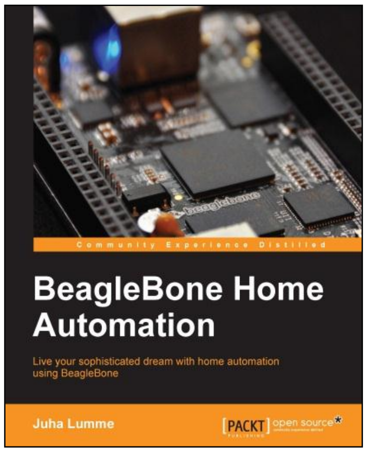
BeagleBone Home Automation

Juha Lumme



Chapter No. 4

"Extending Server Capabilities"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Extending Server Capabilities"

A synopsis of the book’s content

Information on where to buy this book

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.

BeagleBone Home Automation

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.

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.

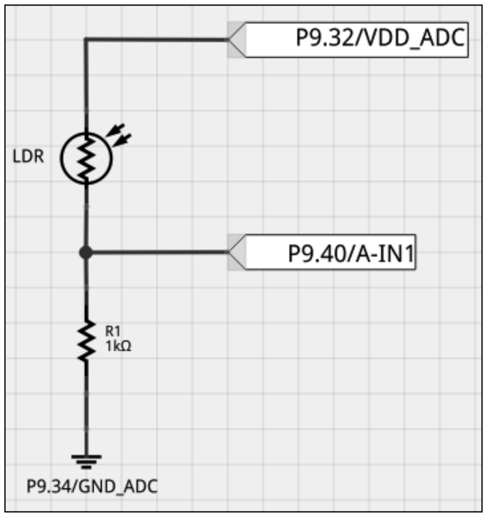
The resistance values will vary depending on the specification, and you should refer to the specification sheet of the resistor you have; however, in general, you can assume that we are talking about a range of couple hundreds to 1000 Ohm when the light is on, and in the magnitudes of 10 KOhm when the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

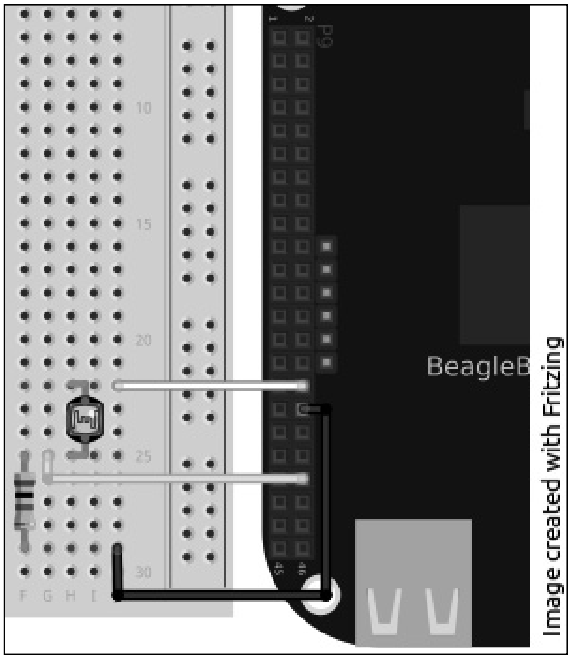
If you need time-critical responsiveness (let's say, for example, reacting to a laser pointer), you should look into photo diodes. They can have response times that are measured in nanoseconds.

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!

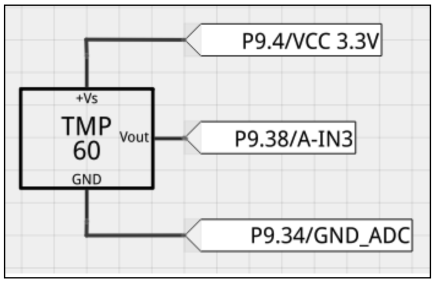
Now that we can measure light, let's look into measuring the current temperature.

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.

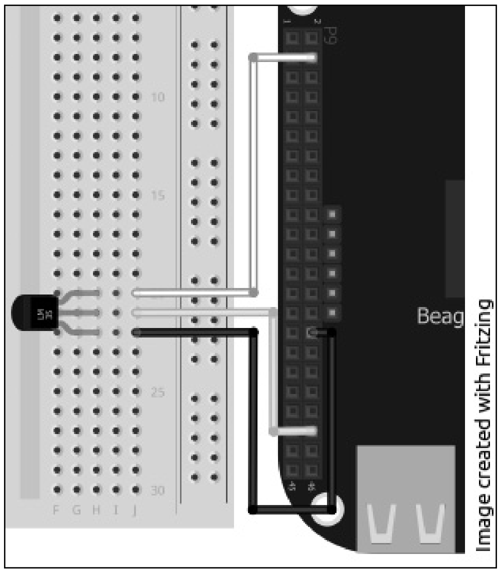
From the data sheet, we can see that the sensor works on voltage between 2.7 V and 10 V. So we can use our 3.3 V in P9.4 as a power source for it. The same sheet also tells us that the maximum output of this part is ~1200 mV, so we can safely use it with our analog input pins. Let's use **A-IN3** in **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: *Vout = (6.25 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.

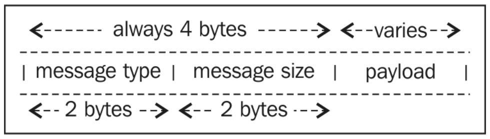
When maximum efficiency is required from a protocol, it is desirable to minimize different overheads and, for example, use bit masking to convey data. You can take a look at how, for example, TCP/IP protocol is implemented at the protocol level to learn further.

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:
      p  rint "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)
```

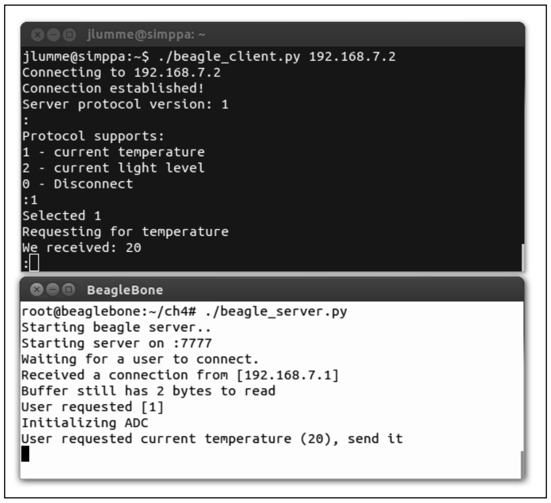
Once the message has been sent, we start expecting a reply from the server. The function that reads reply will currently support only short replies (two-byte payload). We will extend it later to also support larger payloads.

```
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:



```
jlumme@simppa:~$ ./beagle_client.py 192.168.7.2
Connecting to 192.168.7.2
Connection established!
Server protocol version: 1
:
Protocol supports:
1 - current temperature
2 - current light level
0 - Disconnect
:1
Selected 1
Requesting for temperature
We received: 20
:
```

```
root@beaglebone:~/ch4# ./beagle_server.py
Starting beagle server..
Starting server on :7777
Waiting for a user to connect.
Received a connection from [192.168.7.1]
Buffer still has 2 bytes to read
User requested [1]
Initializing ADC
User requested current temperature (20), send it
```

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** (**FET**s) and **bipolar junction transistors** (**BJT**s). 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:

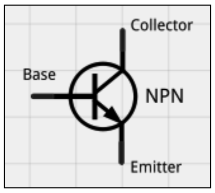
- The **base** is the lead that activates the transistor
- The **collector** is the positive lead
- The **emitter** is the negative lead

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:

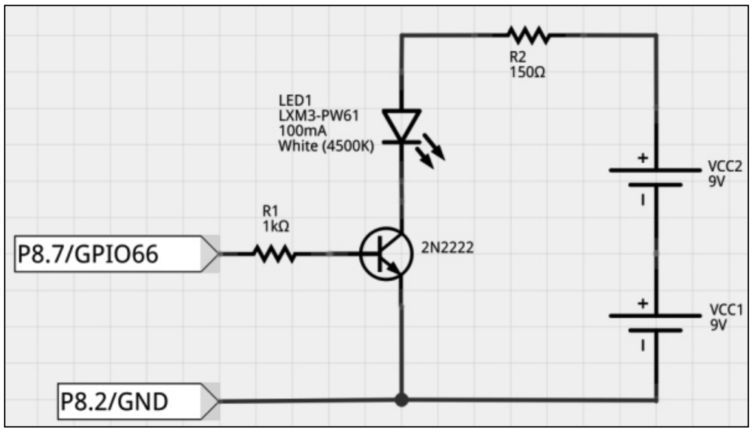
$$Igpio = (3.3V-0.6V)/1000$$
$$Igpio = 2.7mA$$

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

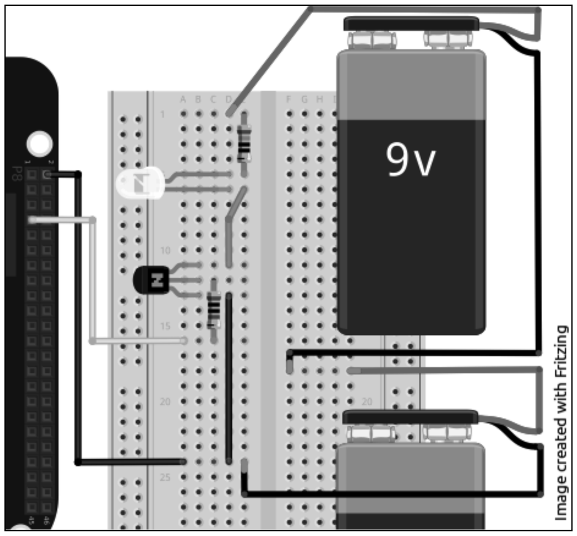
2.7 mA is nicely below the 4 mA our GPIO line can handle. We will be using a fairly common **2N2222** transistor, which has a gain (**hFE**) of 50, when the current is at least 1 mA. We can get maximum current at these levels from the following equation:

$$Inpn= 2.7mA * 50 = 135mA$$

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:

$$P = I * U$$
$$P = 0.1A * (18V - 3V)$$
$$P = 1.5W$$

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.