Sheffield Hallam University

Department of Engineering

BEng (Hons) Computer Systems Engineering

Activity ID		Activity Title			Laboratory Room No.	Level
Lab 102		RESTful Arduino web services			4302	6
Semester	**Duration [hrs]**	**Group Size**	**Max Total Students**	**Date of approval/review**	**Lead Academic**	
1	6	2	25	09-19	Alex Shenfield	

Equipment (per student/group)

Number	Item
1	Arduino kit
1	Ethernet shield

Learning Outcomes

	Learning Outcome
1	Understand the benefits, challenges and pitfalls of developing internet enabled embedded systems.
3	Implement multi-component distributed embedded systems using a flexible network architecture.

Developing RESTful Arduino web services

Introduction

Computing is about more than the PC on your desktop! Embedded devices are everywhere – from wireless telecommunications infrastructure points to electronic point of sale terminals. One definition of an embedded system is:

> "An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints."
>
> (http://en.wikipedia.org/wiki/Embedded_system)

In this series of labs you are going to be introduced to the open source Arduino platform – a cheap, simple to program, well documented prototyping platform for designing electronic systems. The heart of this prototyping system is the Arduino microcontroller board itself which is based on the commonly used ATMega328 chip.

The purpose of this lab session is to demonstrate some of the network functionality we can add to the Arduino platform using the Ethernet shields, before showing how we can run RESTful web services on the Arduino.

Bibliography

There are no essential bibliographic resources for this laboratory session aside from this tutorial sheet. However the following websites and tutorials may be of help (especially if you haven't done much electronics previously or your digital logic and/or programming is a bit rusty):

- http://www.arduino.cc/
- http://www.ladyada.net/learn/arduino/index.html
- http://tronixstuff.wordpress.com/tutorials/

Methodology

Check that you have all the necessary equipment (see Figure 1)!

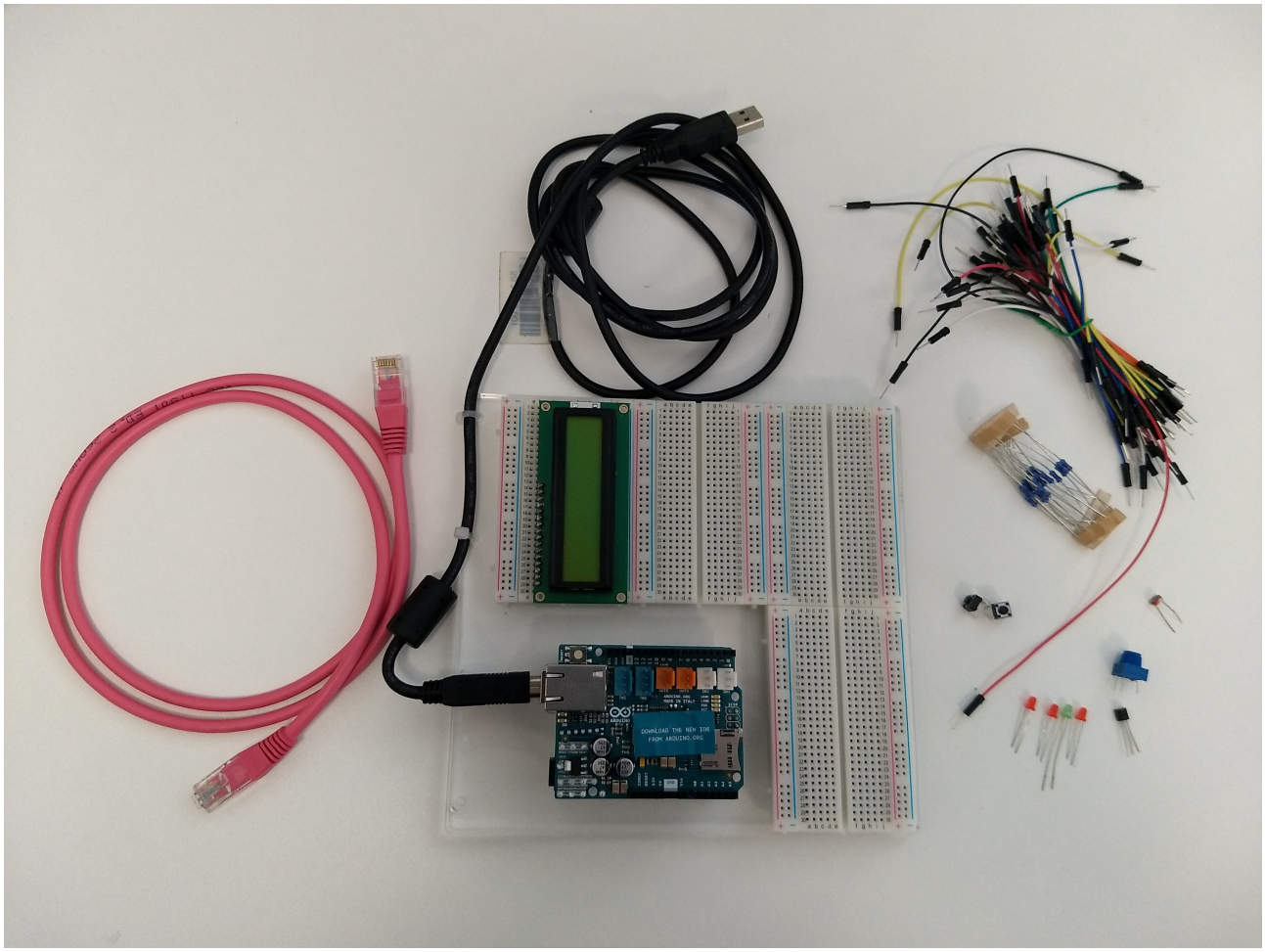

Figure 1 – The necessary equipment for this lab

Task 1

The Arduino Ethernet shield is capable of turning the Arduino into both a web client (like a browser) or a web server. The specific Ethernet shield we will be using in class also has a micro SD slot, making it capable of hosting full websites with minimal power requirements (though we will not be using this functionality in this lab).

As of version 1.0 of the Arduino IDE, the Ethernet library is capable of using DHCP to dynamically allocate IP addresses – earlier versions of the library required you to manually enter the IP address, gateway server and subnet mask to use.

The first task in this lab is to set the Arduino up as a simple web server. Connect the Arduino and Ethernet shield up as in Figure 2 below. Note here that we have the Ethernet cable plugged into the second network port on the benches and the USB cable attached to the computer.

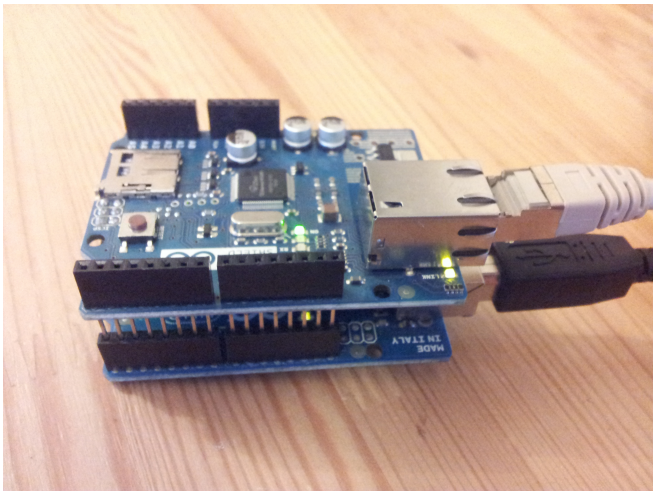

Figure 2 – Arduino and Ethernet shield

We are now going to write a simple program to make the Arduino act like a web server. The code is provided in code listing 1 (below) and goes through the following steps:

1. Sets up an Ethernet connection to the Arduino
2. Listens for data being sent from a client
3. Displays this data on the serial monitor
4. Sends the standard HTTP response header and closes the connection when there is no more data being sent

Code listing 1: A simple web server

```
/*
 * simple_web_server.ino
 *
 * a simple web server for demonstrating arduino and ethernet
 * connectivity
 *
 * author:  alex shenfield
 * date:    10/09/2018
 */

// note: the ethernet shields we have here at SHU use the
// Ethernet2.h library - however, many of the other shields on
// the market use the original Ethernet.h library!
#include <Ethernet2.h>
#include <SPI.h>

// set up a server on port 80 (note: web browsers usually
// assume that the server is running on port 80 unless told
// otherwise)
EthernetServer server = EthernetServer(80);

// the ethernet shields and ethernet libraries support DHCP as long as
// you are plugged into a device with a DHCP server however, if you
// are plugged straight in to the network switch in the cabinet (as
// in the embedded lab) you will need to allocate a static ip address
IPAddress ip(192, 168, 0, 11);
IPAddress gateway(192, 168, 0, 254);
IPAddress subnet(255, 255, 255, 0);

// ethernet shield mac address (i.e. the hexadecimal numbers found
// on the bottom of the shield)
byte mac[] = {0x90, 0xA2, 0xDA, 0x0F, 0xF5, 0xAD};

// CODE

// set up code
void setup()
{
  // set up serial comms for debugging and display of
  // DHCP allocated IP address
  Serial.begin(9600);

  // start the ethernet shield comms - initially try to get a DHCP ip
  // address
  if (Ethernet.begin(mac) == 0)
  {
    // if DHCP fails, allocate a static ip address
    Serial.println("failed to configure ethernet using DHCP");
    Ethernet.begin(mac, ip);
  }

  // start the server, and print the IP address to the serial
  // monitor
  server.begin();
  Serial.println(Ethernet.localIP());
}
```

```
// main code
void loop()
{
  // check for a client connection
  EthernetClient client = server.available();

  // while the client is still connected
  while (client)
  {
    // and has more data to send
    if (client.available() > 0)
    {
      // read bytes from the incoming client and write them to
      // the serial monitor
      Serial.print((char)client.read());
    }
    // when the client is done sending data
    else
    {
      // send standard http response header (to acknowledge the
      // data)
      client.println("HTTP/1.1 200 OK");
      client.println("Content-Type: text/html");
      client.println();

      // disconnect from client
      client.stop();
    }
  }
}
```

Before you go any further, make a note of the MAC address on the bottom of your Ethernet shield (see Figure 3).

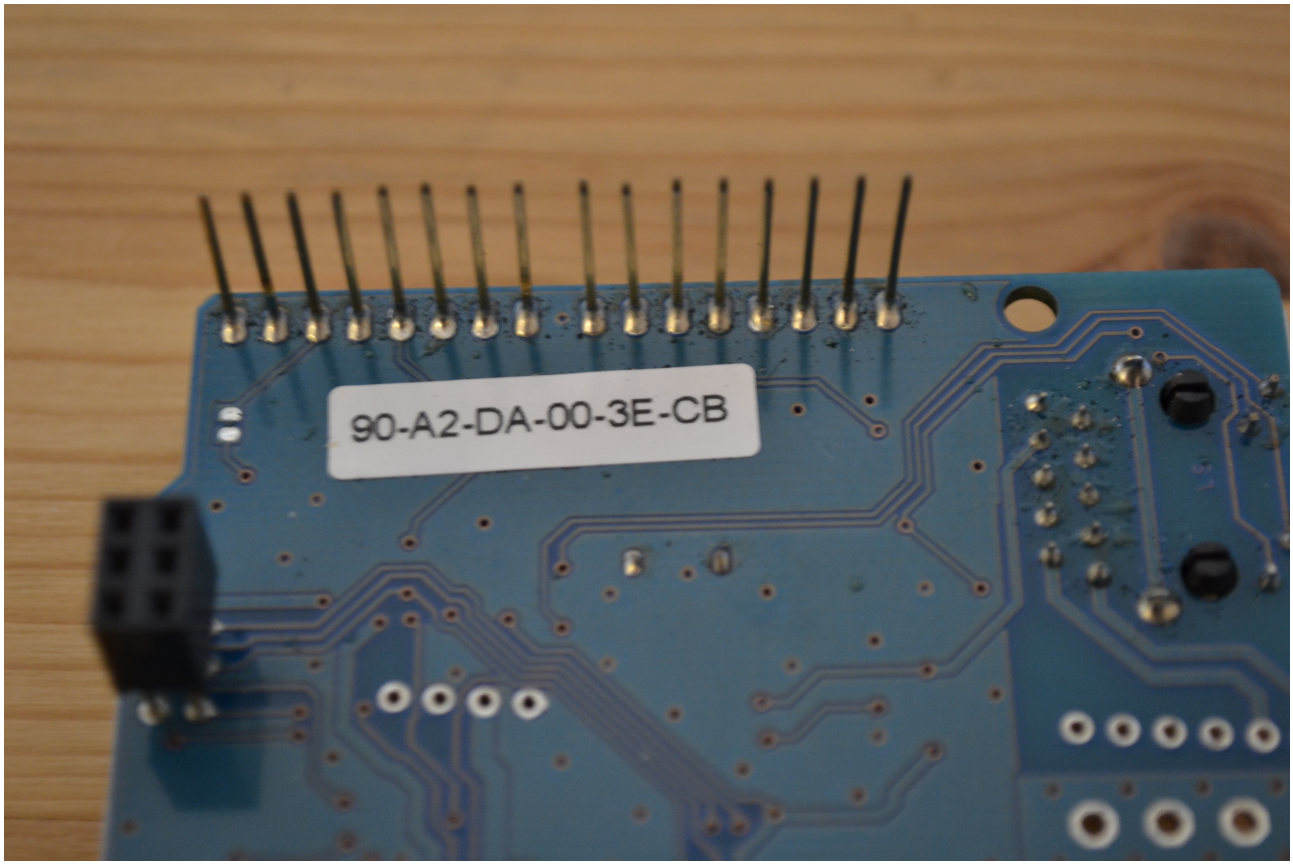

Figure 3 – Ethernet shield MAC address

The MAC address of your Ethernet shield is:

You will need to change this in the code for your webserver.

To set the fall-back static IP address of the sensor system, we will use 192.168.0.xx as the base IP address and the number on the network port (see highlighted port in Figure 5) as the number after the last decimal point. In this case, we would use the IP address 192.168.0.11.

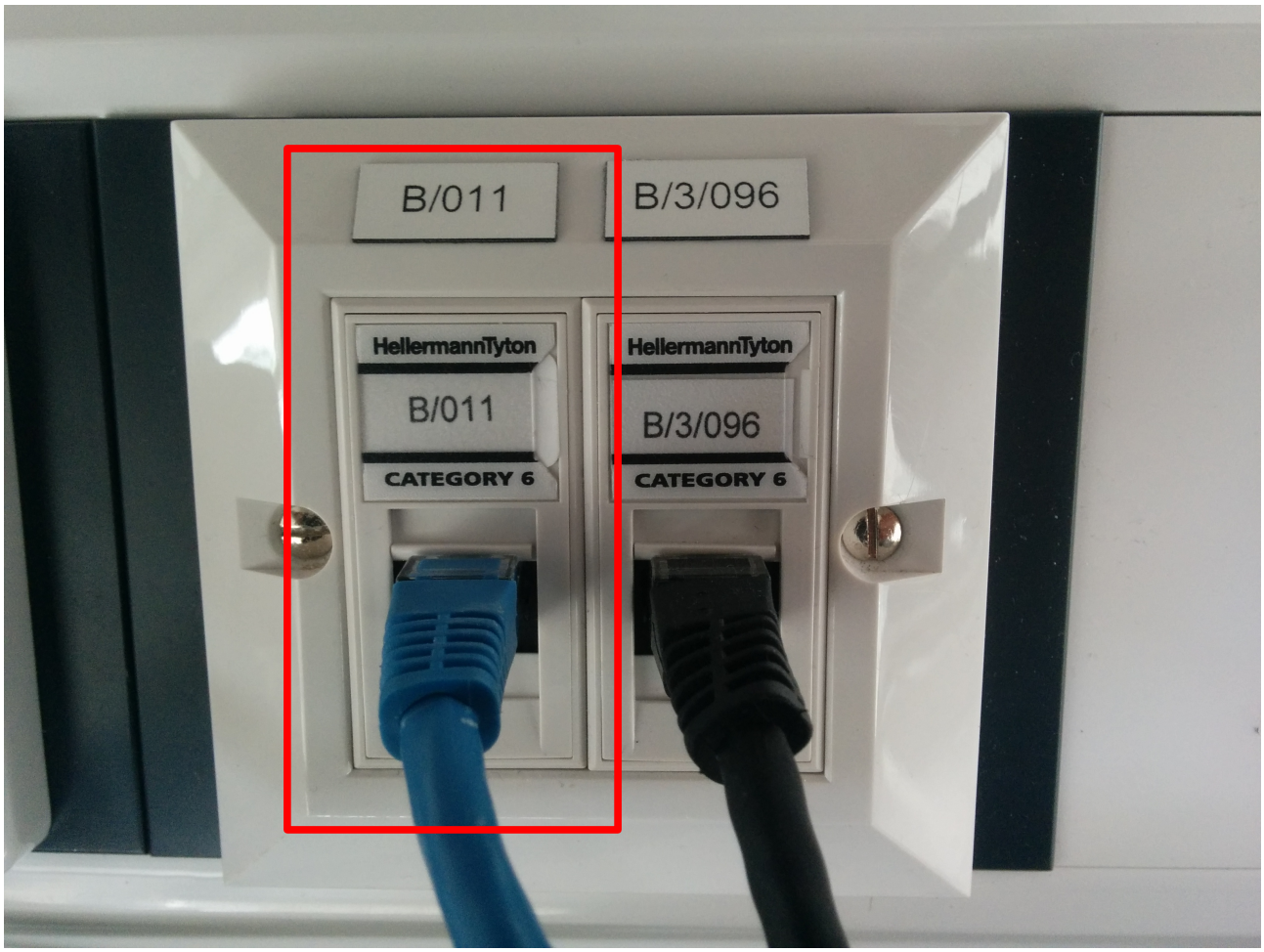

Figure 5 – Network port address

We then need to change this in our Arduino program (as in code snippet 2, below). Change the first highlighted number to the number you wrote down above and the second to the same number but with 254 at the end instead of whatever else was there.

Code snippet 2: Setting a static fall-back IP address

```
// ip address, gateway and network mask for the webserver
IPAddress ip(192, 168, 0, 11);
IPAddress gateway(192, 168, 0, 254);
IPAddress subnet(255, 255, 255, 0);
```

We can then upload our code and open the Arduino IDE serial monitor window. This should show us the IP address of our networked Arduino (whether the fall-back static IP address or the IP address that was allocated via DHCP), as in Figure 6, below.

Figure 6 – The serial monitor display the Arduino web server's IP address

Now, using your favourite web browser, you can navigate to this address (see Figure 7). You should see the serial monitor display all the connection information (see Figure 8).

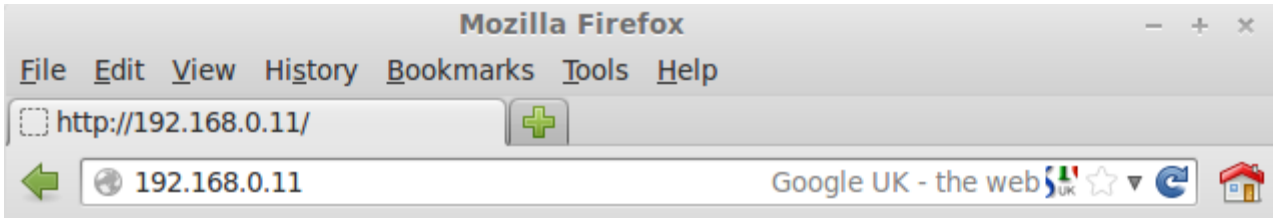

Figure 7 – Navigating to the Arduino web server

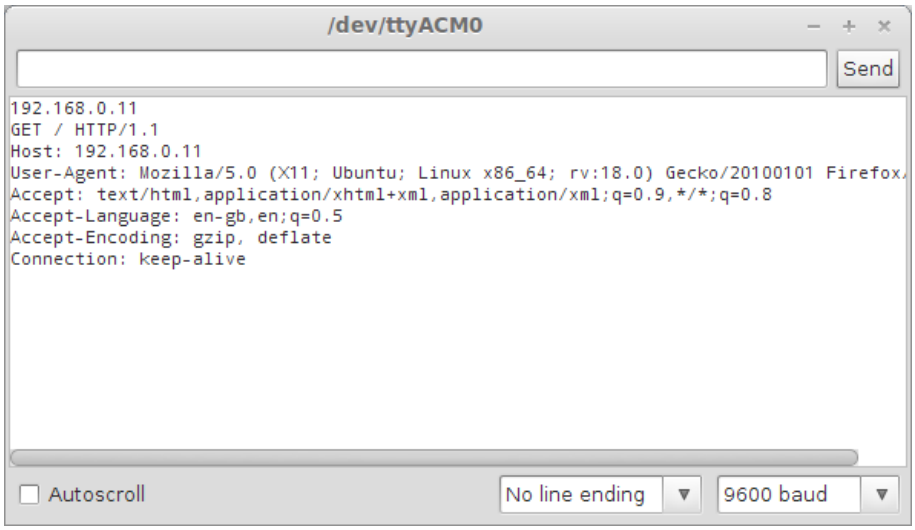

Figure 8 – Serial monitor output

Try entering a different URL and seeing what output you get on the serial monitor – e.g.

```
192.168.0.11/testing
```

Q1 What is the key difference?

Exercises

Now alter your program to actually display some properly formatted HTML in the web page rather than just sending an empty HTTP response header. A good resource for some basic HTML commands is:

http://www.w3schools.com/html/default.asp

Some suggestions are:

1. displaying a simple text string
2. displaying a link to another web page
3. experimenting with the various HTML formatting options

Figure 9 shows a (admittedly hideous) example web page served up by the Arduino. Hopefully you can do better!

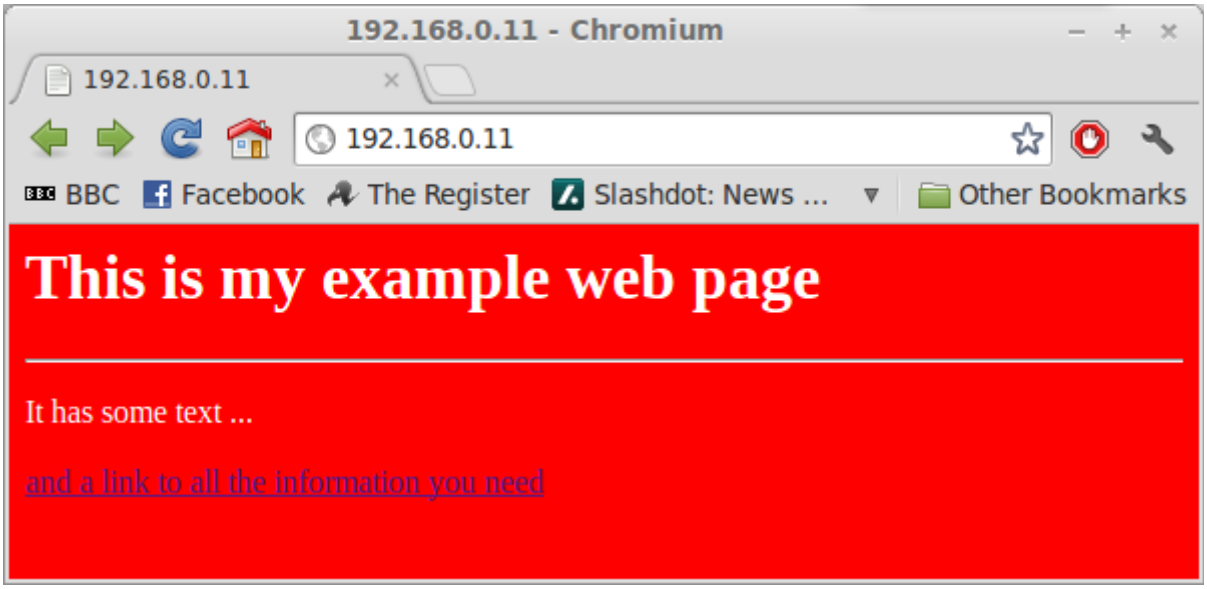

Figure 9 – “My eyes are bleeding!”

Task 2

Now we have mastered the basics and learnt a bit about the structure of HTTP requests and responses, we can use our Arduino as more than just a web server. In fact, by network enabling our Arduino, we are well on our way to creating elements of the “Internet of Things”.

In this task we are going to use the Arduino and Ethernet shield as a web service to control 4 LEDS attached to pins 2, 3, 4, and 5.

First build the circuit shown in Figure 10.

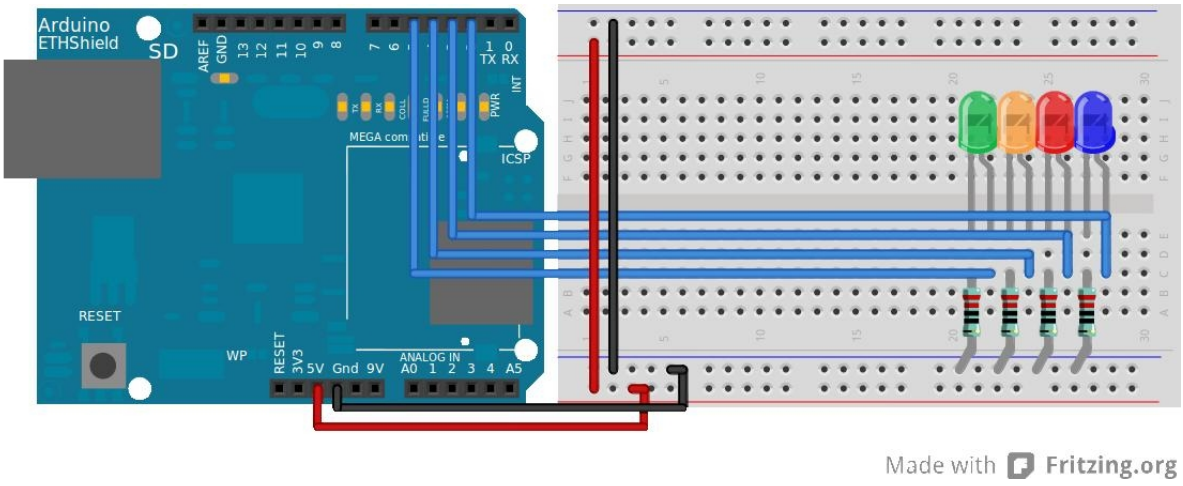

Figure 10 – Ethernet controlled LEDs

We now need to write the code to control these LEDs over the network. We can break this down into 3 parts:

1. the initialisation part (where we set up the Arduino and the network connection) and the main loop of the program (that simply checks for a connection to the server) shown in code listing 3
2. the functions that handle the web client's HTTP requests to the Arduino (shown in code listing 4)
3. the functions that are called to actually control the LEDs (shown in code listing 5)

You should recognise most of the code in code listing 3 from the simple web server we created in Task 1. The HTTP handler code (in code listing 4) basically just parses the information returned from the HTTP client (i.e. the web browser) looking for key markers (in this case a '?' character). Once it see this marker, it knows the the following data is either the Arduino pins to turn on or a command to turn the pins off.

Code listing 3: Setting up the web server

```
/*
 * web_addressable_leds.ino
 *
 * simple web service to control LEDs
 *
 * author:  alex shenfield
 * date:    10/09/2018
 */

#include <Ethernet2.h>
#include <SPI.h>

// set up a server on port 80 (note: web browsers usually
// assume that the server is running on port 80 unless told
// otherwise)
EthernetServer server = EthernetServer(80);

// ip address, gateway and network mask for the web service
IPAddress ip(192, 168, 0, 11);
IPAddress gateway(192, 168, 0, 254);
IPAddress subnet(255, 255, 255, 0);

// ethernet shield mac address (i.e. the hexadecimal numbers found
// on the bottom of the shield)
byte mac[] = {0x90, 0xA2, 0xDA, 0x0F, 0xF5, 0xAD};

// variable indicating when to start paying attention to data
boolean pay_attention = true;

// CODE

// set up code
void setup()
{
  // set up serial comms for debugging and display of
  // DHCP allocated IP address
  Serial.begin(9600);

  // set up pins
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);

  // start the ethernet shield comms - initially try to get a DHCP ip
  // address
  if (Ethernet.begin(mac) == 0)
  {
    // if DHCP fails, allocate a static ip address
    Serial.println("failed to configure ethernet using DHCP");
    Ethernet.begin(mac, ip);
  }

  // start the server, and print the IP address to the serial monitor
  server.begin();
  Serial.println(Ethernet.localIP());
}

// main code
void loop()
{
  // constantly check for connections
  check_for_connections();
}
```

Code listing 4: HTTP request handler functions

```
// method to check for incoming connections and process them
void check_for_connections()
{
  // get any client that is connected to the server and
  // trying to send data
  EthernetClient client = server.available();

  // record whether we have already sent the standard http
  // response header
  boolean header_sent = false;

  // while the client is connected ...
  while (client)
  {
    // if we haven't already sent the http header
    if (!header_sent)
    {
      // send standard http response header (to acknowledge the
      // data)
      client.println("HTTP/1.1 200 OK");
      client.println("Content-Type: text/html");
      client.println();
      header_sent = true;
    }

    // ... and has more data to send
    if (client.available() > 0)
    {
      // read the next byte
      char c = client.read();

      // debugging
      Serial.print(c);

      // pay attention to all the data between the '?' character
      // and a space
      if (c == '?')
      {
        pay_attention = true;
      }
      else if (c == ' ')
      {
        pay_attention = false;
      }
```

```
      // if we are paying attention ...
      if (pay_attention)
      {
        // use a switch statement to decide what to do
        switch (c)
        {
          case '2':
            trigger_pin(2, client);
            break;
          case '3':
            trigger_pin(3, client);
            break;
          case '4':
            trigger_pin(4, client);
            break;
          case '5':
            trigger_pin(5, client);
            break;
          case 'x':
            clear_pins(client);
            break;
        }
      }
    }
    // when the client is done sending data
    else
    {
      // disconnect from client
      client.stop();
    }
  }
}
```

Code listing 5: The LED control functions

```
// separate function for triggering pins - note we only need
// the ethernet client so we can send data to it
void trigger_pin(int pin_number, EthernetClient client)
{
  // print a status message
  client.print("Turning on pin <b>");
  client.println(pin_number);
  client.println("</b><br>");

  // turn on appropriate pin
  digitalWrite(pin_number, HIGH);
}

// another function to clear all pins - again the ethernet client
// is only needed to return data to the web page
void clear_pins(EthernetClient client)
{
  // print a status message
  client.println("Clearing all pins!<br>");

  // turn off pins
  digitalWrite(2, LOW);
  digitalWrite(3, LOW);
  digitalWrite(4, LOW);
  digitalWrite(5, LOW);
}
```

Enter this code, make sure you understand it, and then upload it to the Arduino. You should then be able to control the LEDs by navigating to the IP address shown in the serial monitor.

You can turn on the LEDs on pins 3 and 5 by going to the URL:

```
192.168.0.11/?35
```

and turn off all the LEDs by going to the URL:

```
192.168.0.11/?x
```

This is the first step in exposing some simple functionality of our Arduino as a RESTful web service! We can now turn on some of our digital pins (i.e. the LEDs) by sending simple HTTP GET commands to our Arduino web service.

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Connect up a potentiometer to one of the analog pins and adapt the code above to read (and return) the potentiometer value if we call:

   ```
   192.168.0.11/?p
   ```

2. Add a temperature sensor and light sensor to the system and hook into the web service to return those values when a given URL is called.

3. Add a pushbutton to the system and write a method that returns the last latched button state (i.e. every time the button is pressed we toggle the state and when the method is called we return the last state). Hook that into the web service.

Task 3

So in the last task we exposed some basic functionality of our system as a RESTful web service, providing the ability to control some of the digital outputs of our Arduino via HTTP calls. However, in this case we have had to implement all the HTTP connection parsing code ourselves. For the simple example in the previous task, this is not too difficult – however, if we want to implement a more complex system (for example, monitoring various analog inputs and and providing the ability to call more complex functions via HTTP), the code gets substantially more complex!

Fortunately there is no need to reinvent the wheel! One of the key advantages of the Arduino platform is that there is a vast ecosystem of existing libraries and projects to help us. In this case, Marco Schwartz has written the excellent aREST library for Arduino that enables us to write programs that register variables and methods with a REST API – without having to write all the low level HTTP handling code! See:

- https://github.com/marcoschwartz/aREST
- https://arest.io/

This allows us to focus on the application design rather than the REST API and web service details.

In this task we are going to create a system that uses the aREST framework to allow us to control the brightness of an LED and monitor the ambient light levels within a room – all via HTTP calls to a RESTful web service running on our Arduino.

Code listing 6 shows a pseudo-code outline of this system.

Code listing 6: RESTful light controller pseudo code

```
Procedure LIGHTS:

  initialise_system()

  // register variables and functions with
  // the REST api
  register_variable_with_api(light_level)
  register_function_with_api(led_control)

  LOOP:
    light_level = read_sensor()

    // if we have a client connection,
    // parse the HTTP commands, extract
    // the parameters, and service the
    // callbacks
    if client.connected()
       handle_connection()
    end if

  end LOOP

end Procedure LIGHTS
```

Now build the circuit shown in Figure 11.

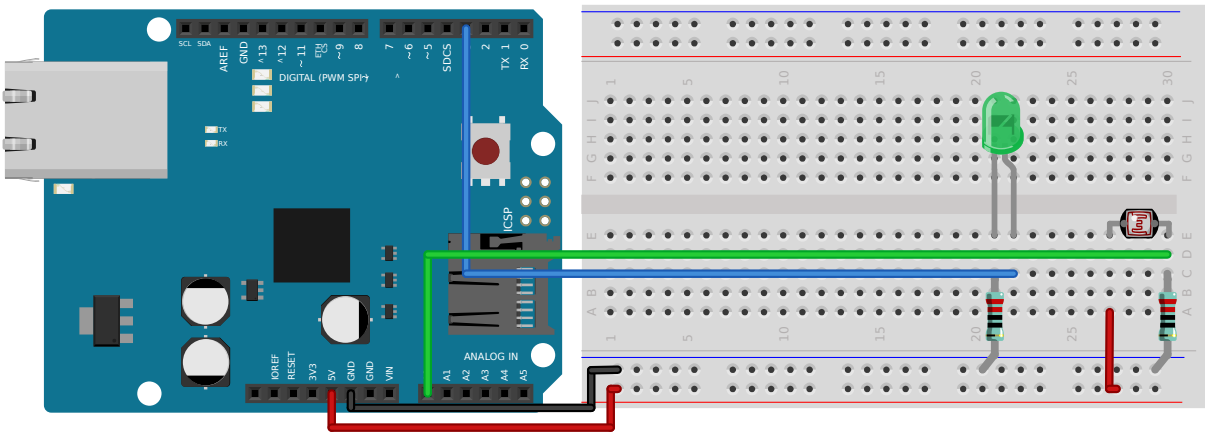

Figure 11 – The light controller circuit

Code listing 7 provides the full code for this RESTful light controller application. Don't forget to change the IP addresses and the Ethernet shield MAC address (as in the previous examples).

Code listing 7: A RESTful lighting controller

```
/*
 * restful_arduino.ino
 *
 * arduino restful api example using the aREST library - in this example we are
 * setting the brightness of an led on pin 3 and reading light values
 *
 * author:  alex shenfield
 * date:    10/09/2018
 */

// INCLUDES

// note: the ethernet shields we have here at SHU use the Ethernet2.h
// library
#include <Ethernet2.h>
#include <SPI.h>

// include the aREST library
#include <aREST.h>

// include the avr watchdog timer library
#include <avr/wdt.h>
```

```
// ETHERNET DECLARATIONS

// set up a server on port 80 to host our webservice
EthernetServer server = EthernetServer(80);

// the ethernet shields and ethernet libraries support DHCP as long as
// you are plugged into a device with a DHCP server however, if you
// are plugged straight in to the network switch in the cabinet (as
// in the embedded lab) you will need to allocate a static ip address
IPAddress ip(192, 168, 137, 11);
IPAddress gateway(192, 168, 137, 254);
IPAddress subnet(255, 255, 255, 0);

// ethernet shield mac address
byte mac[] = {0x90, 0xA2, 0xDA, 0x0F, 0xF5, 0xAD};

// AREST DECLARATIONS

// create an aREST instance
aREST rest = aREST();

// variables to monitor with our webservice
int light_level = 0;
```

```
// CODE

// set up code
void setup()
{
  // set up serial comms for debugging and display of DHCP allocated ip
  // address
  Serial.begin(9600);
  Serial.println("starting rest web service on arduino ...");

  // set pin modes
  pinMode(3, OUTPUT);
  pinMode(A0, INPUT);

  // expose the light level variable to the REST api
  rest.variable("light_level", &light_level);

  // expose the led trigger function to the REST api
  rest.function("led", led_control);

  // set the name and id for this webservice node
  rest.set_id("001");
  rest.set_name("alexs_arduino");

  // start the ethernet shield comms - initially try to get a DHCP ip
  // address
  if (Ethernet.begin(mac) == 0)
  {
    // if DHCP fails, allocate a static ip address
    Serial.println("failed to configure ethernet using DHCP");
    Ethernet.begin(mac, ip);
  }

  // start the server, and print the IP address to the serial
  // monitor
  server.begin();
  Serial.print("web service is at: ");
  Serial.println(Ethernet.localIP());

  // start watchdog timer
  wdt_enable(WDTO_4S);
}

// main loop
void loop()
{
  // update our light level each time round the loop
  light_level = analogRead(A0);

  // listen for incoming clients
  EthernetClient client = server.available();
  rest.handle(client);
  wdt_reset();
}
```

```
// FUNCTIONS EXPOSED TO THE REST API

// led control function accessible by the API
int led_control(String command)
{
  // debugging information about the actual command string
  Serial.print("command is ");
  Serial.println(command);

  // get pwm signal from command
  int pwm = command.toInt();
  pwm = constrain(pwm, 0, 255);

  // send pwm signal to the led
  analogWrite(3, pwm);

  // return 1 (indicating success)
  return 1;
}
```

Enter this code, make sure you understand it, and then upload it to the Arduino. Now open the serial monitor - you should see something like that shown in Figure 12. You can see here that my Arduino isn't plugged in to a switch / router that supports DHCP so I am using the fall-back static IP address in my code. In the labs your Arduino **should** get a DHCP address.

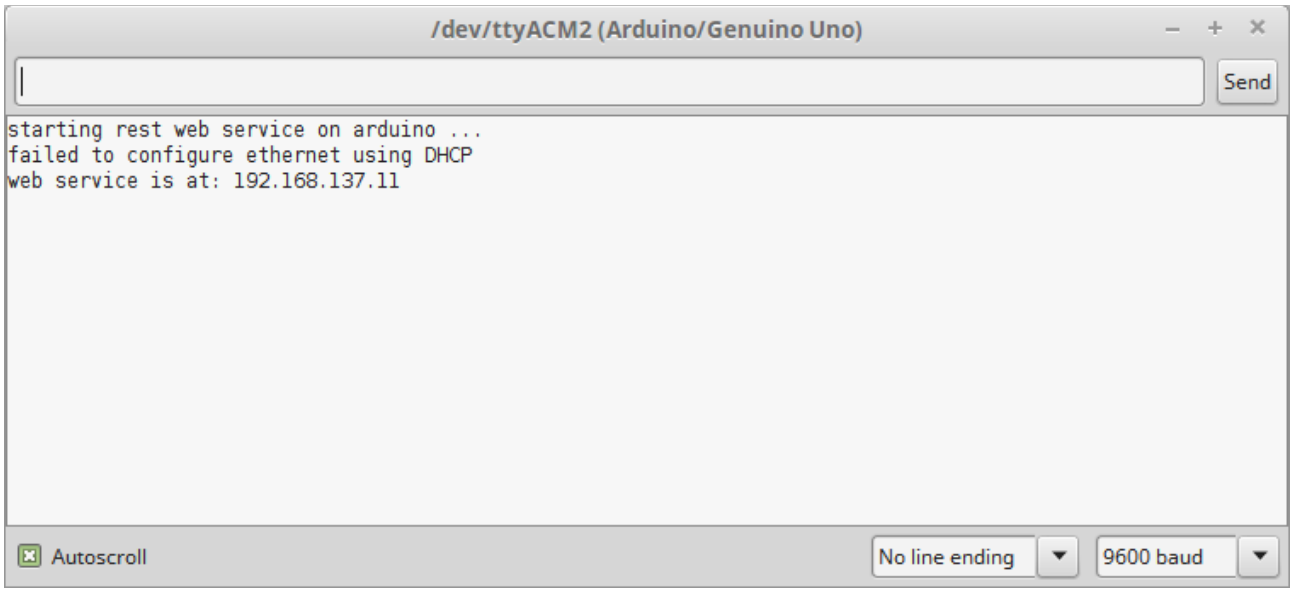

Figure 12 – Serial monitor window

To test the aREST api is working you can open your favourite web browser and enter the URL[1]:

```
192.168.137.11/mode/3/o
```

This sets digital pin 3 on the Arduino to be an output and should return the JSON output shown in Figure 13.

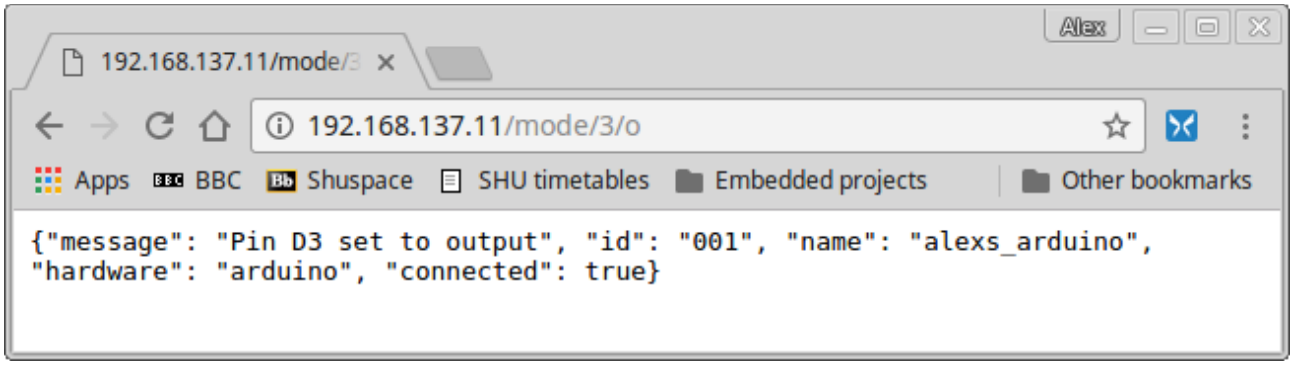

Figure 13 – JSON output for setting D3 as an output

Now if we go to the URL:

```
192.168.137.11/digital/3/1
```

We should see the LED connected to pin 3 light up and the following JSON output returned (see Figure 14).

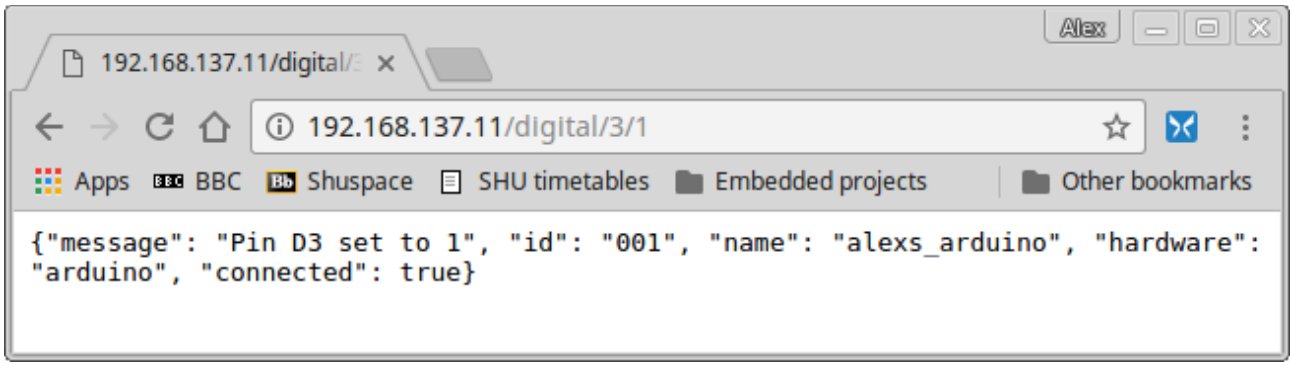

Figure 14 – JSON output for turning pin D3 on

1 Don't forget to change this to the IP address of your IoT device!

Ultimately, when we enter a URL using our web browser, what we are really doing is sending HTTP GET commands to the web service. Whilst we can carry on doing this using a web browser, I am going to use **curl**[2] to send commands to the web service in the rest of these examples.

If we send an HTTP GET request for the web service root the aREST api provides us with some information about what variables are being monitored (see Figure 15).

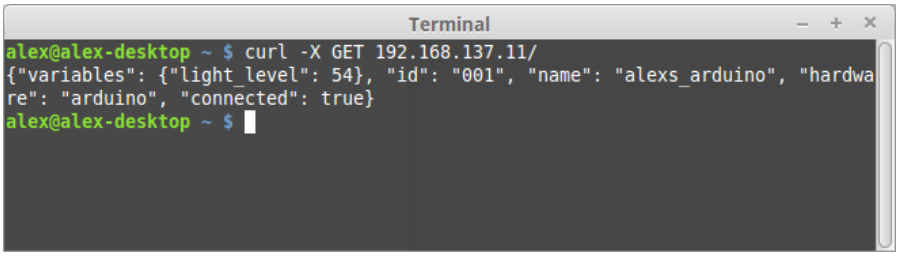

Figure 15 – HTTP GET of the web service root

We can see in Figure 15 that we are monitoring a variable called “light_level” (if we were monitoring more than one variable, they would all show up here). We can then send an HTTP GET request for just that variable to find out what it's current value is (see Figure 16).

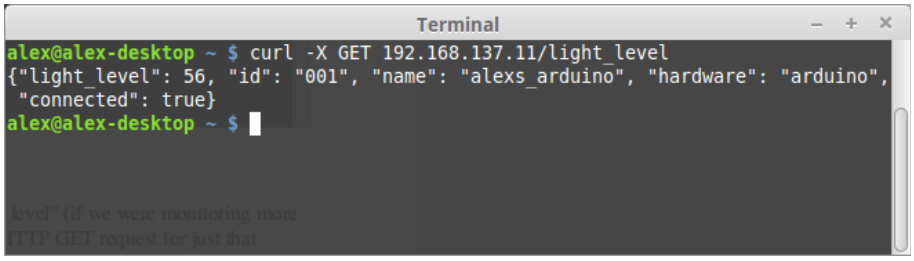

Figure 16 – HTTP GET of the “light_level” variable

2 See https://curl.haxx.se/ for more information - a copy of the windows version of curl is available on blackboard

Now we are going to remotely call the **`led_control`** function via the aREST api. To do this, we need to provide some parameters (i.e. the brightness of the LED as a PWM signal) to the method call.

Send an HTTP GET command with the following format (either via curl or your web browser):

```
192.168.137.11/led?param=_255
```

This will turn on the LED and set it to maximum brightness. Figure 17 shows the curl HTTP GET call and response, Figure 18 shows the debugging output in the serial monitor window, and Figure 19 shows the actual system.

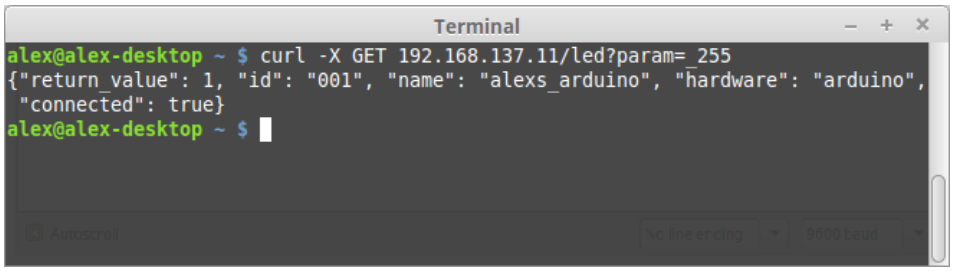

Figure 17 – HTTP GET command setting the brightness of the LED

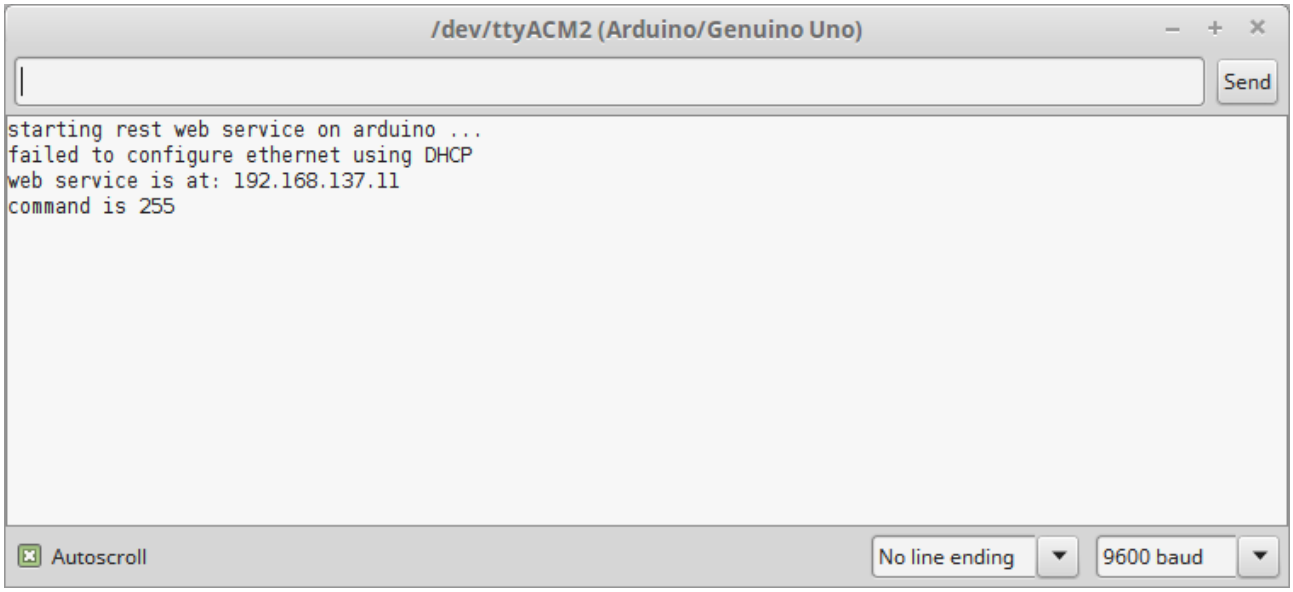

Figure 18 – Serial monitor window showing the debugging output for the application (i.e. exactly what parameter has been passed to the **`led_control`** method)

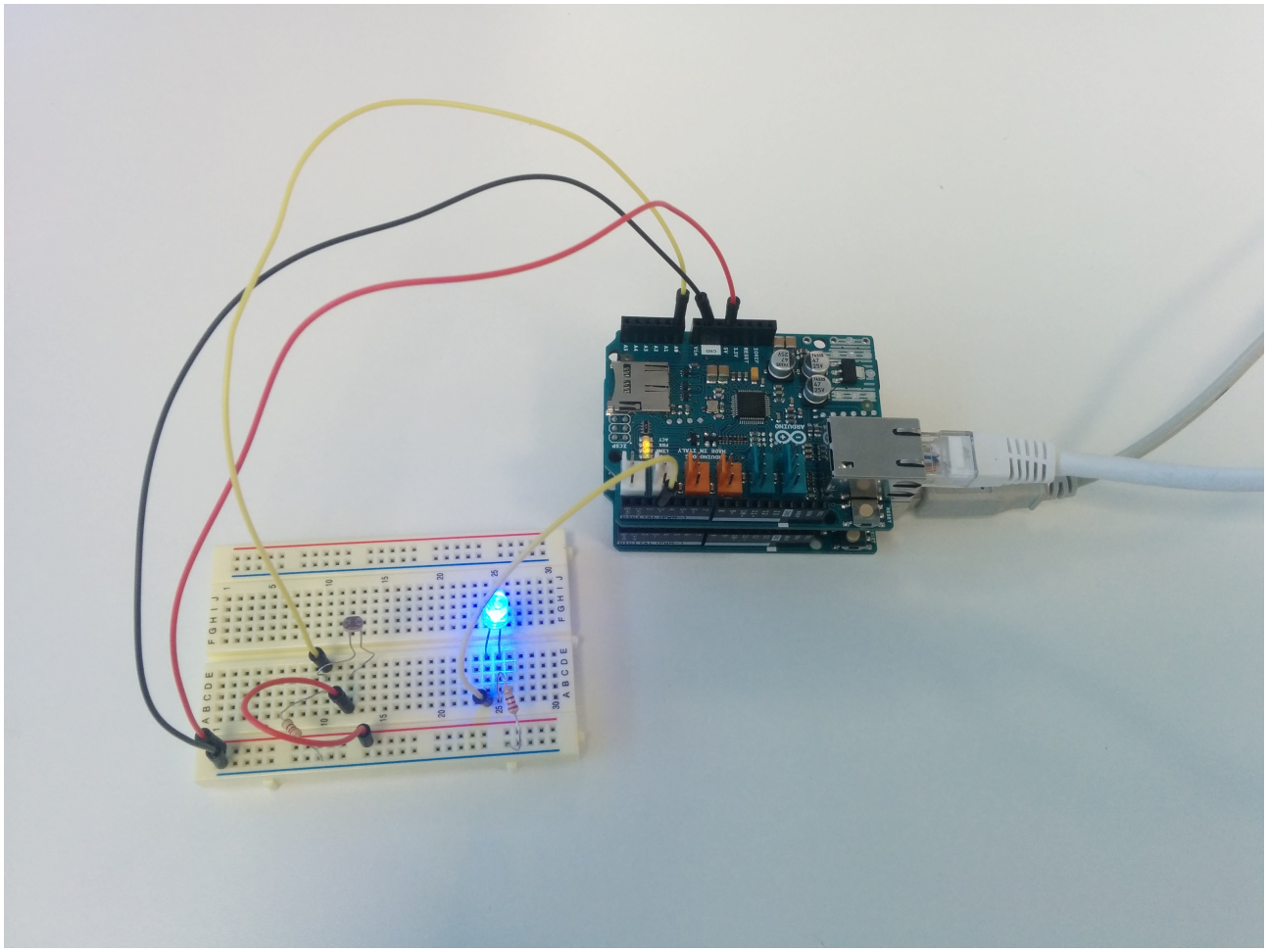

Figure 19 – The embedded web service with the LED turned on at maximum brightness

Now try modifying the HTTP GET request to set the brightness at a different level (e.g. try a value of 50 or 100).

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Provide a calibration routine for the light sensor during the setup() function that means that the light value is remapped as a percentage.

2. Connect up a temperature sensor and potentiometer to some of the other analog pins and adapt the code above (in code listing 7) to monitor those variables too.

3. Add more LEDs to your system and write an additional method that cycles through them every time it is called. Make sure this is hooked in to the web service api.

4. Add an additional method to generate random dice rolls on the Arduino and return the random number. Again, hook this in to the web service api.

Check list

Task 1 – Program	
Task 1 – Q1	
Task 1 – Exercises	
Task 2 – Program	
Task 2 – Additional functionality 1	
Task 2 – Additional functionality 2	
Task 2 – Additional functionality 3	
Task 3 – Program	
Task 3 – Additional functionality 1	
Task 3 – Additional functionality 2	
Task 3 – Additional functionality 3	
Task 3 – Additional functionality 4	

Feedback