Server protocol v:" + getSingleShort(readArray, 4));

        //Inform the UI about protocol version
        Parcel p = new Parcel();
        p.protocollevel = getSingleShort(readArray, 4);
    }
}
```

```

Message msg = new Message();
msg.what = BeagleProtocol.PROTOCOL_READ;
msg.obj = p;

parent.handleMessage(msg);

```

When the `NetworkActivity` thread is started, the `run()` method is called. Much in the same way as with our Python client, we create a socket connection, and read the initial welcome message before we start acting on any particular requests. In Java, we also need to initialize the input and output streams for our socket. Here you can also see the first use of our `Parcel` class as the content for the message handler. We initialize a `Message` class to hold the type of message (`msg.what`) and our object pointer to the `Parcel` class. This is the way we can pass data to `MainActivity`, without breaking the UI-threading rules.

Next, `NetworkTask` will start a wait loop, where it will wait for input from the UI to perform other actions within the stream.

```

//Message handler loop
while (alive) {
    if (messageQueued == 0) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ie) {}
    } else {

        debug("msg received, processing");

        //Our reply parcel to UI
        Parcel reply = new Parcel();

        //The output data package
        byte[] dout = new byte[6];
        dout[0] = (byte)
            ((BeagleProtocol.MT_REQUEST>> 8) & 0xff);
        dout[1] = (byte) (BeagleProtocol.MT_REQUEST& 0xff);
        dout[2] = (byte)
            ((BeagleProtocol.PAYLOAD_FIXED_SINGLE_VALUE>> 8) & 0xff);
        dout[3] = (byte)
            (BeagleProtocol.PAYLOAD_FIXED_SINGLE_VALUE& 0xff);

        switch (messageQueued) {
        case BeagleProtocol.REFRESH_DATA:
            debug("Data refresh requested");

```



```
//Construct message to fetch temperature
dout[4] = (byte)
    ((BeagleProtocol.ACTION_READ_TEMPERATURE>> 8) & 0xff);
dout[5] = (byte)
    (BeagleProtocol.ACTION_READ_TEMPERATURE& 0xff);

os.write(dout); // Send request
```

When the thread notices that a message has been scheduled to be sent (variable `messageQueued` is not zero), it wakes up, prepares a `Parcel` variable for holding the reply, and prepares a byte array of 6 bytes that will hold the outgoing message. Notice how we preset the values of the first two headers before we even checked the message type. This is because our client currently only sends `MT_REQUEST` messages, and we can reuse the same array. Next, the thread starts responding to the `REFRESH_DATA` action from the UI.



The byte array `dout` is controlled byte by byte, and if you are not familiar with **bit shifting**, this array population might look quite confusing. As you remember, our protocol uses `Short` types to relay different kinds of information. On 32-bit machines, one `Short` is usually 2 bytes. So we need to fill out each byte by the proper `Short` "part".

Consider our `MT_REQUEST` (value 20); on bit level, this is represented as 0000 0000 0001 0100. So here we have 16 bits, or 2 bytes. Now we have a memory block of 2 bytes to fill with this value of 20. Since our system is big-endian, we need to store the most significant bytes first, thus, we have to shift 0000 0000 0001 0100 *right* by 8 bits. Technically, we are filling the byte `dout[0]` with zeros because all our values in the protocol are under 255 (the maximum value is represented by 1 byte), but if you were to have a lot of message types, or use values bigger than 255, bit shifting is needed to get the correct data for a single byte. Phew, that was a mouthful!

After sending a message, we will do the same as with our client; we wait for a reply:

```
// Reading reply
reply.temperature = readReplyFromServer
    (BeagleProtocol.ACTION_READ_TEMPERATURE);
```

We handle a reply from a server in a function `readReplyFromServer` (defined shortly), and once we have read the temperature we fill the `Parcel` variable with data, and move onto reading the next variable.

```
//We can use the previous data as is
dout[4] = (byte)
    ((BeagleProtocol.ACTION_READ_LIGHT_LEVEL>> 8) & 0xff);
```

```

dout[5] = (byte)
(BeagleProtocol.ACTION_READ_LIGHT_LEVEL& 0xff);
os.write(dout);

reply.lightlevel = readReplyFromServer
(BeagleProtocol.ACTION_READ_LIGHT_LEVEL);

//We can use the previous data as is
dout[4] = (byte)
((BeagleProtocol.ACTION_READ_ALARM_STATE>> 8) & 0xff);
dout[5] = (byte)
(BeagleProtocol.ACTION_READ_ALARM_STATE& 0xff);
os.write(dout);

reply.alarmState =readReplyFromServer
(BeagleProtocol.ACTION_READ_ALARM_STATE);

//Send the reply to UI thread
Message refresh = new Message();
refresh.what = BeagleProtocol.REFRESH_DATA;
refresh.obj = reply;

parent.handleMessage(refresh);
break;

```

Once we have filled our Parcel variable with all the necessary data, we set the appropriate Message type, and send the parcel along with it to the UI.

The handling of a RESET_ALARM message is very similar; only the bytes that define the action change.

```

case BeagleProtocol.RESET_ALARM:
    debug("Alarm reset requested");
    dout[4] = (byte)
    ((BeagleProtocol.ACTION_RESET_ALARM>> 8) & 0xff);
    dout[5] = (byte) (BeagleProtocol.ACTION_RESET_ALARM& 0xff);
    os.write(rstMsg);

    reply.alarmState = readReplyFromServer
    (BeagleProtocol.ACTION_RESET_ALARM);
    break;

```

Lastly, we handle the RETRIEVE_IMAGE case.

```
case BeagleProtocol.RETRIEVE_IMAGE:
    debug("Retrieve camera image");
    //Read image from server:

    dout[4] = (byte)
        ((BeagleProtocol.ACTION_RETRIEVE_IMAGE >> 8) & 0xff);
    dout[5] = (byte)
        (BeagleProtocol.ACTION_RETRIEVE_IMAGE & 0xff);

    os.write(dout); // Send request
    readReplyFromServer(
        BeagleProtocol.ACTION_RETRIEVE_IMAGE);

    reply.imagePointer = alarmImage;

    //Send the reply to UI thread
    Message camImg = new Message();
    camImg.what = BeagleProtocol.IMAGE_RETRIEVED;
    camImg.obj = reply;

    parent.handleMessage(camImg);
    //Image reading done
    break;
}
dout = null;
messageQueued = 0;
} // messageQueued != 0
} //while(alive)
```

After this, we need to close the streams.

```
debug("Closing streams");
is.close();
os.close();
socket.close();
} catch (IOException e) {
    //Failed closing streams... not much we can do here.
    e.printStackTrace();
}
} //run()
```

There is one more function that we need to talk about; this is the function that reads a reply from the server and processes the result:

```
private String readReplyFromServer(int messageType) {
    String ret = "";
    int i = 0;
    byte[] incomingData;
    try {
        switch (messageType) {
            case BeagleProtocol.ACTION_READ_TEMPERATURE:
            case BeagleProtocol.ACTION_READ_LIGHT_LEVEL:
                debug("reading alarm temp/light");

                incomingData = new byte[6];
                i = is.read(incomingData, 0, 6);

                if (datastreamDebug) {
                    debug("Server replied with integer:" +
                        getSingleShort(incomingData, 4) + "
                        (we read " + i + " bytes)");
                }
                i = getSingleShort(incomingData, 4);
                ret = Integer.toString(i);

                incomingData = null; //release for garbage collection
                break;
        }
    }
}
```

This function is called to read a reply from the input stream. Since we know the message we have sent, this information is sent to the function as input, to make the processing simpler. We need to be more careful when we are reading the alarm state and receiving the captured camera image, since we don't know the exact size of the reply. Otherwise, we can read 6 bytes right away. The rest of the function is very similar in structure.

```
case BeagleProtocol.ACTION_READ_ALARM_STATE:
    debug("reading alarm state");
    incomingData = new byte[4];
    int additionalBytesWaiting = 0;

    i = is.read(incomingData, 0, 4);
    if (datastreamDebug) {
        debug("Server informed about remaining bytes:" +
            getSingleShort(incomingData,
                2) + " (we read " + i + " bytes)");
    }
}
```

```
        additionalBytesWaiting = getSingleShort(incomingData, 2);

        //Initialize buffer for reply
        incomingData = new byte[additionalBytesWaiting];
        i = is.read(incomingData, 0, additionalBytesWaiting);
        if (datastreamDebug) {
            debug("Server sent us rest of datam, (we read " + i + "
                bytes)");
        }
        ret = new String(incomingData);

        break;
    case BeagleProtocol.ACTION_RESET_ALARM:
        debug("reading alarm reset reply");
        incomingData = new byte[6];
        i = is.read(incomingData, 0, 6);

        if (datastreamDebug) {
            debug("Server replied with integer:" +
                getSingleShort(incomingData, 4) + "
                (we read " + i + " bytes)");
        }
        i = getSingleShort(incomingData, 4);

        if (i == BeagleProtocol.REPLY_ALARM_RESET_OK) {
            ret = "Alarm reset";
        } else {
            ret = "Reset failed";
        }
        break;
```

Now we are left with one more reply type to process, which is the ACTION_RETRIEVE_IMAGE reply type; it's a bit different from the previous functions, as we could (theoretically) have payload data in the size of megabytes here. It starts in the usual way:

```
    case BeagleProtocol.ACTION_RETRIEVE_IMAGE:
        //Header in this message is 4+4 bytes:
        //2 bytes for msg type, 2 bytes for "zero size"
        //and 4 for actual size of the incoming image
        //Maximum image size for 2 bytes would have been only 65535B
        incomingData = new byte[8];
        i = is.read(incomingData, 0, 8);
```

```
//To construct the incoming image size
byte[] size = new byte[4];
System.arraycopy(incomingData, 4, size, 0, 4);

ByteBuffer bb = ByteBuffer.wrap(size);
int remaining = bb.getInt();

if (datastreamDebug) {
    debug("Server informed about remaining bytes:" + remaining +
        " (we read " + i + " bytes)");
}
```

You can see here that our header is actually twisting the rules of our protocol a bit. If we would use the regular 2 bytes for the image size, we would be limited to a 65535-byte image, which we could of course do, but the quality would suffer and we would possibly have to drop the resolution as well. What the server did here instead, is send 0 bytes in the regular size field, and then append another 4-byte value that holds the actual image size. Since the message still has a distinctive header `MT_CAMERA_IMAGE_DATA`, we can always prepare for this (though technically, the client already knows that the next reply should be image data). Now that the client knows the incoming image size, it can proceed with reading it.

```
//Separate byte array that holds the alarm image
alarmImage = new byte[remaining];

//Holds the value how many bytes we have already read
int read_index = 0;

//Loop to read all the data
while (read_index < remaining) {
    i = is.read(alarmImage, read_index, remaining - read_index);
    //Reading the complete package, as long as it takes
    read_index += i;
}

if (datastreamDebug) {
    debug("Server sent us rest of datam, (we read " + read_index
        + " bytes)");
}

size = null;
break;
```

```
    }  
    } catch (IOException ioe) {  
        debug("ERROR/Failed reading from the stream:");  
        ioe.printStackTrace();  
    }  
  
    incomingData = null;  
    return ret;  
} //readReplyFromServer  
} //NetworkTask
```

We are sending quite a bit of data here (well, as compared to our normal message sizes), and while it's okay to ask the socket to read, for example, 100 kilobytes from the stream, depending on its internal implementation, it most likely won't do that. The `read` function always returns the actual number of bytes read; so here we have to keep track of how much data the socket has already retrieved, and how much is still left, and keep looping and filling our `alarmImage` byte array with all the image data until the transfer completes.

This is all that is required from the application to read data from the server, and also send commands to it. We didn't build all the interfaces yet; from here, it should be fairly simple for you to extend the application and implement other interfaces that we haven't implemented yet. For example, as you can see, the **Settings** button we have doesn't currently do anything. Wouldn't it be nice if we could set the timers in a similar fashion as with the client application (perhaps this time without having to type them)?

Before you can test the application against your server, you also need to add the new protocol definitions to the server-side protocol, and add the functionality to handle these messages. Let's look at the required additions to send images over to the client.

The new server features

We assume that you will use a regular USB camera for Beagle, so we will base this example also on that premise. However, if you are using a Camera Cape here, you will need to change the GPIO mappings, as the Camera Cape uses a lot of the same pins as the header P8 (the one we have used). Both cameras use the default `/dev/video0` interface, so we do not need to modify `camera_ctrl.py` from *Chapter 5, Implementing Periodic Tasks*.

First, we also need to add the following new headers to the `beagle_protocol.py` file:

```
MT_INFORM_ALARM_STATE = 16
MT_CAMERA_IMAGE_DATA   = 17

ACTION_READ_ALARM_STATE = 5
ACTION_RESET_ALARM      = 6
ACTION_RETRIEVE_IMAGE   = 7

REPLY_ALARM_RESET_OK    = 43
```

Also update your protocol version to Version 3. Let's create a new class called `pir_control.py`; we basically just encapsulate the code that we wrote for the test application in the previous chapter, and add some file writing logic and camera controls to take the picture when the alarm is triggered.

```
import Adafruit_BBIO.GPIO as GPIO
import time

import camera_ctrl as camera

# Configuration
ALARM_TRIGGER = 6
alert_datafile = "" #set when caller starts this thread

def update_alarm_data(new_state):
    global alert_datafile

    try:
        output = open (alert_datafile, "w")
        output.write(new_state + "\n")
        output.flush()
        output.close()
    except Exception, e:
        print "pir_control/Failed writing new alarm state"
        print e
```


We define two configuration variables, and create a function that will save the alarm state to a file. Then we add the same old GPIO initialization code and LED-triggering functions.

```
def indicate_alert(alarm_level):

    if alarm_level == 0:
        GPIO.output("P8_10", GPIO.LOW)
        GPIO.output("P8_12", GPIO.LOW)
        GPIO.output("P8_14", GPIO.LOW)
        GPIO.output("P8_16", GPIO.LOW)
        GPIO.output("P8_18", GPIO.LOW)
        GPIO.output("P8_26", GPIO.LOW)

    if alarm_level >= 1:
        GPIO.output("P8_10", GPIO.HIGH)
    if alarm_level >= 2:
        GPIO.output("P8_12", GPIO.HIGH)
    if alarm_level >= 3:
        GPIO.output("P8_14", GPIO.HIGH)
    if alarm_level >= 4:
        GPIO.output("P8_16", GPIO.HIGH)
    if alarm_level >= 5:
        GPIO.output("P8_18", GPIO.HIGH)
    if alarm_level >= 6:
        GPIO.output("P8_26", GPIO.HIGH)

def init_gpio():
    GPIO.setup("P8_8", GPIO.IN)
    GPIO.setup("P8_10", GPIO.OUT)
    GPIO.setup("P8_12", GPIO.OUT)
    GPIO.setup("P8_14", GPIO.OUT)
    GPIO.setup("P8_16", GPIO.OUT)
    GPIO.setup("P8_18", GPIO.OUT)
    GPIO.setup("P8_26", GPIO.OUT)
```

Lastly, the code that will manage the PIR sensor and use `camera_ctrl.py` to take a picture when the alarm is triggered is as follows:

```
def start_monitoring(alarm_state_filename, alarm_image_filename):
    global alert_datafile
    alert_datafile = alarm_state_filename
```

```
alarm_status = "All ok"
update_alarm_data(alarm_status)

alarm_buffer = 0
init_gpio()

# Initialize the camera
camera.init_camera()

while True:
    new_state = GPIO.input("P8_8")
    if new_state == 0:
        alarm_buffer += 1
        indicate_alert(alarm_buffer)

        if alarm_buffer > ALARM_TRIGGER:
            alarm_buffer = 0

            alarm_status = "@%s ALARM" % time.strftime("%Y-%m-%d
                %H:%M:%S")
            print alarm_status
            update_alarm_data(alarm_status)

            # Take a picture
            camera.take_picture(alarm_image_filename)

            for i in range(0,5):
                GPIO.output("P8_26", GPIO.HIGH)
                time.sleep(0.3)
                GPIO.output("P8_26", GPIO.LOW)
                time.sleep(0.3)

    else:
        alarm_buffer = 0
        indicate_alert(alarm_buffer)

    time.sleep(0.1)
if __name__ == "__main__":
    init_gpio()
    start_monitoring("alarm_testfile.txt", "alarm.jpg")
```

Now our support code is ready, and we can start working on the interfaces to control data transfer with the client. Open our `beagle_server.py` file and start by adding a couple of variables to the configuration section, and functions to handle the alarm operations, as shown in the following lines of code:

```
import pir_ctrl
...
alert_datafile = ".alert_data.txt"
alarm_image_name = "alarm.jpg"
def reset_alarm_state():
    alarm_status = "All ok"
    try:
        alarmfile = open(alert_datafile, "w")
        alarmfile.write(alarm_status + "\n")
        alarmfile.close() # Will also flush the data
    except Exception, err:
        print "Problem writing to alarm file"
        print err
    return BP.REPLY_ALARM_RESET_OK
def get_alarm_state():
    print "someone asking for alarm status"
    alarm = "unknown"
    try:
        alarm = open(alert_datafile, "r").read()
        alarm = alarm.replace("\n", "") #Remove line breaks
        print "read: %s" % alarm
    except Exception, err:
        print "Problem reading alarm file"
        print err
    return alarm
```

Both of these functions are quite simple, they only read or clear the `alarm` file. The actual file population with the alarm data is the responsibility of the `pir_ctrl.py` class. Let's define the function that starts the monitoring.

```
def monitor_pir():
    global alarm_image_name
    global alert_datafile
    print "PIR sensor monitoring starting"
    pir_ctrl.start_monitoring(alert_datafile, alarm_image_name)
```

This function also needs to be called in our main method. We start the following new process for it:

```
if __name__ == "__main__":
    print "Starting Beagle home automation server.."
    reload_configuration()
    #Start PIR monitoring
    alarm_monitor = Process(target=monitor_pir, args=())
    alarm_monitor.start()
```

We have our new operative functions ready; we are now left with modifying the `handle_client_request` and `create_data_packet` functions to serve the clients. First, we will add the following support code to handle the newly defined client requests:

```
...
data_packet = create_data_packet
    (BP.MT_REPLY_PL_CURRENT_CONFIG, reply, length)
if not data_packet == None:
    cs.sendall(data_packet)

elif request == BP.ACTION_READ_ALARM_STATE:
    reply = get_alarm_state()
    length = len(reply)

    data_packet = create_data_packet(BP.MT_INFORM_ALARM_STATE,
        reply, length)
    if not data_packet == None:
        cs.sendall(data_packet)

elif request == BP.ACTION_RESET_ALARM:
    reply = reset_alarm_state()

    data_packet = create_data_packet
        (BP.MT_REPLY_SHORT,reply, BP.PAYLOAD_FIXED_SINGLE_VALUE)
    if not data_packet == None:
        cs.sendall(data_packet)

elif request == BP.ACTION_RETRIEVE_IMAGE:

    try:
        imagefile = open (alarm_image_name, 'rb').read()
        length = os.stat(alarm_image_name).st_size #Get size
```

```
data_packet = create_data_packet
                (BP.MT_CAMERA_IMAGE_DATA, imagefile, length)
            if not data_packet == None:
                cs.sendall(data_packet) # Send the header data first
                cs.sendall(imagefile)  # No packing for image bytes
        except Exception, err:
            print "Sending alarm image failed:"
            print err
```

There is some new code in the `ACTION_RETRIEVE_IMAGE` section that is required when we send the image to the client. First of all, we use the `File` open function with the parameter `'rb'` to read the image data as binary. Then we use `os.stat().st_size` to get the exact size in bytes. With this data, we first prepare a header message (which needs to be packed as usual) and send just the byte size of the image data. Since this is already in native format, no packing is needed. Lastly, let's add the following support code to send two new messages to `create_data_packet`:

```
elif msg_type == BP.MT_REPLY_CONFIG_CHANGE_RESULT:
    packet = pack("!HHH", BP.MT_REPLY_CONFIG_CHANGE_RESULT,
                  data_length, data)

elif msg_type == BP.MT_INFORM_ALARM_STATE:
    pack_string = "!HH%s" % data_length
    packet = pack(pack_string, BP.MT_INFORM_ALARM_STATE,
                  data_length, data)

elif msg_type == BP.MT_CAMERA_IMAGE_DATA:
    pack_string = "!HHL"
    packet = pack(pack_string, BP.MT_CAMERA_IMAGE_DATA, 0,
                  data_length)
```

The only new bit here is the `pack_string` field for `MT_CAMERA_IMAGE`, as we are using a long value to send a 4-byte value (to accommodate the possible size of the image).

These are all of the changes that this chapter requires. It was quite a bit of new code, wasn't it? Now our server will be able to monitor its designated area, and if any movement is detected in there, it will take a picture of the person/thing triggering the alarm, and send the image over to our Android application. With these additions, our server can currently only remember the latest alarm that has been triggered. How about adding support for the server to log all of the alarms that are triggered, instead of just remembering the last one? Also, how about providing support for the client to retrieve the whole list, or just the latest alarm? This can be a selection the user can choose from. There is a lot of stuff we could do with the camera as well. For example, we can now use a module named `pygame.movie`.

There is one last thing we would like to talk about in this chapter, and that's how you can connect your shiny new application to your home server from outside your home, using your cell phone provider's network.

Working from outside your home network

There are two things that you have to consider if you want to use your application from outside your home network. First of all, the IP address will no longer be part of your home network, so you will not be able to connect to your server from another network as easily as with the internal **LAN IP**. If you are so lucky as to have a fixed IP and possibly even a domain address, you could still use a similar method as of now, but a majority of the users are using connections where the IP address will change from time to time.

Fortunately, there are services on the Internet that also allow the "static" addressing of dynamic IP addresses. Such services are called **DDNS** or **Dynamic DNS**. They work in this manner. The users first register for the service, and inform the service of their current IP address. After that, the service will map a (more or less) easily remembered DNS address to this IP address. After this, the users will be able to connect to their dynamic IP address through this "static" DNS record (of course, you need to have some type of mechanism to keep this information up-to-date. Many of these services have a program that can do this for you). As of this writing, the following web services provide this functionality for free:

- **No-IP**: The URL is [http:// www.no-ip.com/](http://www.no-ip.com/)
- **DNSdynamic**: The URL is <http://www.dnsdynamic.org/>
- **Change IP**: The URL is <http://www.changeip.com/>
- **FreeDNS**: The URL is <http://freedns.afraid.org/>

Another thing one needs to keep in mind is that your home network most likely has a router that is handling the connection to the Internet. In these cases, the users will most likely have only one external IP address that belongs to the router. The rest of the machines in your home network will have a different IP address that is not visible on the Internet. For these machines to connect to the Internet, there has to be a **Network Address Translation (NAT)** service running on the router.

To connect to a machine behind a NAT, you need the port forwarding service to be active on the router. This means that if someone is trying to access your external IP address with a specific port (such as 7777), the router will know that this connection is intended for Beagle and forward it to the appropriate LAN IP address.

The port forwarding setup will, of course, vary depending on the router manufacturer, but the principle will always remain the same. First, you need to find out the internal IP address of your router. This is the **Default Gateway** address, and it is listed in your Internet connection settings.

On Windows, to open the command prompt press **Start**, and in the search field type "cmd" and press the *Enter* key. It should open a new command prompt window; here you need to type `ipconfig /all`. This will show you all the network settings, and you should be able to find the default gateway.

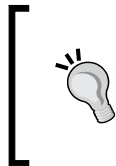
On Mac OS X, open the terminal, and type `netstat -nr`.

On Linux, also in the terminal, type `route`.

Then you will need to use your web browser, to connect to that address. Enter the password that you have set, and look for something like "Port forwarding" or "WAN forwarding/settings". You can usually recognize this service as the different services are listed along with their port numbers. Also the functionality to enter different IP addresses to forward to would be listed there, and so on. Googling for "port forwarding settings *your router brand/model*" should give you quite a good idea of what these settings will look like on your router.

Once you have found the settings, add the local IP address (for the `eth0` address, check with `ifconfig eth0` on the board) of Beagle, and set the port to 7777 (or to the port you have chosen, if you have chosen a different one) to be forwarded from outside to inside.

You can also test your port-forwarding configuration using several online tools. For example, the tool at <http://www.yougetsignal.com/tools/open-ports/> is quite nice as it automatically retrieves your external IP address, and all you have to do is just enter the port for testing.



Also take note that if you don't connect Beagle to the router for several days (it depends on the manufacturer to set the IP lease time), the router might "recycle" the IP address back to its "free pool", and assign it to someone else next time. So you might want to consider assigning a static IP address to it.



Summary

This chapter was quite Android-heavy, and if you just started working with Android, you might feel somewhat nauseous. I hope we didn't scare you off by practically dumping all that code on you. Even if you didn't have much experience before, we're sure that you're off to a great start now. It's exciting to play around with your embedded server from your phone. Hopefully, you have also had the time to extend the examples that we showed in each chapter, as we're sure you have many ideas on how to build something on your own. In *Appendix, Security, Debugging, and I2C and SPI*, among other things we will talk about some more complicated hardware integration examples that we will not touch in this book. We will introduce serial port debugging and data transfer basics using the SPI and I2C buses.

We would like to leave you with one more thought. Consider the security aspect seriously. If you are going to add an interface to control the electric appliances of your house, nobody else is responsible for this other than you. You need to be sure about your electronic circuits, and also consider the possibility of someone trying to crack their way into your server if they find out about it. Throughout this book we have used a very simple protocol; it can be deciphered in minutes if needed, and without any kind of safeguards. In *Appendix, Security, Debugging, and I2C and SPI*, we will show you how to implement secure encryption and login functionalities, and you might want to consider applying the same to all of your communication tools at some point.

Security, Debugging, and I2C and SPI

In this appendix, we would like to bring you more general information about interfacing your Beagle hardware with more complicated devices, such as GPS receivers, wireless modules, or perhaps your own microcontroller-driven design for some special purpose. When interfacing other self-operating devices, there are a few industry standards which we will talk about in detail. An introduction to advanced embedded-device-debugging techniques will also be presented, and we will talk about the security issues in the current protocol implementation and how to increase the security by using data encryption. After reading you will have:

- A general knowledge of the I2C and SPI buses
- An understanding of what is needed for the Linux kernel, or ARM processor debugging
- Implemented an encrypted data transfer between our client and server

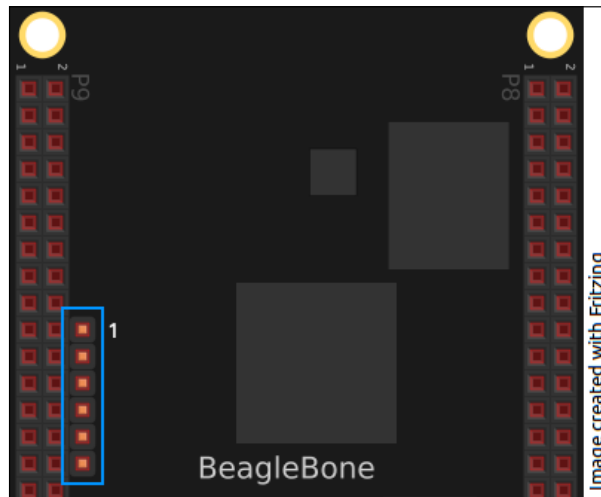
In the last section of this appendix, you will also find the complete mapping of headers P8 and P9. This can serve as a handy reference to find out which pins are usable for you, and where to find the desired functionality.

Kernel traces and advanced debugging

We haven't touched upon this topic earlier in the book, but once you get more familiar with Beagle, at some point you might start wondering how to integrate some other hardware, change the kernel configuration, or even change something in the current hardware setup. In all these cases, you will need some more debugging capabilities.

Boot time kernel traces

The least you will need to do to identify the possible problems with your new kernel configuration is to see what is happening during the boot process. For this purpose Beagle has serial connector pins in the header **J1**. This header is next to the **P9** pin:



The **pin1** pin is marked in the preceding screenshot, and it's the one that is closest to the power supply plug. To get serial input/output from this port, the required pins are:

Board pin	Function
1	Ground
4	Receive (Rx)
5	Transmit (Tx)

Keep in mind that the voltage in these pins is 3.3 volts, and that your USB port operates at 5 volts, so you will need a **level shifter** between them to properly read/transmit data. There are ready-made USB serial-level shifter cables from several companies available. The price should be roughly around 10 USD, and you should take one with "female jumper wires" on the other end. The settings for the terminal connection will need to be as follows:

Baud rate: 115200, Bits 8, No parity, Stop bits 1, Handshake none

Now you will be able to see all the kernel traces even during the initial boot time when connecting to the serial port on your host computer.

JTAG debugging

Our board also has the capability for proper hardcore debugging through JTAG with, for example, **OpenOCD**. However, as the board ships without the necessary header, you will need to solder that in place yourself. This board uses the Texas Instruments **cTI** male **JTAG** header, so you will need to solder an ARM20cTI20 JTAG adapter to the board in order to use the normal JTAG cable and debugger. The unpopulated header **P2** is found on the backside of the board, next to **P9**. Most of the users will not need its deep-level debugging capabilities, so we won't talk about setting it up. With this small introduction we're sure that you can find the necessary information to go forward, if you do.

The I2C and SPI buses

These two buses are a fairly common way to interconnect the different devices inside embedded designs. They are not very complicated, and they do not need many wires to operate. Also, their big advantage is easy debugging, as there is only one relatively slow data line, which you can decode easily with just a pen and paper (OK, maybe you do need a bit of additional hardware, such as a logic analyzer), when trying to find the reasons why your bus is not working, or if the data is getting corrupted.

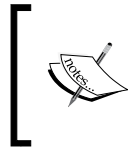
Their disadvantage is perhaps their speed, since they are not too fast. Depending on the protocol and its version, the speeds are at maximum in the Mbps range, usually less. But with low power peripherals, this is often more than enough.

Most of the time the devices using these communication buses have low-level drivers available, so you only need to make the physical connection, configure the bus settings, and implement the necessary data-handling logic.

The I2C bus

The **Inter-Integrated Circuit (I2C)** bus requires only two lines, **SCL (serial clock)** and **SDA (serial data)**. It has addressing properties, so you can use it to communicate with multiple devices connected to the same bus. The SCL bus is the clock line, and it is used by the master to synchronize the transfers across the bus. The SDA bus is the data line, and this line is operated on by both the master and the slave.

Both of the lines are open-drain designs, so they both require one pull-up resistor, as only the controller can drive the line low.

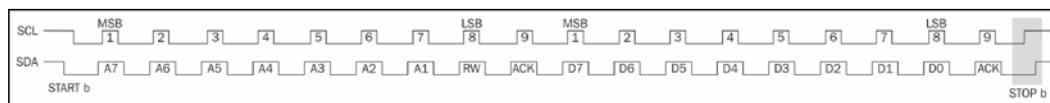


You can think of an open-drain circuit acting as a switch to the ground, driven with a FET. The circuit can only connect the line to the ground and because of this, we need a pull-up resistor that drives the line high otherwise. The open-collector design creates the same circuit, but the transistor implementing the switch is a BJT.

Only the master initiates the connections, and the slave waits for commands from a master device. Any device can, in theory, be the master; the point is that the master "sets the pace" of the clock line. The operation is performed in the following order:

1. Start signal generation
2. Slave address transfer
3. Data transfer
4. Stop signal generation

Transferring a single byte looks like this:



Generating the start signal

When there is no activity on the bus, both the SDA and SCL lines are high (the pull-up resistors) and the master can initiate a new transfer. The transfer is initiated by sending the "start sequence" to the bus. The start sequence (often referred as START or S-bit) is a high-to-low transition of the SDA line while the SCL line is high. This informs all the slave devices to start listening to the next incoming addressing information.

The Slave address transfer

The first byte after the START bit is the addressing information. This is a 7-bit address, followed by the R/W bit. The R/W bit informs the slave whether it is supposed to transmit or receive data. The corresponding slave needs to then respond with the **ACK** bit.

Transferring data

When the transmission has been agreed upon, the byte-by-byte data transmission takes place. A transferred byte is always acknowledged by the receiver with a 9th ACK bit. If the receiver doesn't acknowledge the byte, the master can generate a STOP event to abort or another START event to retransmit. In case the master does not acknowledge a transmission from the slave, the slave releases the SDA line for the master to decide whether the transmission will be retried.

Generating the STOP signal

When the transmission is deemed over by the master, it generates a STOP bit. This is a low-to-high transition of SDA while the SCL line is high.

The SPI bus

The **Serial Peripheral Interface (SPI)** bus is a peripheral bus that, like I2C, can connect multiple devices to each other, and has a one-master-to-multiple-slaves configuration.

This bus is somewhat more advanced than I2C, as it allows full-duplex operation, so that data can be read and transmitted at the same time. It can also operate at a significantly faster speed than I2C, as even a speed of 10 Mbps can be reached between some devices. It does, however, demand for more lines to operate upon, as it needs four wires to function with at least one slave device. Each additional device will need an additional line from the master that serves as slave (chip) select line for that particular device. The master cannot drive the bus in a "broadcast" mode; the specification states that it can only activate one slave at a time. The specified logic signals are:

Abbreviation	Meaning
SCLK	Serial CLock
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SS	Slave Select

Sometimes those lines are named slightly differently, but the idea behind them is always the same. There is a line for the clock that defines the transfer speed, separate lines for the transmit and receive operations, and a slave select line to select the receiver of the transmission. Since the data is sent on separate lines, and the addressing takes place with the slave select line, decoding the SPI transmissions is actually easier than it is on I2C, even though you have more lines to watch for.

In general, if you start integrating peripheral devices into your board, we cannot emphasize this enough: a simple USB logic analyzer is invaluable. For example, implementing the SPI transmission between your board and a microcontroller is surprisingly painstaking without the proper tools.



There are a lot of USB logic analyzers available on the Internet, and all of them are usually good enough to investigate any I2C or SPI issues that you might have during development (you can lower the bus speeds so that even cheaper logic analyzers will be able to decode the transmissions). When selecting a logic analyzer for home use you should select it mainly considering the software quality. We can recommend a product from Saleae, as the software they have created is simple to use and very powerful.

Considering the security aspects

Our Android application from *Chapter 6, Creating an Android Client*, can operate over public networks (public from the point of view of your home network). This means that basically anyone can connect to your socket, if the connection is currently not busy and the person knows your IP address and the socket port you decide to use. We have currently not implemented any type of security enforcement; we only drop the connection from the server side if we receive an unclear response to our Hello Message. However, since we do send the initial data packet in a clearly readable form, the person receiving the data can deduce that our server operates with a clear text protocol. If this is a malicious attempt, reverse engineering the protocol from this point onwards is not too difficult, if time and energy is put to the task.

Of course, all socket communications work on streams, and there are several types of security measures that you can take depending on the level of security you feel your connection requires.

You are basically only limited by your imagination and the type of security you can implement. Let me give you some ideas.

Making the client identify his intentions first

One of the simplest things that you could do is to reverse the initial connection procedure. Instead of sending a welcome message, the server would never respond to a new socket, but immediately wait for a valid input. You could easily implement a protocol-version-checking sequence, so that if this functionality is required, a client has to initiate it himself.

In addition to this, a connection would be silently dropped, if input that is not valid is detected. Then a temporary counter would be incremented from this particular IP address, and if, for example, three consecutive incorrect sequence initiations follow, a ban would be set for this IP address for a specified length of time. Of course, all these failed connection attempts need to be logged and perhaps an e-mail should be sent to the administrator about it.

Implementing the encrypted password login

Much stronger security is provided by requiring all clients to authenticate themselves when they connect. But since sending a password without encryption over the network is not a healthy habit, we will also require encryption. Valid login and password combinations will be held on a file on the server, and this way only server administrator can manage logins. During the initial connection establishment, a valid encrypted login-password sequence must be sent to the server, else again the server initiates its cold shoulder disconnection sequence. There are several ways to encrypt and decrypt data in Python, and we will choose 128-bit AES encryption with preshared keys, as that encryption is secure, and available in Android by default as well.

To use encryption on our Beagle, we will need to download a library for this purpose from the Internet. So, make sure you are connected (and you have done the automatic date configuration part from *Chapter 2, Input and Output*) and type in the following code:

```
root@beaglebone:~# pip install pycrypto
```

This will install the necessary libraries to enable encryption on our Python code in Beagle. Next, we should create a file that will hold the password data. For now, we will not encrypt it, but after reading this chapter, we're sure you will have a good idea how to encrypt this data, if you wish to do so. Create a file called `valid_passwords.txt`, and in here, add each user to a separate line in the `username:password` format:

```
root@beaglebone:~/ch7# cat valid_passwords.txt
jlumme:mypassword
reader:anotherpassword
```

Now, we are ready to start modifying the actual code.

Version 2.1 modifications to the server code

Let's start implementing the first two security features to `beagle_server.py` we have talked about so far. First, will need to import an AES cipher from the `Crypto` library:

```
from Crypto.Cipher import AES
```

And then we will need some constants and a lambda definition:

```
password_filename = "valid_passwords.txt"

# Encryption related definitions
BS = 16 #Our key size (16 bytes -> 128bit encryption)
SECRET_KEY="BeagleHomeAutoma" # Keep it 16 characters if you change

# The data needs to (un-)padded to the BS borders
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(s[-1])]
```

You can see here that we first defined a filename that will hold a login-password combination (we will leave it as a clear text file for now). Then we define the variable `BS`; this defines our block size for the encryption. You could use 16, 24, or 32 here for 128-, 192-, or 256-bit encryption respectively. We also defined our preshared secret encryption key `SECRET_KEY` that will be known to both the server and the client, and encryption and decryption will use this key. If you change it, keep in mind that you have to give it a proper block size, otherwise the crypto functions will fail to operate.

Then we defined two **lambda** functions, `pad` and `unpad`. Lambda functions are nameless functions that can be defined at any time and always return a value. Here we defined two functions that will automatically append or remove padding from an input if its size is not equal to the border defined by `BS`.

Now let's define a function that will read the password file and compare whether the supplied login is valid:

```
def compare_to_valid_logins(username, password):
    pw_list = open (password_filename, 'r').read().splitlines()

    for line in pw_list:
        uname = line[:line.index(":")]
        pword = line[line.index(":")+1:]

        # If username and password match, return 1
        if uname == username and pword == password:
            return 1
```

```
# If no successful login combination found
return 0
```

We read the password file line by line, and compared it with the function arguments to see if a matching combination is found. The login-password is supposed to be sent separated by the ":" character. Now we will need to implement the actual message decryption function. It is somewhat similar to a normal `handle_client_request()` function, but of course with some extras for handling the decryption:

```
def verify_password(cs):
    try:
        msg = cs.recv(4) # Retrieve the header, this is a blocking call
        header = unpack("!HH", msg) # Decode the mandatory headers

        # Verify that sender sent the PASSWORD header
        if header[0] == BP.MT_PASSWORD:

            # Check how much data is left in the stream
            remaining_size = int(header[1])
            print "Encrypted package is %d bytes" % remaining_size

            # Read the encrypted package
            encrypted_data = cs.recv(remaining_size)
```

So far everything is the same as when reading a message from a client. But then we need to decrypt the secret data:

```
iv = encrypted_data[:16]

# Initialize our deciphering key
key = AES.new(SECRET_KEY, AES.MODE_CBC, IV=iv)

# Rest of the packet is supposed to be login data
decrypted = key.decrypt(encrypted_data[16:])
login_details = unpad(decrypted) # Remove possible padding
```

You can see that the client has appended the **Initialization Vector (IV)** to the first 16 bytes of the message. An IV is used to initialize the cipher, and it can be chosen randomly (again, as long as it is 16 bytes). After that we call the cipher to create us a key for decryption with the `SECRET_KEY` and IV information. Lastly we decrypt the data, and remove the possible padding using the `unpad` lambda function.

After this we hold the login details in the `login_details` variable. We will verify them with the function that we defined before, as follows:

```
# Verify that the format is correct
if ":" in received_login:
    username = received_login[:received_login.index(":")]
    password = received_login[received_login.index(":")+1:]
    return compare_to_valid_logins(username, password)
else:
    return 0

# Any error in the procedure, we just disconnect the client
except Exception, e:
    print "Something went wrong:"
    print e
    return 0 # Drop the connection
```

Since it's mandatory for our client to authenticate itself, we also have to be sure that the authentication data is available. During startup, we have to check for the existence of the password file. In the `main` function, right in the very beginning, add the following code:

```
#Check that password file has been defined:
try:
    with open(password_filename):
        pass
except IOError:
    print "Password file [%s] not found, exiting" % password_filename
    sys.exit(1)
```

Lastly, we should add a call to the `verify_password()` method in our `main` function, right after the client connects to the server:

```
client = server_socket.wait_for_client(srv) # blocking!

result = verify_password(client)

if not result == 1:
    print "Something wrong, drop the socket"
    client.close()
    continue
```

Now the server will always expect an encrypted login from a newly connecting client. If something doesn't occur the way the server expects it to, it will just drop the connection. Next, let's add support for integrating this with our Android client.

Version 2.2 modifications to Android client code

We do not need too many modifications to `MainActivity.java` for now. All the encryption will be handled in the networking part of our code. On the UI side, we will only query the user for a username and password during the connection attempt and pass that information to our network thread. So, first we add a couple of new variables as follows:

```
private String username = "";
private String password = "";
```

To get the login details from the user, we will not modify the existing UI, but instead add a new login `Alert` popup that is shown when the user clicks on the **Connect** button. For this purpose, we need to create a new UI layout definition file called `password_query.xml`. We will create it by navigating to **File | New | New Android XML file** and add it under resources. To this file, add the following definition:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:id="@+id/login_username_textview"
        android:text="@string/login_username"
        android:textStyle="bold" />
    <EditText
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:id="@+id/login_username_edittext"
        android:inputType="text" />
    <TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:id="@+id/login_password_textview"
        android:text="@string/login_password"
        android:textStyle="bold" />
    <EditText
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:id="@+id/login_password_edittext"
        android:inputType="textPassword" />
</LinearLayout>
```

Now we can implement our `loginDialog` function as follows:

```
public AlertDialog loginDialog(Context c, String message) {
    //Inflate the login query window from resources
    LayoutInflater factory = LayoutInflater.from(c);
    final View textEntryView = factory.inflate(R.layout.password_query,
        null);
    //Set message and title, and button details
    AlertDialog.Builder alert = new AlertDialog.Builder(c);
    alert.setTitle("Login");
    alert.setMessage(message);
    alert.setView(textEntryView);
```

Here we retrieve the layout information from the resources, and start by creating a new alert dialog with the information supplied to the function. Next, we will add button handlers. First, we will add the "confirm" button as shown:

```
    alert.setPositiveButton("Login", \
        new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int whichButton) {
        final EditText usernameInput = (EditText)
            textEntryView.findViewById(R.id.login_username_edittext);
        final EditText passwordInput = (EditText)
            textEntryView.findViewById(R.id.login_password_edittext);

        //Get the text user entered
        username = usernameInput.getText().toString();
        password = passwordInput.getText().toString();

        new Thread(new Runnable() {
            public void run() {
                nt = new NetworkTask(SERVER_ADDR,
                    SERVER_PORT, handler, username + ":" + password);
                nt.run();
            }
        }).start();
    }
});
```

We add an `onClick` listener event for the login query window when the user clicks on the `PositiveButton`. It will fetch the data from the `EditText` fields and pass that information to `NetworkTask` (which we have moved here from the original `uiEventHandler` function). Next, we will add another `onClick` listener event for the `NegativeButton` as follows:

```
//If user chooses to cancel

alert.setNegativeButton("Cancel", new
DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, intwhichButton) {
        // Do nothing
    }
});
```

Now all that is left is to return the created dialog:

```
return alert.create();
```

Now we will modify the `uiEventHandler` function. We will remove the `NetworkTask` starting code from here, and instead add the following code to show the alert dialog that we just created:

```
public void uiEventHandler(View v) {
    //Connect button pressed, and we are not connected -> connect
    if (v == connectButton && !connectedToServer ) {
        AlertDialog a = loginDialog(this, "Login details:");
        a.show();
    }
    //Connect button pressed, and we're connected -> disconnect
    ...
}
```

Our UI can now handle user login prompts, and we can start thinking about the network thread implementation. For this purpose, we will need to define the following new constants in the `BeagleProtocol.java` file:

```
public final static short MT_PASSWORD      = 18;
public final static short LOGIN_FAILED     = 66;
public final static short ENCRYPTION_FAILED = 67;
```

Next, we need to add the piece of code that encrypts our login details and sends them to the server for validation in `NetworkTask.java`. First, we need to add the following new variables:

```
private String loginUsernamePassword = "";
//Encryption related
private IvParameterSpecivspec;
private SecretKeySpeckeyspec;
private Cipher cipher;
private String SecretKey = "BeagleHomeAutoma"; //16 bytes -> 128bit
    encryption
```

We also need to modify our constructor as follows:

```
NetworkTask(String address, int port, Handler handler, String login) {
    serverAddress = address;
    serverPort = port;
    parent = handler;
    loginUsernamePassword = login;
    alive = true; //Set to false to end the thread during disconnection
}
```

Add a new function called `sendEncryptedLoginDetails`. This function will be responsible for encryption, and send the encrypted data to the server. Since our server does not reply to invalid logins, all we can do is proceed as follows, and evaluate the connection state later:

```
public int sendEncryptedLoginDetails() {
    //First initialize the random IV, and retrieve AES key spec
    SecureRandom rnd = new SecureRandom();
    byte[] ivbytes = rnd.generateSeed(16);
    IvParameterSpec ivspec = new IvParameterSpec(ivbytes);
    KeySpec keyspec = new SecretKeySpec(SecretKey.getBytes(), "AES");

    byte[] encrypted = null;
```

Here we use the `SecureRandom` class to generate 16 random bytes for our IV. These bytes will be used as the "salt" for the encryption. Then we create the encryption key that is preshared between the client and server (`String SecretKey`). After this we initialize the cipher, and encrypt our message as follows:

```
//Define and initialize the cipher
try {
    cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, keyspec, ivspec);

    //Perform the encryption
    encrypted = cipher.doFinal(loginUsernamePassword.getBytes());
} catch (Exception enc) {
    debug("Encryption failed");
    enc.printStackTrace();
    return 1;
}
```

We first initialize the cipher by defining it with the desired specifics of the encryption, and then encrypt the `loginUsernamePassword` String. In the actual code, we separate the different try-catch blocks to identify which part of the code failed, so that the errors can be properly acted upon. But here, we have combined it with one catch-all call to save space.

Now that we have the encrypted bytes ready we can create our data package and send it, as shown in the following lines of code:

```
int enc_len = encrypted.length;
int iv_len = ivbytes.length;
byte[] pw = new byte[4 + iv_len + enc_len];

//Construct the message
pw[0] = (byte)((BeagleProtocol.MT_PASSWORD>> 8) & 0xff);
pw[1] = (byte)(BeagleProtocol.MT_PASSWORD&0xff);
pw[2] = (byte)((enc_len + iv_len>> 8) & 0xff);
pw[3] = (byte)(enc_len + iv_len& 0xff);

//Copy the IV to the outgoing packet
System.arraycopy(ivbytes,0,pw,4 ,iv_len);
//Copy the encrypted data to outgoing packet
System.arraycopy(encrypted, 0, pw, 4 + iv_len, enc_len);

//Send the data
try {
    os.write(pw);
} catch (IOException ioe) {
    debug("Sending password has failed");
    ioe.printStackTrace();
}

//Cleanup
rnd = null;
ivspec = null;
pw = null;

return 0; //Success
}
```

Now that our encrypted login function is complete, all that is left is to add a proper place to call it in the `run()` method.

This call is placed after we have initialized our socket and data streams, as follows:

```
os = new DataOutputStream(socket.getOutputStream());

//Send the password information to the server:
int res = sendEncryptedLoginDetails();
if (res != 0) {
    debug("Encryption failed...");
    //In reality, we cannot really do much here.
    //Something wrong with selected encryption settings
}
```



```
//Read the initial welcome message
byte[] readArray = new byte[6];
int howmany = is.read(readArray, 0, 6);
//If our login fails, server will just drop the socket
if (howmany == -1) {
    debug("Stream has closed. Something wrong with login ?");

    //Inform the UI about login failure
    Message msg = new Message();
    msg.what = BeagleProtocol.LOGIN_FAILED;
    msg.obj = null;

    //send the message
    parent.handleMessage(msg);

    is.close();
    os.close();
    socket.close();
    return;
}
```

After we initialize our streams, we call the `sendEncryptedLoginDetails` function to identify us with the server along with the details we provided in the previous login query. After that, all we can do is check if the server is still sending us information (try to read the protocol version). If the socket has been dropped (the `read` stream returns `-1`), it means that the server has prevented our login, and we have to handle it. We inform the UI via our `Handler` class, close the streams, and return from this function, thus ending the life of this thread.

With the previous code, you are now securely transmitting your login details over the network.

Encrypting all of the communication

In the previous section, you saw how you can encrypt one message. The next step in beefing up the security would be to encrypt all of the communication. With the tips you saw earlier, this wouldn't be such a big hurdle; you will just have to modify the normal message sequence a bit. You could change the places of the "message request" and "message size" headers, so that only the incoming message size would be unencrypted, and everything else would always be encrypted.

The next step would be to also implement secure key exchange, so that the encryption would always be done with different keys. This will be getting into the advanced areas of cryptography and TCP/IP security, so we will leave it for a topic in another book.

The GPIO mapping of the P8 and P9 headers

The following screenshot contains the complete GPIO mapping for the headers P8 and P9. The pins that are not normally usable for some reason (that is, when mapped to eMMC or HDMI), are marked in red:

Pin muxing in header P8									
PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE4	MODE6	MODE7
1					GND				
2					GND				
3	R9	GPIO1_6	gpmc_ad6	mmc1_dat6					gpio1[6]
4	T9	GPIO1_7	gpmc_ad7	mmc1_dat7					gpio1[7]
5	R8	GPIO1_2	gpmc_ad2	mmc1_dat2					gpio1[2]
6	T8	GPIO1_3	gpmc_ad3	mmc1_dat3					gpio1[3]
7	R7	TIMER4	gpmc_advn_ale		timer4				gpio2[2]
8	T7	TIMER7	gpmc_oen_ren		timer7				gpio2[3]
9	T6	TIMER5	gpmc_be0n_cle		timer5				gpio2[5]
10	U6	TIMER6	gpmc_wen		timer6				gpio2[4]
11	R12	GPIO1_13	gpmc_ad13	lcd_data18	mmc1_dat5*	mmc2_dat1	eQEP2B_in		gpio1[13]
12	T12	GPIO1_12	gpmc_ad12	lcd_data19	mmc1_dat4*	mmc2_dat0	EQEP2A_IN		gpio1[12]
13	T10	EHRPWM2B	gpmc_ad9	lcd_data22	mmc1_dat1*	mmc2_dat5	ehrpwm2B		gpio0[23]
14	T11	GPIO0_26	gpmc_ad10	lcd_data21	mmc1_dat2*	mmc2_dat6	ehrpwm2_tripzone		gpio0[26]
15	U13	GPIO1_15	gpmc_ad15	lcd_data16	mmc1_dat7*	mmc2_dat3	eQEP2_strobe		gpio1[15]
16	V13	GPIO1_14	gpmc_ad14	lcd_data17	mmc1_dat6*	mmc2_dat2	eQEP2_index		gpio1[14]
17	U12	GPIO0_27	gpmc_ad11	lcd_data20	mmc1_dat3*	mmc2_dat7	ehrpwm0_synco		gpio0[27]
18	V12	GPIO2_1	gpmc_clk_mux0	lcd_memory_clk	gpmc_wait1	mmc2_clk		mcasp0_fsr	gpio2[1]
19	U10	EHRPWM2A	gpmc_ad8	lcd_data23	mmc1_dat0*	mmc2_dat4	ehrpwm2A		gpio0[22]
20	V9	GPIO1_31	gpmc_csn2	gpmc_be1n	mmc1_cmd*				gpio1[31]
21	U9	GPIO1_30	gpmc_csn1	gpmc_clk	mmc1_clk*				gpio1[30]
22	V8	GPIO1_5	gpmc_ad5	mmc1_dat5					gpio1[5]
23	U8	GPIO1_4	gpmc_ad4	mmc1_dat4					gpio1[4]
24	V7	GPIO1_1	gpmc_ad1	mmc1_dat1					gpio1[1]
25	U7	GPIO1_0	gpmc_ad0	mmc1_dat0					gpio1[0]
26	V6	GPIO1_29	gpmc_csn0						gpio1[29]
27	U5	GPIO2_22	lcd_vsync*	gpmc_a8					gpio2[22]
28	V5	GPIO2_24	lcd_pclk*	gpmc_a10					gpio2[24]
29	R5	GPIO2_23	lcd_hsync*	gpmc_a9					gpio2[23]
30	R6	GPIO2_25	lcd_ac_bias_en*	gpmc_a11					gpio2[25]
31	V4	UART5_CTSN	lcd_data14*	gpmc_a18	eQEP1_index	mcasp0_axr1	uart5_rxd	uart5_ctsn	gpio0[10]
32	T5	UART5_RTSN	lcd_data15*	gpmc_a19	eQEP1_strobe	mcasp0_ahclkx	mcasp0_axr3	uart5_rtsn	gpio0[11]
33	V3	UART4_RTSN	lcd_data13*	gpmc_a17	eQEP1B_in	mcasp0_fsr	mcasp0_axr3	uart4_rtsn	gpio0[9]
34	U4	UART3_RTSN	lcd_data11*	gpmc_a15	ehrpwm1B	mcasp0_ahclkr	mcasp0_axr2	uart3_rtsn	gpio2[17]
35	V2	UART4_CTSN	lcd_data12*	gpmc_a16	eQEP1A_in	mcasp0_aclkr	mcasp0_axr2	uart4_ctsn	gpio0[8]
36	U3	UART3_CTSN	lcd_data10*	gpmc_a14	ehrpwm1A	mcasp0_axr0		uart3_ctsn	gpio2[16]
37	U1	UART5_TXD	lcd_data8*	gpmc_a12	ehrpwm1_tripzone	mcasp0_aclkx	uart5_txd	uart2_ctsn	gpio2[14]
38	U2	UART5_RXD	lcd_data9*	gpmc_a13	ehrpwm0_synco	mcasp0_fsx	uart5_rxd	uart2_rtsn	gpio2[15]
39	T3	GPIO2_12	lcd_data6*	gpmc_a6		eQEP2_index			gpio2[12]
40	T4	GPIO2_13	lcd_data7*	gpmc_a7		eQEP2_strobe	pr1_edio_data_out7		gpio2[13]
41	T1	GPIO2_10	lcd_data4*	gpmc_a4		eQEP2A_in			gpio2[10]
42	T2	GPIO2_11	lcd_data5*	gpmc_a5		eQEP2B_in			gpio2[11]
43	R3	GPIO2_8	lcd_data2*	gpmc_a2		ehrpwm2_tripzone			gpio2[8]
44	R4	GPIO2_9	lcd_data3*	gpmc_a3		ehrpwm0_synco			gpio2[9]
45	R1	GPIO2_6	lcd_data0*	gpmc_a0		ehrpwm2A			gpio2[6]
46	R2	GPIO2_7	lcd_data1*	gpmc_a1		ehrpwm2B			gpio2[7]

Pin muxing in header P9								
PIN	PROC	NAME	MODE0	MODE2	MODE3	MODE4	MODE6	MODE7
1					GND			
2					GND			
3					DC_3.3V			
4					DC_3.3V			
5					VDD_5V			
6					VDD_5V			
7					SYS_5V			
8					SYS_5V			
9					PWR_BUTTON			
10	A10				RESET_OUT			
11	T17	gpmc_wait0	mii2_crs	gpmc_csn4	mii2_crs_dv	mmc1_sdc	uart4_rxd_mux2	gpio0[30]
12	U18	gpmc_be1n	mii2_col	gpmc_csn6	mmc2_dat3	gpmc_dir	mcasp0_aclkr_mux3	gpio1[28]
13	U17	gpmc_wpn	mii2_rxerr	gpmc_csn5	mii2_rxerr	mmc2_sdc	uart4_txd_mux2	gpio0[31]
14	U14	gpmc_a2	mii2_txd3	rgmii2_td3	mmc2_dat1	gpmc_a18	ehrpwm1A_mux1	gpio1[18]
15	R13	gpmc_a0	gmii2_txen	mii2_tctl	mii2_txen	gpmc_a16	ehrpwm1_tripzone	gpio1[16]
16	T14	gpmc_a3	mii2_txd2	rgmii2_td2	mmc2_dat2	gpmc_a19	ehrpwm1B_mux1	gpio1[19]
17	A16	spi0_cs0	mmc2_sdwp	I2C1_SCL	ehrpwm0_synci			gpio0[5]
18	B16	spi0_d1	mmc1_sdwp	I2C1_SDA	ehrpwm0_tripzone			gpio0[4]
19	D17	uart1_rtsn	timer5	dcan0_rx	I2C2_SCL	spi1_cs1		gpio0[13]
20	D18	uart1_ctsn	timer6	dcan0_tx	I2C2_SDA	spi1_cs0		gpio0[12]
21	B17	spi0_d0	uart2_txd	I2C2_SCL	ehrpwm0B		EMU3_mux1	gpio0[3]
22	A17	spi0_sclk	uart2_rxd	I2C2_SDA	ehrpwm0A		EMU2_mux1	gpio0[2]
23	V14	gpmc_a1	gmii2_rxdv	rgmii2_rxdv	mmc2_dat0	gpmc_a17	ehrpwm0_synco	gpio1[17]
24	D15	uart1_txd	mmc2_sdwp	dcan1_rx	I2C1_SCL			gpio0[15]
25	A14	mcasp0_ahclkx	eQEP0_strobe	mcasp0_axr3	mcasp1_axr1	EMU4_mux2		gpio3[21]
26	D16	uart1_rxd	mmc1_sdwp	dcan1_tx	I2C1_SDA			gpio0[14]
27	C13	mcasp0_fsr	eQEP0B_in	mcasp0_axr3	mcasp1_fsx	EMU2_mux2		gpio3[19]
28	C12	mcasp0_ahclk	ehrpwm0_synci	mcasp0_axr2	spi1_cs0	eCAP2_in_PWM2_out		gpio3[17]
29	B13	mcasp0_fsx	ehrpwm0B		spi1_d0	mmc1_sdc_mux1		gpio3[15]
30	D12	mcasp0_axr0	ehrpwm0_tripzone		spi1_d1	mmc2_sdc_mux1		gpio3[16]
31	A13	mcasp0_aclkx	ehrpwm0A		spi1_sclk	mmc0_sdc_mux1		gpio3[14]
32					VADC			
33	C8				AIN4			
34					AGND			
35	A8				AIN6			
36	B8				AIN5			
37	B7				AIN2			
38	A7				AIN3			
39	B6				AIN0			
40	C7				AIN1			
41A	D14	xdma_event_intr1		tcclk	clkout2	timer7_mux1	EMU3_mux0	gpio0[20]
41B	D13	mcasp0_axr1	eQEP0_index		Mcasp1_axr0	emu3		gpio3[20]
42A	C18	eCAP0_in_PWM0_out	uart3_txd	spi1_cs1	pr1_eca0_eca0_pcapin_apwm0	spi1_sclk	xdma_event_intr2	gpio0[7]
42B	B12	Mcasp0_aclkr	eQEP0A_in	Mcasp0_axr2	Mcasp1_aclkx			gpio3[18]
43					GND			
44					GND			
45					GND			
46					GND			

Index

Symbols

7-Zip
URL 95

A

Adafruit-BBIO 28
Analog-to-Digital converter (ADC) 8, 40
Android client code
 modifying 153-158
Android Development Tools (ADT) 101
Android project
 emulator, creating 104-106
 setting up 101-104
Angstrom 9
application
 using, from outside home network 139, 140
application permissions 114

B

BeagleBone
 hardware requisites 7, 8
 Hello World program, running on 18, 19
 software requisites 8
BeagleBone Black 21
BeagleBone HD camera cape
 about 94
 boot media, changing 94-97
 cameras, controlling with Python 97-100
Beagle protocol
 defining 62, 63
bipolar junction transistors (BJTs) 71

bit shifting 126
boot media
 changing 94-97
boot time kernel traces 144

C

cameras
 controlling, with Python 97-100
capex
 about 93
 URL 94
Change IP
 URL 139
client code 67-71
client side 81-83
client socket
 about 42
 and server socket, difference between 42
client view
 identifying 148, 149
communication
 encrypting 158
Connect button 153
create_data_packet() function 67, 84, 137

D

data transfer 147
DC Current gain (hFE) 72
debug function 117
DNSdynamic
 URL 139
d property 16

E

echo client 48-53
echo server 45-48
EHRPWM2B 22
emulator
 creating 104-106
encrypted password login
 Android client code, modifying 153-158
 implementing 149
 server code, modifying 150-152
environmental sensors
 about 55
 light sensor 56-59
 temperature sensor 60, 61
external hardware input 34, 36
external output 31, 33

F

field effect transistors (FETs) 71
FileOpen function 138
file permissions 16, 17
filesystem operation 15
findViewById function 118
FreeDNS
 URL 139

G

General-Purpose Input/Output pins. *See* **GPIO**
General Purpose Memory Controller (GPMC) 22, 94
gethostbyname() 43
get_image() function 100
gmtime() function 86
GPIO
 about 22-24
 external hardware input 34, 36
 external output 31, 33
 on-board LEDs 24-28
 Pulse width modulator 37-40
GPIO library
 for Python 28
GPIO mapping
 of P8 header 159
 P9 header 159

Graphical User Interface (GUI) 14

H

handle_client_request() function 137, 151
handle_client_requests method 65
hardware extensions
 about 93
 BeagleBone HD camera cape 94
hardware interfaces 21, 22
hardware requisites, BeagleBone 7, 8
Hello World program
 running, on BeagleBone 18, 19
host machine
 Linux 10
 Mac OS X 10
 preparing 9
 Windows 10

I

I2C bus
 about 145
 data transfer 147
 Slave address transfer 146
 start signal, generating 146
 STOP signal, generating 147
IC (integrated circuit) 60
IDE (Integrated Development Environment) 26
ifconfig eth0 command 14
Initialization Vector (IV) 151
initialize_server function 46
Inter-Integrated Circuit. *See* **I2C bus**
Internet sockets 43

J

JTAG debugging 145

L

Light Dependent Resistors (LDR) 56
light sensor 56-59
Linux
 about 10
 file permissions 16, 17
 filesystem operation 15

- operating, from console 14
- load framework**
 - implementing 77-80
- login_details variable** 152
- loginDialog function** 154
- ls command** 16

M

- Mac OS X** 10
- main method** 137
- Manager button** 104
- movement-detection alarm system** 88-93

N

- nano** 17
- Network Address Translation (NAT)** 139
- network thread** 123-132
- Network Time Protocol (NTP)** 29
- No-IP**
 - URL 139

O

- on-board LEDs** 24-28
- onCreate method** 117

P

- P8 header**
 - GPIO mapping, using for 159
- P9 header**
 - GPIO mapping, using for 159
- pack function** 67
- Parcel variable** 127
- passive infrared (PIR)** 88
- periodic tasks**
 - on server 85, 87
- permanent settings**
 - changing 81
 - retrieving 81
- print_help() function** 81
- Process.start() function** 87
- proper time**
 - setting 29, 30
- pull down resistor** 34
- pull-up resistor** 34

- Pulse width modulator** 37-40
- Pulse Width Modulators (PWMs)** 8
- PuTTY Download Page**
 - URL 12
- pygame**
 - URL 98
- Python**
 - cameras, controlling with 97-100
 - GPIO library, using for 28

R

- recv() function** 48
- reload_configuration() function** 86
- Rev A5A** 21
- r property** 16
- run() method** 122, 125, 157

S

- save framework**
 - implementing 77-80
- scripting** 14
- SecureRandom class** 156
- security aspects**
 - considering 148
- sendEncryptedLoginDetails function** 158
- send_message() function** 68, 81
- Serial Peripheral Interface.** *See* SPI bus
- server code**
 - about 64-67
 - modifying 150-152
- server features** 132-138
- server side** 83, 84
- server socket**
 - about 42
 - and client socket, difference between 42
- Settings button** 132
- shebang** 19
- Slave address transfer** 146
- socket**
 - about 42
 - client socket 42
 - server socket 42
- socket.accept() function** 46
- socket application**
 - example 43, 44

socket client

- application permissions 114
- network thread 123-132
- on Android 107
- support classes 114-116
- UI 116-123
- UI components, defining 107-113

socket documentation

- URL 43

software requisites, BeagleBone 8, 9

SPI bus 147, 148

start signal

- generating 146

STOP signal

- generating 147

strftime() function 86

support classes 114-116

symbolic link 29

system

- logging in 12-14

T

take_picture function 100

target board

- starting 11

temperature sensor 60, 61

temperature sensor LM60

- URL 60

transistors 71-74

U

UI 116-123

uiEventHandler function 155

Universal Asynchronous Receivers/ Transmitters (UARTs) 8

unpack method 65

update_configuration() function 84

USB ID

- URL 103

V

verify_password() method 152

W

wait_for_client function 47

Windows 10

Windows package

- URL 49

WinSCP

- URL 26

w property 16

X

x property 16

xz command 96



Thank you for buying BeagleBone Home Automation

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

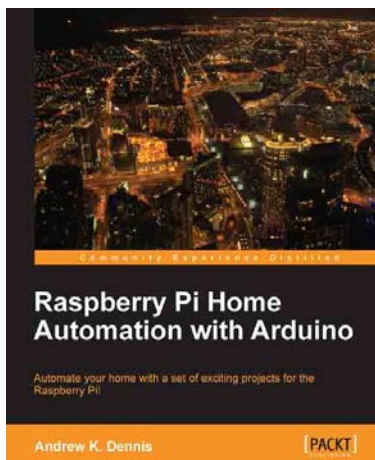
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



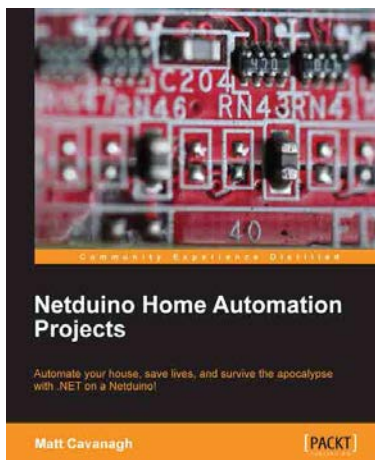
Raspberry Pi Home Automation with Arduino

ISBN: 978-1-84969-586-2

Paperback: 176 pages

Automate your home with a set of exciting projects for the Raspberry Pi

1. Learn how to dynamically adjust your living environment with detailed step-by-step examples
2. Discover how you can utilize the combined power of the Raspberry Pi and Arduino for your own projects
3. Revolutionize the way you interact with your home on a daily basis



Netduino Home Automation Projects

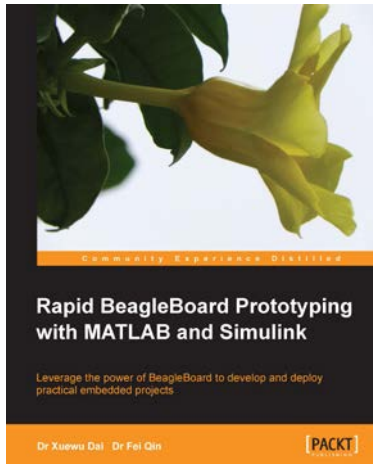
ISBN: 978-1-84969-782-8

Paperback: 108 pages

Automate your house, save lives, and survive the apocalypse with NET on a Netduino!

1. Automate your house using a Netduino and a bunch of common components
2. Learn the fundamentals of Netduino to implement them in almost any project
3. Create cool projects ranging from self-watering plants to a homemade breathalyzer

Please check www.PacktPub.com for information on our titles

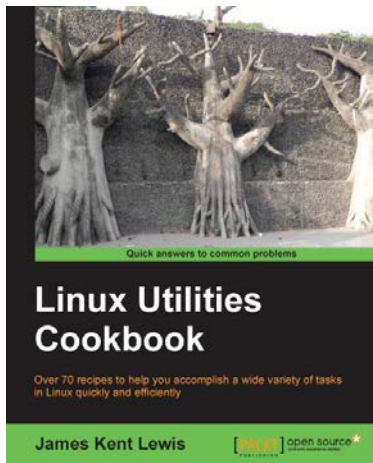


Rapid BeagleBoard Prototyping with MATLAB and Simulink

ISBN: 9781849696043 Paperback: 152 pages

Leverage the power of BeagleBoard to develop and deploy practical embedded projects

1. Develop and validate your own embedded audio/video applications rapidly with BeagleBoard
2. Create embedded Linux applications on a pure Windows PC
3. Full of illustrations, diagrams, and tips for rapid BeagleBoard prototyping with clear, step-by-step instructions and hands-on examples



Linux Utilities Cookbook

ISBN: 978-1-78216-300-8 Paperback: 224 pages

Over 70 recipes to help you accomplish a wide variety of tasks in Linux quickly and efficiently

1. Use the command line like a pro
2. Pick a suitable desktop environment
3. Learn to use files and directories efficiently

Please check www.PacktPub.com for information on our titles