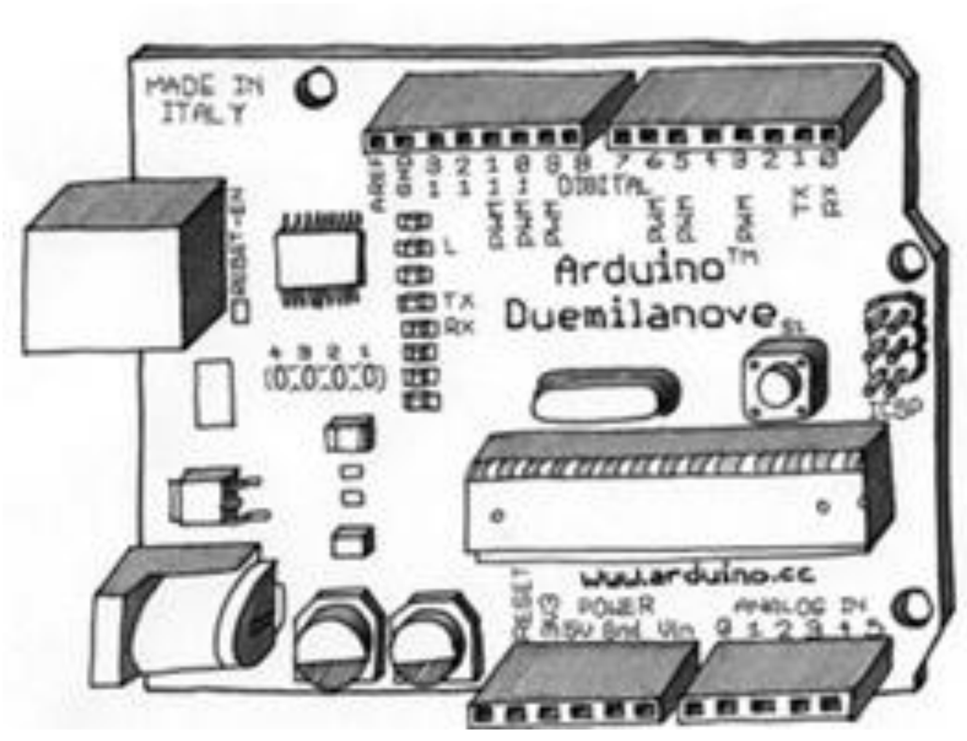
ARDUINO WEB SEVER



Arduino Ethernet Shield Web Server

Part 1 of the Arduino Ethernet Shield Web Server Tutorial

This multi-part tutorial shows how to set up an Arduino with Ethernet shield as a web server. The web servers in this tutorial are used to serve up web pages that can be accessed from a web browser running on any computer connected to the same network as the Arduino.
Some of the Arduino web server pages allow access to the Arduino hardware – this allows hardware to be controlled (e.g. switching on and off an LED from the web page) and monitored (e.g. reading the state of a switch and displaying it on a web page).

The tutorial teaches what is required to build a web server including all the technology such as HTTP, HTML, CSS, JavaScript, AJAX, etc. It starts with the very basics of hosting a simple web page on the Arduino and advances step-by-step from there.

Hardware Components

The hardware required for following this series of tutorials is:

- An Arduino board such as the Arduino Uno
- An Arduino Ethernet shield
- An Ethernet cable, wired straight for connecting to your network router
- A USB cable for powering and programming the Arduino

A micro SD card, e.g. a 2Gb card that is SPI compatible – only required for some of the servers
A computer with a micro SD card slot or a card reader with a micro SD card slot – only required for SD card servers
There will be additional components required as listed in each tutorial, such as LEDs, resistors, switches, etc. and a breadboard and wire kit for building the circuits.

Hardware Setup

Before starting:

- Plug the Ethernet shield into the Arduino, connect it to the network and test it.

Test the SD card in the Ethernet shield.

Basic Arduino Web Server

Part 2 of the Arduino Ethernet Shield Web Server Tutorial

A very basic web server that serves up a single web page using the Arduino Ethernet shield. An SD card is not used in this example as the web page forms part of the Arduino sketch.

Serving Up a Web Page from the Arduino

The following Arduino sketch will enable the Arduino with Ethernet shield to serve up a single web page that can be viewed in a web browser.

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

void setup()
{
   Ethernet.begin(mac, ip);  // initialize Ethernet device
   server.begin();           // start to listen for clients
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
      while (client.connected()) {
         if (client.available()) {   // client data available to read
            char c = client.read(); // read 1 byte (character) from client
            // last line of client request is blank and ends with \n
            // respond to client only after last line received
            if (c == '\n' && currentLineIsBlank) {
               // send a standard http response header
               client.println("HTTP/1.1 200 OK");
               client.println("Content-Type: text/html");
               client.println("Connection: close");
               client.println();
               // send web page
               client.println("<!DOCTYPE html>");
               client.println("<html>");
               client.println("<head>");
               client.println("<title>Arduino Web Page</title>");
               client.println("</head>");
               client.println("<body>");
               client.println("<h1>Hello from Arduino!</h1>");
               client.println("<p>A web page from the Arduino server</p>");
```

```
                client.println("</body>");
                client.println("</html>");
                break;
            }
            // every line of text received from the client ends with \r\n
            if (c == '\n') {
                // last character on line of received text
                // starting new line with next character read
                currentLineIsBlank = true;
            }
            else if (c != '\r') {
                // a text character was received from client
                currentLineIsBlank = false;
            }
        } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}
```

Important Note!

If an uninitialized SD card is left in the SD card socket of the shield, it can cause problems with code in the sketch that is accessing the Ethernet chip. This may cause symptoms such as the sketch running once or twice, then hanging up.

This is because both the Ethernet chip and the SD card are accessed by the Arduino using the same SPI bus.

If the SD card is not being used with an Ethernet application, either remove it from the socket or add the following code to disable the SD card:

```
void setup()
{
  // disable the SD card by switching pin 4 high
  // not using the SD card in this program, but if an SD card is left in the socket,
  // it may cause a problem with accessing the Ethernet chip, unless disabled
  pinMode(4, OUTPUT);
  digitalWrite(4, HIGH);

  // other initialization code goes here...
}
```

Using the Sketch

Copy the above sketch and paste it into the Arduino IDE. Change the MAC address in the sketch to match the numbers on the sticker on the bottom of your Ethernet shield. Change the IP address in the sketch to match the IP address range of your network.

Your hardware must be set up as described in part 1 of this tutorial.

Load the sketch to the Arduino and then open a web browser on a computer that is connected to the same network as the Arduino.

Surf to the Arduino by typing the IP address of the Arduino into the URL field of the browser, e.g. 10.0.0.20 in the above sketch.

The browser should display a web page as shown below.

Web Page Served by Arduino Web Server

Problem Solving

Resetting

If you were not able to connect to the Arduino, try resetting it by pressing the reset button on the Ethernet shield and then surf to the web server again.

IP Address and Address Range

Make sure that you have set the correct Arduino IP address for the address range of your network. The first three numbers of the IP address must match your network. The last number must be unique – i.e. it must be the only device on the network with that number.

Gateway and Subnet Mask

Try specifying the network gateway and subnet mask in the sketch if there are still network connection problems. You will need to change the addresses in the code below to match your network.

Add the gateway and subnet under the MAC address in the sketch:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// the router's gateway address:
byte gateway[] = { 10, 0, 0, 1 };
// the subnet:
byte subnet[] = { 255, 255, 0, 0 };
```

And then initialize the Ethernet device with these settings in the **setup()** part of the sketch:

```
Ethernet.begin(mac, ip, gateway, subnet);
```

Ethernet Cable

When connecting to the network through an Ethernet router/hub/switch, an Ethernet cable that is wired one-to-one must be used to connect the Arduino. Do not use a crossover cable.

Basic Web Server Explained

Read the comments in the above sketch to see what specific lines of code do. This explanation shows what request the server must respond to and what data it must send back.

Client Request

When you surf to the IP address of the Arduino server, the web browser (client) will send a request, such as the one shown below, to the server.

```
GET / HTTP/1.1\r\n
Host: 10.0.0.20\r\n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:17.0)
Gecko/20100101 Firefox/17.0\r\n
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8\r\n
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n
\r\n
```

The information in the request will differ, depending on the browser and operating system that the request is sent from.

The **\r\n** characters that you see at the end of every line of text in the request are non-visible characters (non-printable characters). **\r** is the carriage return character and **\n** is the linefeed character (or newline character).

The last line of the request is simply **\r\n** without and preceding text. This is the blank line that the Arduino sketch checks for before sending a response to the client web browser.

In other words, the sketch reads every character from the above request and knows when the end of the request has been reached because it finds the blank line.

Server Response

After receiving the request for a web page from the client, the server first sends a standard HTTP response and then the web page itself.

The response sent from the Arduino is as follows:

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
```

```
Connection: close\r\n
\r\n
```

Again the non-visible characters **\r\n** are shown in the above response. The **println()** function in the the sketch automatically adds the **\r\n** characters to the end of each line. The empty **println()** function at the end of the HTTP response simply sends the **\r\n** with no text in front of it.

The above request and response are part of HTTP (Hypertext Transfer Protocol).

Web Page

After the server has sent the HTTP response, it sends the actual web page which is then displayed in the browser.

The web page consists of text with HTML tags. You do not see the tags in the browser as these tags are interpreted by the browser.

To see the actual HTML source code, in the browser right-click on the page from the Arduino server and then click **View Page Source**.

The actual HTML markup tags are shown below.

```
File  Edit  View  Help

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Arduino Web Page</title>
5  </head>
6  <body>
7  <h1>Hello from Arduino!</h1>
8  <p>A web page from the Arduino server</p>
9  </body>
10 </html>
11
```

Web Page HTML Code (Markup)

HTML and other web page code is explained in the next part of this tutorial.

Web Page Structure (HTML)

Part 3 of the Arduino Ethernet Shield Web Server Tutorial

The Arduino web servers in this tutorial are used to serve up HTML web pages, so it makes sense at this stage to find out more about HTML, which is what this part of the tutorial covers.

HTML Structure and Pages

The basic structure of an HTML page is shown below (this code is from the previous tutorial).

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino Web Page</title>
    </head>
    <body>
        <h1>Hello from Arduino!</h1>
        <p>A web page from the Arduino server</p>
    </body>
</html>
```

HTML Tags

HTML markup code consists of tags between angle brackets: **< >**

The name of the html tag is put between the opening and closing angle brackets.

Most tags will have an opening and closing tag. The text or resource placed between the opening and closing set of tags will be formatted by the browser according to the type of tag. The closing tag is exactly the same as the opening tag, except that the closing tag has a forward slash after the opening angle bracket. e.g.: **<p>Paragraph text...</p>** – here the paragraph tag (<p>) is used to tell the browser that the text between the opening <p> and closing </p> is a paragraph of text. The browser will format it accordingly.

An example of a tag that does not have a closing tag is the line break which moves to the next line in the web page. This is written as **
** (following the HTML standard) or **
** (following the XHTML standard).

Learning HTML is about learning HTML tags – what tags are available, what they do and which tags can be inserted between which other tags.

Web Page Structure

Web pages consist of two main sections – a head section and a body section. These two sections are placed between opening and closing html tags as shown here.

```
<html>
    <head>
    </head>
    <body>
    </body>
```

```
</html>
```

Things that are to be visible on the web page or apply to the web page content are placed between the body tags.

Things that do not appear on the page are placed between the head tags, e.g. the text for the title of the page that appears on the top bar of the web browser window. Also files such as style sheets can be included here.

Basic HTML Tags

We have already seen the paragraph HTML tag – <p>, and the invisible tags that make up sections of an HTML page – <html>, <head> and <body>. Below are two more HTML tags that were used in the first Arduino server example.

Additional tags will be introduced in this tutorial as they are used.

Heading Tag

Heading tags create heading text which is normally made bold and larger than the paragraph text by the browser. In our first Arduino server, the heading 1 tag was used – <h1>. This is a top level heading and has a corresponding closing tag. All text placed between <h1> and </h1> is marked as heading level 1.

Sub-heading text is normally smaller than h1 text and is designated h2, h3, h4, etc. (<h2>, <h3>, <h4>, etc)

The main heading h1 is used to mark a chapter heading for example – e.g. Chapter 1, the h2 marks a sub heading, e.g. heading 1.1, 1.2, 2.1, etc., h3 marks a sub heading of an h2 heading, e.g. 1.1.1 and 1.1.2, etc.

Each additional heading level will be rendered in smaller text by the browser.

Title Tag

The title tag, <title>, is placed in the <head> section of the HTML page and will display text in the top bar of the web browser. This tag is intended to display the web page title.

Web Server Example

The WebServer example sketch found in the Arduino software (found under **File → Examples → Ethernet → WebServer** – already covered in the article Plugging In and Testing the Arduino Ethernet Shield) actually does not conform to the full HTML page structure, but instead places text directly between the opening and closing <html> tags.

In the WebServer example, each line is ended with a line break so that the next line is shown below the previous line in the browser. The following image shows the output of the WebServer sketch in the browser and the HTML code used to produce the output text.

10.0.0.20

analog input 0 is 1023
analog input 1 is 1023
analog input 2 is 695
analog input 3 is 564
analog input 4 is 406
analog input 5 is 340

Source of: http://10.0.0.20/ - Mozill

File Edit View Help

```
<!DOCTYPE HTML>
<html>
<meta http-equiv="refresh" content="5">
analog input 0 is 1023<br />
analog input 1 is 1023<br />
analog input 2 is 691<br />
analog input 3 is 561<br />
analog input 4 is 405<br />
analog input 5 is 331<br />
</html>
```

Output from the WebServer Sketch – Web Page on Left, HTML Code on Right

Learning HTML

HTML tags will be introduced as needed in this tutorial, but if you would like to learn more about HTML, either search on the Internet or pick up an HTML book.

Arduino SD Card Web Server

Part 4 of the Arduino Ethernet Shield Web Server Tutorial

The Arduino, Arduino Ethernet shield and micro SD card are used to make a web server that hosts a web page on the SD card. When a browser requests a web page from the Arduino web server, the Arduino will fetch the web page from the SD card.

Creating the Web Page

Because the web page is to be stored on the SD card, it must first be created using a text editor and then copied to the SD card.

Web Page Editor

A text editor such as Geany can be used – it is available to download for Windows and will be in the repositories for most Ubuntu based Linux distributions. Geany has syntax highlighting and will automatically close HTML tags for you which makes web page editing easier. It is possible to use any other text editor, even Windows Notepad.

Web Page

Create the following web page in a text editor. When you save the text file, give it the name: **index.htm**

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
    </head>
    <body>
        <h1>Hello from the Arduino SD Card!</h1>
        <p>A web page from the Arduino SD card server.</p>
    </body>
</html>
```

Nothing new here, it is the same as the web page from the first web server in this tutorial with just the text changed. Test this web page by opening it in a web browser.

Copying the Web Page

You will need a micro SD card slot on your computer or a card reader that is capable of reading and writing a micro SD card.

Insert the micro SD card into the slot on the computer or card reader that is plugged into the computer and copy the **index.htm** file to the micro SD card.

Now plug the SD card into the micro SD card slot on the Ethernet shield.

SD Card Web Server

Hardware

You should now have the micro SD card with web page copied to it inserted into the card slot on the Arduino Ethernet shield. The Ethernet shield should be plugged into a compatible Arduino and into an Ethernet cable connected to your network. The Arduino / Ethernet shield should be powered from a USB cable.

Arduino Sketch

The Arduino sketch that fetches the web page from the SD card and sends it to the browser is shown below.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

File webFile;

void setup()
{
    Ethernet.begin(mac, ip);  // initialize Ethernet device
    server.begin();           // start to listen for clients
    Serial.begin(9600);       // for debugging

    // initialize SD card
    Serial.println("Initializing SD card...");
    if (!SD.begin(4)) {
        Serial.println("ERROR - SD card initialization failed!");
        return;    // init failed
    }
    Serial.println("SUCCESS - SD card initialized.");
    // check for index.htm file
    if (!SD.exists("index.htm")) {
        Serial.println("ERROR - Can't find index.htm file!");
        return;  // can't find index file
    }
    Serial.println("SUCCESS - Found index.htm file.");
}

void loop()
{
    EthernetClient client = server.available();  // try to get client

    if (client) {  // got client?
        boolean currentLineIsBlank = true;
```

```
    while (client.connected()) {
      if (client.available()) {   // client data available to read
        char c = client.read(); // read 1 byte (character) from client
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: close");
          client.println();
          // send web page
          webFile = SD.open("index.htm");        // open web page file
          if (webFile) {
            while(webFile.available()) {
              client.write(webFile.read()); // send web page to client
            }
            webFile.close();
          }
          break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
          // last character on line of received text
          // starting new line with next character read
          currentLineIsBlank = true;
        }
        else if (c != '\r') {
          // a text character was received from client
          currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}
```

Using the Sketch

Copy the above sketch and paste it into the Arduino IDE. Load the sketch to the Arduino and then surf to the IP address set in the sketch with your web browser. The web page that you created should be displayed in the browser as it is served up by the Arduino SD card web server.

Fault Finding

If the previous sketch in this tutorial worked, then the only thing that can go wrong is

with initializing the SD card and finding the index.htm file on the card. If the file is not on the card or does not have the exact name index.htm, then the server will not be able to display the web page.

Open up the Arduino serial monitor window to see SD card diagnostic information.

Sketch Explanation

This sketch is a modified version of the **eth_websrv_page** sketch from the Basic Arduino Web Server part of this tutorial.

Additional Code

The sketch now initializes the SD card in the **setup()** function and sends diagnostic information out of the serial port that can be viewed in the Arduino serial monitor window.

Instead of sending the web page line by line from within the code as in the **eth_websrv_page** sketch, this new sketch now opens the **index.htm** file from the SD card and sends the contents to the web client (the web browser).

Arduino Web Server LED Control

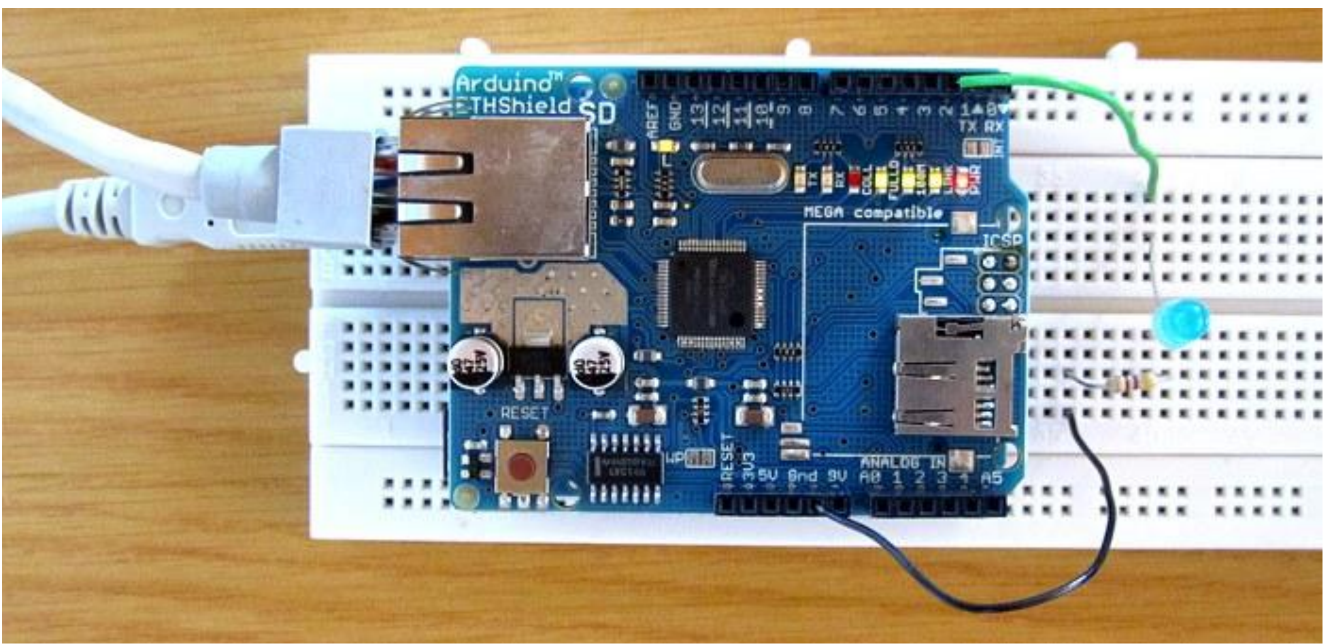
Part 5 of the Arduino Ethernet Shield Web Server Tutorial
In this part of the tutorial, the Arduino and Ethernet shield serves up a web page that allows an LED to be switched on and off. The LED is connected to one of the Arduino pins – this simple circuit can be built on a breadboard.

Arduino Web Server LED Controller Hardware

The LED is interfaced to the Arduino as shown in the circuit diagram in the Starting with Arduino tutorial. It is simply an LED and series resistor connected between Arduino pin 2 and GND.

An SD card is not used in this web server.

The hardware is shown in the image below.



LED Web Server Hardware

How the LED is Controlled

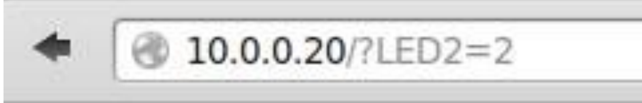
Web Page and HTML

Web Page and HTML Code with Checkbox Unchecked

The Arduino web server serves up a page that allows the user to click a check box to switch the LED on and off. The web page is shown here:

Web Page and HTML Code with Checkbox Checked

After clicking the checkbox to switch the LED on, the web page and HTML code now look as follows:

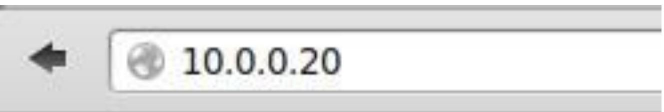


LED

Click to switch LED on and off.

LED Web Page with Checkbox Checked

Take note in the above image that the web browser added **/?LED2=2** to the end of the URL field after the checkbox was clicked.



LED

Click to switch LED on and off.

LED2

LED Web Server Web Page - Checkbox Unchecked

The HTML code that the Arduino web server sends to the web browser is shown below.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino LED Control</title>
    </head>
    <body>
        <h1>LED</h1>
        <p>Click to switch LED on and off.</p>
        <form method="get">
            <input type="checkbox" name="LED2" value="2" onclick="submit();">LED2
        </form>
    </body>
</html>
```

LED Web Server Web Page HTML Code - Checkbox Unchecked

In the above image, the Arduino changed the HTML page that it sent to the browser so that the checkbox will be shown with a check mark in it. The change to the code is highlighted in the image and it can be seen that **checked** was added.

New HTML Tags

Two new HTML tags are introduced in the above HTML code, namely **<form>** and **<input>**.

HTML <form> Tag

A form tag contains form controls, such as the checkbox used in this example. In this form, **method="get"** in the opening form tag will result in the form being submitted using an HTTP GET request. This also results in the **/?LED2=2** text being added in the URL field of the web browser.

HTML <input> Tag

A single control is added to the HTML form using the **<input>** tag. The input tag does not have a corresponding closing tag.

In this example, the input tag is used to create a checkbox. The following fields are included in the input tag:

type="checkbox" – displays this input control as a checkbox
name="LED2" – user defined name of the control
value="2" – user defined value of the control
onclick="submit();" – submit the form when the checkbox control is clicked
checked – when present the checkbox is checked, otherwise it is blank

HTTP Request and Response

When the checkbox is clicked, it will generate an HTTP GET request that sends the name and value from the checkbox to the Arduino server.

The following is an example of an HTTP request sent from the Firefox browser to the Arduino server after clicking the checkbox:

```
GET /?LED2=2 HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:18.0)
Gecko/20100101 Firefox/18.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.20/
Connection: keep-alive
```

When unchecking the checkbox, the following HTTP request is sent from the browser to the Arduino web server:

```
GET / HTTP/1.1
```

```
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:18.0) Gecko/20100101 Firefox/18.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.20/?LED2=2
Connection: keep-alive
```

The Arduino sketch in this example reads the HTTP request header and checks for the text **LED2=2** and if found, the Arduino will toggle the LED from off to on or on to off.

Both of the above requests contain the **LED2=2** text although in different places. When checking the box, the text is part of the GET request line. When unchecking the box, the text is part of the **Referer:** header.

With this background information, we can now see how the Arduino sketch works.

LED Web Server Sketch

The Arduino sketch for the LED web server is shown below.

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

String HTTP_req;          // stores the HTTP request
boolean LED_status = 0;   // state of LED, off by default

void setup()
{
   Ethernet.begin(mac, ip);  // initialize Ethernet device
   server.begin();           // start to listen for clients
   Serial.begin(9600);       // for diagnostics
   pinMode(2, OUTPUT);       // LED on pin 2
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
```

```
    while (client.connected()) {
      if (client.available()) {   // client data available to read
        char c = client.read(); // read 1 byte (character) from client
        HTTP_req += c;  // save the HTTP request 1 char at a time
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: close");
          client.println();
          // send web page
          client.println("<!DOCTYPE html>");
          client.println("<html>");
          client.println("<head>");
          client.println("<title>Arduino LED Control</title>");
          client.println("</head>");
          client.println("<body>");
          client.println("<h1>LED</h1>");
          client.println("<p>Click to switch LED on and off.</p>");
          client.println("<form method=\"get\">");
          ProcessCheckbox(client);
          client.println("</form>");
          client.println("</body>");
          client.println("</html>");
          Serial.print(HTTP_req);
          HTTP_req = "";    // finished with request, empty string
          break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
          // last character on line of received text
          // starting new line with next character read
          currentLineIsBlank = true;
        }
        else if (c != '\r') {
          // a text character was received from client
          currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

// switch LED and send back HTML for LED checkbox
```

```
void ProcessCheckbox(EthernetClient cl)
{
  if (HTTP_req.indexOf("LED2=2") > -1) {  // see if checkbox was clicked
    // the checkbox was clicked, toggle the LED
    if (LED_status) {
      LED_status = 0;
    }
    else {
      LED_status = 1;
    }
  }

  if (LED_status) {    // switch LED on
    digitalWrite(2, HIGH);
    // checkbox is checked
    cl.println("<input type=\"checkbox\" name=\"LED2\" value=\"2\" \
    onclick=\"submit();\" checked>LED2");
  }
  else {           // switch LED off
    digitalWrite(2, LOW);
    // checkbox is unchecked
    cl.println("<input type=\"checkbox\" name=\"LED2\" value=\"2\" \
    onclick=\"submit();\">LED2");
  }
}
```

Modification to Sketch

This sketch is a modified version of the **eth_websrv_page** sketch from the basic Arduino web server.

The sketch creates the HTML page as usual, but calls the **ProcessCheckbox()** function to take care of the line that draws the checkbox.

The **ProcessCheckbox()** function checks to see if the HTTP request contains the text **LED2=2**. If the HTTP request does contain this text, then the LED will be toggled (switched from on to off or from off to on) and the web page is sent again with the checkbox control also toggled to reflect the state of the LED.

Improvements

The sketch has been kept simple for learning purposes, but some improvements can be made to this sketch to make it more reliable.

The sketch currently only checks for the presence of the text **LED2=2** in the HTTP request to see if the checkbox was clicked. It would be more reliable to check where the **LED2=2** text is in the HTTP message to determine whether the checkbox is being checked or unchecked. This would then make it impossible for the state of the LED and the state of the checkbox to become unsynchronized.

Reading Switch State using an Arduino Web Server

Part 6 of the Arduino Ethernet Shield Web Server Tutorial

A push button switch interfaced to the Arduino is read to see whether it is on or off. The state of the switch is displayed on a web page. The Arduino with Ethernet shield is set up as a web server and accessed from a web browser.
The browser refreshes the web page every second, so it can take up to a second for the new state of the switch to be displayed after pressing or releasing the button.

Arduino Web Server Hardware for Reading the Switch

The switch is interfaced to the Arduino / Ethernet shield as done in the circuit diagram from this article: Project 4: Switch a LED on when Switch is Closed (Button) except that **the switch is connected to pin 3 and not pin 2 of the Arduino** (the article actually uses the circuit diagram from one of the Arduino examples on the Arduino website).

Switch Web Server Sketch

The source code for the switch status Arduino web server is shown below.

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

void setup()
{
   Ethernet.begin(mac, ip);    // initialize Ethernet device
   server.begin();           // start to listen for clients
   pinMode(3, INPUT);  // input pin for switch
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
      while (client.connected()) {
         if (client.available()) {   // client data available to read
            char c = client.read(); // read 1 byte (character) from client
```

```
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connnection: close");
          client.println();
          // send web page
          client.println("<!DOCTYPE html>");
          client.println("<html>");
          client.println("<head>");
          client.println("<title>Arduino Read Switch State</title>");
          client.println("<meta http-equiv=\"refresh\" content=\"1\">");
          client.println("</head>");
          client.println("<body>");
          client.println("<h1>Switch</h1>");
          client.println("<p>State of switch is:</p>");
          GetSwitchState(client);
          client.println("</body>");
          client.println("</html>");
          break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
          // last character on line of received text
          // starting new line with next character read
          currentLineIsBlank = true;
        }
        else if (c != '\r') {
          // a text character was received from client
          currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

void GetSwitchState(EthernetClient cl)
{
  if (digitalRead(3)) {
    cl.println("<p>ON</p>");
  }
  else {
    cl.println("<p>OFF</p>");
  }
```

}

Modification to Sketch

Again, this sketch is a modified version of the **eth_websrv_page** sketch from the basic Arduino web server.

Reading the Switch

The web page is created as usual, except that the function **GetSwitchState()** is called when the text for the switch is to be displayed.

In the **GetSwitchState()** function, the state of the switch is read. The text that is sent to the browser will be a HTML paragraph that contains either "ON" or "OFF", depending of the state of the switch.

Refreshing the Browser

A line of HTML in the <head> part of the HTML page sent to the browser is used to tell the browser to refresh the page every second. This allows the new state of the switch to be displayed if it has changed.

The line of code in the sketch that does this is shown here:

```
client.println("<meta http-equiv=\"refresh\" content=\"1\">");
```

This will be sent to the browser as the following HTML code:

```
<meta http-equiv="refresh" content="1">
```

Remember that you can right-click on the web page in your browser and then select **View Page Source** on the pop-up menu (or similar menu item depending on the browser you are using).

The "1" in the code tells the browser to refresh every 1 second.

This is the same method used to read the analog inputs of the Arduino in the **WebServer** example that is built into the Arduino software (found in the Arduino IDE under **File** → **Examples** → **Ethernet** → **WebServer**).

Improving this Example

The annoying thing about this method of refreshing the page is that the browser flashes every second as it updates the page. In the next part of this tutorial we will use a method called AJAX that will refresh only part of the web page that displays the switch state.

Arduino Web Server Switch Status Using AJAX Manually

Part 7 of the Arduino Ethernet Shield Web Server Tutorial
The state of a switch connected to the Arduino / Ethernet shield is shown on a web page that is hosted by the Arduino. AJAX is used to fetch the state of the switch when a button on the web page is clicked.

The reason for using a button on the web page to refresh the state of the switch is to keep the code simple for those who are new to AJAX. The next part of this series will automate the reading of the switch using AJAX for a more practical application.

This video shows the Arduino web server displaying the switch status using AJAX.

What is AJAX?

AJAX stands for Asynchronous JavaScript and XML.

AJAX is basically the use of a set of JavaScript functions for getting information from the web server (our Arduino). This means that data on a web page can be updated without fetching the whole page each time.

Using AJAX will be an improvement on the previous part of this tutorial as HTML refresh code that makes the page flicker each time it is reloaded is no longer needed. Only the information that has changed (the state of the switch) will be updated on the page eliminating the flicker.

What is JavaScript?

JavaScript is a client side scripting language. This means that it is code that will run on the web browser.

JavaScript is included in the HTML page. When you surf to the HTML web page hosted by the Arduino, the page and the JavaScript is loaded to your browser. Your browser then runs the JavaScript code (provided that you have not disabled JavaScript in your browser).

Web Server Hardware

The switch is interfaced to the Arduino / Ethernet shield as done in the circuit diagram from this article: Project 4: Switch a LED on when Switch is Closed (Button) except that **the switch is connected to pin 3 and not pin 2 of the Arduino** (the article actually uses the circuit diagram from one of the Arduino examples on the Arduino website).

Arduino AJAX Sketch

The sketch for this part of the tutorial is shown below. Copy it and paste it into your Arduino IDE and then load it to the Arduino.

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

String HTTP_req;          // stores the HTTP request

void setup()
{
  Ethernet.begin(mac, ip);  // initialize Ethernet device
  server.begin();          // start to listen for clients
  Serial.begin(9600);      // for diagnostics
  pinMode(3, INPUT);       // switch is attached to Arduino pin 3
}

void loop()
{
  EthernetClient client = server.available();  // try to get client

  if (client) {  // got client?
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {   // client data available to read
        char c = client.read(); // read 1 byte (character) from client
        HTTP_req += c;  // save the HTTP request 1 char at a time
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println("Connection: keep-alive");
          client.println();
          // AJAX request for switch state
          if (HTTP_req.indexOf("ajax_switch") > -1) {
            // read switch state and send appropriate paragraph text
            GetSwitchState(client);
          }
          else {  // HTTP request for web page
            // send web page - contains JavaScript with AJAX calls
            client.println("<!DOCTYPE html>");
            client.println("<html>");
            client.println("<head>");
            client.println("<title>Arduino Web Page</title>");
            client.println("<script>");
```

```
                client.println("function GetSwitchState() {");
                client.println("nocache = \"&nocache=\"\
                                        + Math.random() * 1000000;");
                client.println("var request = new XMLHttpRequest();");
                client.println("request.onreadystatechange = function() {");
                client.println("if (this.readyState == 4) {");
                client.println("if (this.status == 200) {");
                client.println("if (this.responseText != null) {");
                client.println("document.getElementById(\"switch_txt\")\
.innerHTML = this.responseText;");
                client.println("}}}}");
                client.println(
                "request.open(\"GET\", \"ajax_switch\" + nocache, true);");
               //client.println("request.open(\"GET\", \"ajax_switch\", true);");
                client.println("request.send(null);");
                client.println("}");
                client.println("</script>");
                client.println("</head>");
                client.println("<body>");
                client.println("<h1>Arduino AJAX Switch Status</h1>");
                client.println(
                "<p id=\"switch_txt\">Switch state: Not requested...</p>");
                client.println("<button type=\"button\"\
                  onclick=\"GetSwitchState()\">Get Switch State</button>");
                client.println("</body>");
                client.println("</html>");
              }
              // display received HTTP request on serial port
              Serial.print(HTTP_req);
              HTTP_req = "";          // finished with request, empty string
              break;
            }
            // every line of text received from the client ends with \r\n
            if (c == '\n') {
              // last character on line of received text
              // starting new line with next character read
              currentLineIsBlank = true;
            }
            else if (c != '\r') {
              // a text character was received from client
              currentLineIsBlank = false;
            }
          } // end if (client.available())
        } // end while (client.connected())
        delay(1);      // give the web browser time to receive the data
        client.stop(); // close the connection
      } // end if (client)
}
```

```
// send the state of the switch to the web browser
void GetSwitchState(EthernetClient cl)
{
   if (digitalRead(3)) {
      cl.println("Switch state: ON");
   }
   else {
      cl.println("Switch state: OFF");
   }
}
```

HTML and JavaScript

The above sketch will send the following HTML and JavaScript to the web browser.

```
<!DOCTYPE html>
<html>
<head>
    <title>Arduino Web Page</title>
    <script>
        function GetSwitchState()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseText != null) {
                            document.getElementById("switch_txt").innerHTML = this.responseText;
                        }
                    }
                }
            }
            request.open("GET", "ajax_switch" + nocache, true);
            request.send(null);
        }
    </script>
</head>
<body>
    <h1>Arduino AJAX Switch Status</h1>
    <p id="switch_txt">Switch state: Not requested...</p>
    <button type="button" onclick="GetSwitchState()">Get Switch State</button>
</body>
</html>
```

HTML and JavaScript Hosted by the Arduino

Page Structure

In the <head> part of the HTML code, a JavaScript function can be found between the opening and closing <script> tags.

Whenever the button on the web page is clicked, the **GetSwitchState()** JavaScript function is called.

JavaScript Function

When the web page button is clicked and the **GetSwitchState()** function is called, it sends a HTTP GET request to the Arduino that contains the text "ajax_switch". This request looks as follows:

```
GET /ajax_switch&nocache=29860.903564600583 HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:18.0)
Gecko/20100101 Firefox/18.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.20/
Connection: keep-alive
```

When the Arduino receives this request (containing the ajax_switch text), it responds with a standard HTTP response header followed by text that contains the state of the switch.

In the Arduino code, the function **GetSwitchState()** will read the switch state on the Arduino pin and send the text **Switch state: ON** or **Switch state: OFF**.

When the JavaScript in the browser receives this response, it runs the code in the unnamed function **request.onreadystatechange = function()**. This function runs every time that the Arduino sends a response to the browser. It replaces the **Switch state: x** text on the web page (or the default text **Switch state: Not requested...**) with the new text received from the Arduino.

This JavaScript request from the browser and response from the Arduino is AJAX in action.

AJAX Summarized

The AJAX operation performed in this example can be summarized as follows:

1. AJAX Request from Browser

When the button on the web page is clicked, the JavaScript function **GetSwitchState()** is run. This function does the following:

1. Generates a **random** number to send with the GET request: **nocache = "&nocache=" + Math.random() * 1000000;**
2. **Creates** a **XMLHttpRequest()** object called **request**: **var request = new XMLHttpRequest();**
3. Assigns a **function** to handle the response from the web server: **request.onreadystatechange = function()** (and following code between braces { }).
4. Sets up a **HTTP** GET request to send to the web server: **request.open("GET",**

"ajax_switch" + nocache, true);

5. Sends the HTTP request: **request.send(null);**

2. Response from Arduino Web Server

When the Arduino web server receives the HTTP GET request, it sends back a standard HTTP response followed by text that represents the state of the switch. The state of the switch and the text sent is obtained from the Arduino's own **GetSwitchState()** function.

3. Browser JavaScript Handles Response

The HTTP response from the Arduino web server is handled by the JavaScript code. The JavaScript event handler function runs when the response from the Arduino is received (the event handler function is the unnamed function assigned to **request.onreadystatechange**).

If the received response is OK and not empty, then this line of JavaScript is run:

```
document.getElementById("switch_txt").innerHTML =
this.responseText;
```

This JavaScript finds the paragraph in the HTML that is marked with the ID **switch_txt** and replaces the current text with the text received from the Arduino. The HTML for this paragraph looks as follows:

```
<p id="switch_txt">Switch state: Not requested...</p>
```

This example has illustrated the use of AJAX used to update a single paragraph of text in the browser. The next part of this tutorial will automate the AJAX request so that a button does not have to be clicked to initiate the request.

Reading Switch Status Automatically using AJAX on the Arduino Web Server

Part 8 of the Arduino Ethernet Shield Web Server Tutorial

With a slight modification to the HTML and JavaScript code in the Arduino sketch from the previous part of this tutorial, the Arduino web server can be made to automatically update the status of a switch on the web page. The button on the web page used to make the AJAX call from the previous part of this tutorial is no longer needed.

Before continuing with this part of the tutorial, you will need to have completed the previous part of this tutorial and understand it.

This video shows the Arduino web server displaying the switch status automatically using AJAX.

Arduino AJAX Web Server Sketch

Use the same hardware as the previous part of this tutorial – a push button switch interfaced to pin 3 of the Arduino with Ethernet shield.

Only three modifications need to be made to the previous sketch (eth_websrv_AJAX_switch) to automate the AJAX call that updates the switch status on the web page.

The modified sketch is shown here:

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

String HTTP_req;          // stores the HTTP request

void setup()
{
    Ethernet.begin(mac, ip);  // initialize Ethernet device
    server.begin();           // start to listen for clients
    Serial.begin(9600);       // for diagnostics
    pinMode(3, INPUT);        // switch is attached to Arduino pin 3
}
```

```
void loop()
{
    EthernetClient client = server.available();  // try to get client

    if (client) {  // got client?
        boolean currentLineIsBlank = true;
        while (client.connected()) {
            if (client.available()) {   // client data available to read
                char c = client.read(); // read 1 byte (character) from client
                HTTP_req += c;  // save the HTTP request 1 char at a time
                // last line of client request is blank and ends with \n
                // respond to client only after last line received
                if (c == '\n' && currentLineIsBlank) {
                    // send a standard http response header
                    client.println("HTTP/1.1 200 OK");
                    client.println("Content-Type: text/html");
                    client.println("Connection: keep-alive");
                    client.println();
                    // AJAX request for switch state
                    if (HTTP_req.indexOf("ajax_switch") > -1) {
                        // read switch state and send appropriate paragraph text
                        GetSwitchState(client);
                    }
                    else {  // HTTP request for web page
                        // send web page - contains JavaScript with AJAX calls
                        client.println("<!DOCTYPE html>");
                        client.println("<html>");
                        client.println("<head>");
                        client.println("<title>Arduino Web Page</title>");
                        client.println("<script>");
                        client.println("function GetSwitchState() {");
                        client.println("nocache = \"&nocache=\"\
                                        + Math.random() * 1000000;");
                        client.println("var request = new XMLHttpRequest();");
                        client.println("request.onreadystatechange = function() {");
                        client.println("if (this.readyState == 4) {");
                        client.println("if (this.status == 200) {");
                        client.println("if (this.responseText != null) {");
                        client.println("document.getElementById(\"switch_txt\")\
.innerHTML = this.responseText;");
                        client.println("}}}}");
                        client.println(
                        "request.open(\"GET\", \"ajax_switch\" + nocache, true);");
                        client.println("request.send(null);");
                        client.println("setTimeout('GetSwitchState()', 1000);");
                        client.println("}");
                        client.println("</script>");
                        client.println("</head>");
```

```
                client.println("<body onload=\"GetSwitchState()\">");
                client.println("<h1>Arduino AJAX Switch Status</h1>");
                client.println(
                "<p id=\"switch_txt\">Switch state: Not requested...</p>");
                client.println("</body>");
                client.println("</html>");
              }
              // display received HTTP request on serial port
              Serial.print(HTTP_req);
              HTTP_req = "";          // finished with request, empty string
              break;
            }
            // every line of text received from the client ends with \r\n
            if (c == '\n') {
              // last character on line of received text
              // starting new line with next character read
              currentLineIsBlank = true;
            }
            else if (c != '\r') {
              // a text character was received from client
              currentLineIsBlank = false;
            }
          } // end if (client.available())
        } // end while (client.connected())
        delay(1);      // give the web browser time to receive the data
        client.stop(); // close the connection
    } // end if (client)
}

// send the state of the switch to the web browser
void GetSwitchState(EthernetClient cl)
{
    if (digitalRead(3)) {
        cl.println("Switch state: ON");
    }
    else {
        cl.println("Switch state: OFF");
    }
}
```

Modification to HTML and JavaScript

The image below shows the modifications that were made to the HTML file that the Arduino sketch sends to the web browser (this file is sent line by line using client.println() in the sketch).

Web Page Button Code

Firstly, the code that creates a button on the web page has been removed as the button is no longer needed. It can be seen commented out in the above image.

```
<!DOCTYPE html>
<html>
<head>
    <title>Arduino Web Page</title>
    <script>
        function GetSwitchState()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseText != null) {
                            document.getElementById("switch_txt").innerHTML = this.responseText;
                        }
                    }
                }
            }
            request.open("GET", "ajax_switch" + nocache, true);
            request.send(null);
            setTimeout('GetSwitchState()', 1000); // new <--- added this line
        }
    </script>
</head>
<body onload="GetSwitchState()">    <!-- new --> <--- modified this line
    <h1>Arduino AJAX Switch Status</h1>
    <p id="switch_txt">Switch state: Not requested...</p>
    <!--<button type="button" onclick="GetSwitchState()">Get Switch State</button>--> removed
</body>
</html>
```

Calling the GetSwitchState() Function

The GetSwitchState() function that was previously being called each time the button was pressed is now being called when the page is loaded. This is done by calling the function when the page load event occurs by modifying the **<body>** tag of the HTML: **<body onload="GetSwitchState()">**

This is added to the Arduino sketch with the following line of code: **client.println("<body onload=\"GetSwitchState()\">");**

Making the AJAX Call Every Second

The GetSwitchState() function would only be called once when the web page loads, unless we change the code to periodically call this function.

The following line of code is added to the bottom of the GetSwitchState() function to make sure that this function is called every second: **setTimeout('GetSwitchState()', 1000);**

What this line of JavaScript code does is call GetSwitchState() every 1000 milliseconds (every second). An AJAX call is therefore made every one second which fetches the status of the switch and updates it on the web page.

This code is added to the web page by adding this line to the Arduino sketch: **client.println("setTimeout('GetSwitchState()', 1000);");**

Analog Inputs and Switches using AJAX

Part 9 of the Arduino Ethernet Shield Web Server Tutorial

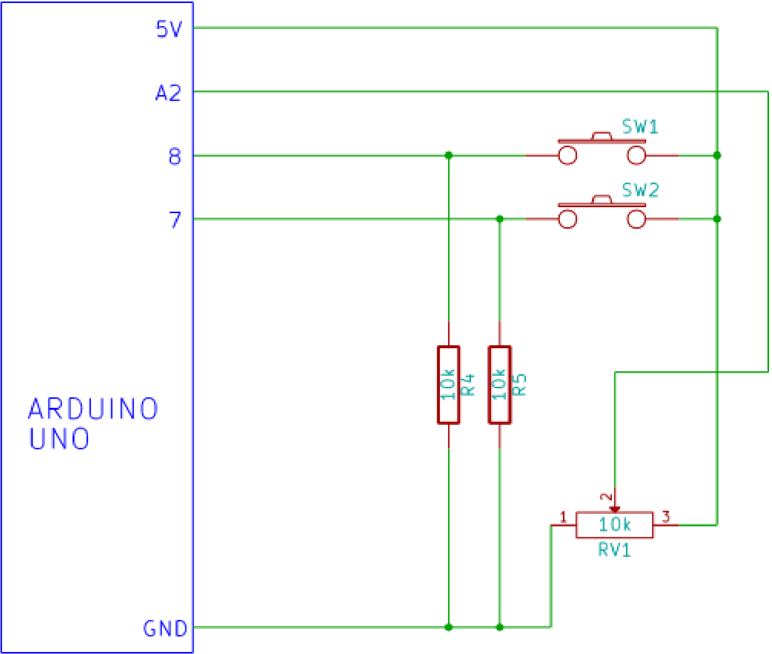
Updating the status of more than one switch that is interfaced to the Arduino web server, as well as showing the value of one of the analog inputs.

JavaScript is used to make AJAX calls to request the switch status and analog value from the web server.

This video shows the switches and analog input updated on the web page without flicker. Only parts of the web page are updated using AJAX.

Circuit Diagram

The circuit diagram below shows how the switches are interfaced to the Arduino (with Ethernet shield plugged into it). A potentiometer is interfaced to analog input A2 so that the value on A2 can be changed and updated on the web page.



The Sketch

The Arduino sketch is a modified version of the sketch from the previous tutorial.

```
#include <SPI.h>
#include <Ethernet.h>

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(10, 0, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80

String HTTP_req;          // stores the HTTP request
```

```
void setup()
{
   Ethernet.begin(mac, ip);  // initialize Ethernet device
   server.begin();          // start to listen for clients
   Serial.begin(9600);      // for diagnostics
   pinMode(7, INPUT);       // switch is attached to Arduino pin 7
   pinMode(8, INPUT);       // switch is attached to Arduino pin 8
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
      while (client.connected()) {
         if (client.available()) {   // client data available to read
            char c = client.read(); // read 1 byte (character) from client
            HTTP_req += c;  // save the HTTP request 1 char at a time
            // last line of client request is blank and ends with \n
            // respond to client only after last line received
            if (c == '\n' && currentLineIsBlank) {
               // send a standard http response header
               client.println("HTTP/1.1 200 OK");
               client.println("Content-Type: text/html");
               client.println("Connection: keep-alive");
               client.println();
               // AJAX request for switch state
               if (HTTP_req.indexOf("ajax_switch") > -1) {
                  // read switch state and analog input
                  GetAjaxData(client);
               }
               else {  // HTTP request for web page
                  // send web page - contains JavaScript with AJAX calls
                  client.println("<!DOCTYPE html>");
                  client.println("<html>");
                  client.println("<head>");
                  client.println("<title>Arduino Web Page</title>");
                  client.println("<script>");
                  client.println("function GetSwitchAnalogData() {");
                  client.println(
                     "nocache = \"&nocache=\" + Math.random() * 1000000;");
                  client.println("var request = new XMLHttpRequest();");
                  client.println("request.onreadystatechange = function() {");
                  client.println("if (this.readyState == 4) {");
                  client.println("if (this.status == 200) {");
                  client.println("if (this.responseText != null) {");
```

```
                client.println("document.getElementById(\"sw_an_data\")\
.innerHTML = this.responseText;");
                client.println("}}}}");
                client.println(
                "request.open(\"GET\", \"ajax_switch\" + nocache, true);");
                client.println("request.send(null);");
                client.println("setTimeout('GetSwitchAnalogData()', 1000);");
                client.println("}");
                client.println("</script>");
                client.println("</head>");
                client.println("<body onload=\"GetSwitchAnalogData()\">");
                client.println("<h1>Arduino AJAX Input</h1>");
                client.println("<div id=\"sw_an_data\">");
                client.println("</div>");
                client.println("</body>");
                client.println("</html>");
              }
              // display received HTTP request on serial port
              Serial.print(HTTP_req);
              HTTP_req = "";          // finished with request, empty string
              break;
            }
            // every line of text received from the client ends with \r\n
            if (c == '\n') {
              // last character on line of received text
              // starting new line with next character read
              currentLineIsBlank = true;
            }
            else if (c != '\r') {
              // a text character was received from client
              currentLineIsBlank = false;
            }
          } // end if (client.available())
        } // end while (client.connected())
        delay(1);      // give the web browser time to receive the data
        client.stop(); // close the connection
    } // end if (client)
}

// send the state of the switch to the web browser
void GetAjaxData(EthernetClient cl)
{
    int analog_val;

    if (digitalRead(7)) {
        cl.println("<p>Switch 7 state: ON</p>");
    }
    else {
```

```
      cl.println("<p>Switch 7 state: OFF</p>");
    }
    if (digitalRead(8)) {
      cl.println("<p>Switch 8 state: ON</p>");
    }
    else {
      cl.println("<p>Switch 8 state: OFF</p>");
    }
    // read analog pin A2
    analog_val = analogRead(2);
    cl.print("<p>Analog A2: ");
    cl.print(analog_val);
    cl.println("</p>");
}
```

Web Page Code

The above sketch produces the following HTML code:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino Web Page</title>
        <script>
            function GetSwitchAnalogData() {
                nocache = "&nocache=" + Math.random() * 1000000;
                var request = new XMLHttpRequest();
                request.onreadystatechange = function() {
                    if (this.readyState == 4) {
                        if (this.status == 200) {
                            if (this.responseText != null) {
                                document.getElementById("sw_an_data").innerHTML = this.responseText;
                            }
                        }
                    }
                }
                request.open("GET", "ajax_switch" + nocache, true);
                request.send(null);
                setTimeout('GetSwitchAnalogData()', 1000);
            }
        </script>
    </head>
    <body onload="GetSwitchAnalogData()">
        <h1>Arduino AJAX Input</h1>
        <div id="sw_an_data">
        </div>
    </body>
</html>
```

HTML Code Produced by Arduino Sketch - click for a bigger image

Modifications to the Sketch

Arduino pins 7 and 8 are both configured as inputs in the **setup()** part of the sketch.

The JavaScript function that handles the AJAX call has been renamed. The Arduino function that responds to the AJAX call has also been renamed.

A HTML <div> has been created below the H1 header in the HTML code and given the id "sw_an_data". The div is invisible on the page, but it serves as a place for the JavaScript to put the information (switch and analog values) sent back from the Arduino.

Sending the Request for Data from the Browser

The JavaScript function **GetSwitchAnalogData()** is called every second. Every second, it sends a GET request to the Arduino web server.

Receiving and Processing the AJAX Request on the Arduino

When the Arduino receives the AJAX request, it runs the **GetAjaxData()** function. This function reads the state of the two switches and sends the switches' statuses (ON or OFF) back to the web browser. The function also reads the value on the A2 analog pin and sends the value back to the browser.

Displaying the New Data in the Web Browser

When the web browser receives the data requested from the Arduino, it simply inserts it into the div that has the ID sw_an_data.

Arduino SD Card Web Server Linking Pages

Part 10 of the Arduino Ethernet Shield Web Server Tutorial

This part of the Arduino Ethernet shield web server tutorial shows how to create links between web pages that are hosted on the micro SD card of the Arduino web server.

These are links on a hosted web page that can be clicked in order to go to or open a different web page.

Creating Links in HTML

Links are created in HTML by using the HTML <a> tag. Text between the opening <a> tag and closing tag becomes a clickable link on the web page.

The value of the **href** attribute of the <a> tag must contain the file name of the web page that is linked to, e.g.:

```
<p>Go to <a href="page2.htm">page 2</a>.</p>
```

The above line of HTML will create a paragraph of text with the **page 2** part of the paragraph becoming a link to a file called **page2.htm**.

The file **page2.htm** must exist and also be in the same directory as the page that contains the link to it.

Example HTML Files

Two HTML files will be used as examples in this part of the tutorial. They must be saved to the micro SD card and the micro SD card must be plugged into the Ethernet shield.

The main page that will be loaded first from the server is made from the following HTML code (file name is index.htm):

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
    </head>
    <body>
        <h1>Arduino SD Card Page with Link</h1>
        <p>Go to <a href="page2.htm">page 2</a>.</p>
    </body>
</html>
```

The above page links to a second page called page2.htm:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page 2</title>
    </head>
    <body>
        <h1>Arduino SD Card Page 2</h1>
        <p>Go back to <a href="index.htm">main page</a>.</p>
    </body>
</html>
```

page2.htm links back to the main page **index.htm**.

Create the above two files (index.htm and page2.htm) and copy them to your micro SD card. Insert the micro SD card into the Ethernet shield micro SD card holder.

These pages can be tested on a computer (with the two files in the same folder on the hard-drive) by opening index.htm in a browser and clicking the link. page2.htm should open when the link is clicked. Clicking the link on page2.htm should send the browser back to index.htm.

HTTP Page Requests

When a web browser first requests a page from the Arduino web server, it sends an HTTP request similar to this:

```
GET / HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:19.0)
Gecko/20100101 Firefox/19.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

We have already seen this HTTP request in previous parts of this tutorial.

When the link on the page is clicked (the link on the index.htm page to the page2.htm page in our example), the web browser sends the following HTTP request:

```
GET /page2.htm HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:19.0)
Gecko/20100101 Firefox/19.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
```

```
Referer: http://10.0.0.20/
Connection: keep-alive
```

So the initial HTTP request contains a GET request for the root file: **GET /** (this would be our index.htm file).

When the link is clicked, the request is now for a specific page: **GET /page2.htm** – now we know that we must check the HTTP request to see whether it is requesting the root file or a specific file that was linked to. This check will be done in the Arduino sketch.

Arduino Sketch for Linked Web Pages on Web Server

The sketch below is a modified version of the first SD card web server from part 4 of this series.

The **eth_websrv_SD_link** Arduino sketch:

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>

// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   20

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20);    // IP address, may need to change depending on
network
EthernetServer server(80);     // create a server at port 80
File webFile;                  // handle to files on SD card
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null
terminated string
char req_index = 0;            // index into HTTP_req buffer

void setup()
{
  // disable Ethernet chip
  pinMode(10, OUTPUT);
  digitalWrite(10, HIGH);

  Serial.begin(9600);      // for debugging

  // initialize SD card
  Serial.println("Initializing SD card...");
  if (!SD.begin(4)) {
    Serial.println("ERROR - SD card initialization failed!");
    return;    // init failed
  }
```

```
    Serial.println("SUCCESS - SD card initialized.");
    // check for index.htm file
    if (!SD.exists("index.htm")) {
        Serial.println("ERROR - Can't find index.htm file!");
        return;  // can't find index file
    }
    Serial.println("SUCCESS - Found index.htm file.");

    Ethernet.begin(mac, ip);  // initialize Ethernet device
    server.begin();           // start to listen for clients
}

void loop()
{
    EthernetClient client = server.available();  // try to get client

    if (client) {  // got client?
        boolean currentLineIsBlank = true;
        while (client.connected()) {
            if (client.available()) {   // client data available to read
                char c = client.read(); // read 1 byte (character) from client
                // buffer first part of HTTP request in HTTP_req array (string)
                // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
                if (req_index < (REQ_BUF_SZ - 1)) {
                    HTTP_req[req_index] = c;          // save HTTP request character
                    req_index++;
                }
                Serial.print(c);    // print HTTP request character to serial monitor
                // last line of client request is blank and ends with \n
                // respond to client only after last line received
                if (c == '\n' && currentLineIsBlank) {
                    // send a standard http response header
                    client.println("HTTP/1.1 200 OK");
                    client.println("Content-Type: text/html");
                    client.println("Connnection: close");
                    client.println();
                    // open requested web page file
                    if (StrContains(HTTP_req, "GET / ")
                                 || StrContains(HTTP_req, "GET /index.htm")) {
                        webFile = SD.open("index.htm");        // open web page file
                    }
                    else if (StrContains(HTTP_req, "GET /page2.htm")) {
                        webFile = SD.open("page2.htm");        // open web page file
                    }
                    // send web page to client
                    if (webFile) {
                        while(webFile.available()) {
                            client.write(webFile.read());
```

```
          }
          webFile.close();
        }
        // reset buffer index and all buffer elements to 0
        req_index = 0;
        StrClear(HTTP_req, REQ_BUF_SZ);
        break;
      }
      // every line of text received from the client ends with \r\n
      if (c == '\n') {
        // last character on line of received text
        // starting new line with next character read
        currentLineIsBlank = true;
      }
      else if (c != '\r') {
        // a text character was received from client
        currentLineIsBlank = false;
      }
    } // end if (client.available())
  } // end while (client.connected())
  delay(1);      // give the web browser time to receive the data
  client.stop(); // close the connection
 } // end if (client)
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
  for (int i = 0; i < length; i++) {
    str[i] = 0;
  }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
{
  char found = 0;
  char index = 0;
  char len;

  len = strlen(str);

  if (strlen(sfind) > len) {
    return 0;
  }
  while (index < len) {
```

```
        if (str[index] == sfind[found]) {
            found++;
            if (strlen(sfind) == found) {
                return 1;
            }
        }
        else {
            found = 0;
        }
        index++;
    }
    return 0;
}
```

NOTE: The IP address is set to **192.168.0.20** in this sketch and not **10.0.0.20** as in other sketches in this tutorial, so change it for your system if necessary.

The changes to the original SD card sketch from part 4 are described below.

HTTP Request

The sketch was modified to store the HTTP request from the web browser in the string **HTTP_req**. This string can then be searched to find out which page is being requested.

The HTTP request is sent out of the serial port and can be viewed in the Arduino serial monitor window for diagnostics and debugging purposes.

Sending the Correct Web Page

After the Arduino has received the HTTP request from the browser, it responds with a standard HTTP header and then sends the requested web page.

The code that selects which web page to send is shown here:

```
// open requested web page file
if (StrContains(HTTP_req, "GET / ")
             || StrContains(HTTP_req, "GET /index.htm")) {
    webFile = SD.open("index.htm");   // open web page file
}
else if (StrContains(HTTP_req, "GET /page2.htm")) {
    webFile = SD.open("page2.htm");  // open web page file
}
```

All this code does is open either index.htm or page2.htm from the SD card. The code that sends the file is the same as the code from part 4 of this series.

The code to select the correct file looks at the received HTTP request using the **StrContains()** function. **HTTP_req** is the string in our sketch that contains the HTTP request. If the HTTP request contains "GET / ", then this is a request for our root file index.htm.

If the HTTP request string contains "GET /page2.htm", then page2.htm will be opened and sent to the web browser.

When the link on page2.htm is clicked, it links back to index.htm and not /. This is the reason for checking if the HTTP request contains "GET / " or "GET /index.htm" in the first **if** statement in the above code listing.

Sketch Improvements

The above sketch is used to demonstrate the mechanism for opening page links on the Arduino web server, so was kept simple. Some improvements that could be made to the code would be firstly to extract the file name after the GET in the HTTP request and then open the file without checking for the specific name in the code. A second improvement would be to handle the case where a page is requested by the browser, but it does not exist on the SD card.

Arduino SD Card Web Server Displaying Images

Part 11 of the Arduino Ethernet Shield Web Server Tutorial
A page hosted by the Arduino web server on the SD card contains an image. This tutorial shows how to insert a JPEG image into a HTML web page and how to send the image to the web browser when an HTTP request for the image is received by the web server.

Uses the Arduino Uno with Ethernet shield and micro SD card.

HTML for Displaying an Image

The HTML **<img>** tag is used to insert an image into a web page. The web pages from the previous part of this tutorial series are used again. The **index.htm** file is modified to add an image – the HTML for this file is shown below.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
    </head>
    <body>
        <h1>Arduino SD Card Page with Image and Link</h1>
        <img src="pic.jpg" />
        <p>Go to <a href="page2.htm">page 2</a>.</p>
    </body>
</html>
```

In the above HTML code, an image called **pic.jpg** is inserted into the web page using the following line of HTML code:

```
<img src="pic.jpg" />
```

The **src** attribute is used to specify the name of the image to display.

Source Code

The three files for this example can be downloaded and copied to a micro SD card that will be inserted into the card slot of the Arduino Ethernet shield.

SD_card_image.zip (8.2 kB) – contains index.htm, page2.htm and pic.jpg used in this part of the tutorial.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page 2</title>
    </head>
```

```
    <body>
        <h1>Arduino SD Card Page 2</h1>
        <p>Go back to <a href="index.htm">main page</a>.</p>
    </body>
</html>
```

HTTP Requests

When connecting to the Arduino web server in this example, the web browser will first send an HTTP request to the server as normal. After the web browser has received the web page, it will find that the web page contains an image. It will then send a second HTTP request for the image.

Arduino Sketch

The Arduino sketch for this example is called **eth_websrv_SD_image** and is shown below. It is a modified version of the sketch from the previous part of this tutorial series.

```
#include <SPI.h>

#include <Ethernet.h>
#include <SD.h>

// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   20

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80
File webFile;
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null terminated string
char req_index = 0;              // index into HTTP_req buffer

void setup()
{
   // disable Ethernet chip
   pinMode(10, OUTPUT);
   digitalWrite(10, HIGH);

   Serial.begin(9600);       // for debugging

   // initialize SD card
   Serial.println("Initializing SD card...");
   if (!SD.begin(4)) {
      Serial.println("ERROR - SD card initialization failed!");
      return;    // init failed
```

```
    }
    Serial.println("SUCCESS - SD card initialized.");
    // check for index.htm file
    if (!SD.exists("index.htm")) {
        Serial.println("ERROR - Can't find index.htm file!");
        return;  // can't find index file
    }
    Serial.println("SUCCESS - Found index.htm file.");

    Ethernet.begin(mac, ip);  // initialize Ethernet device
    server.begin();           // start to listen for clients
}

void loop()
{
    EthernetClient client = server.available();  // try to get client

    if (client) {  // got client?
        boolean currentLineIsBlank = true;
        while (client.connected()) {
            if (client.available()) {   // client data available to read
                char c = client.read(); // read 1 byte (character) from client
                // buffer first part of HTTP request in HTTP_req array (string)
                // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
                if (req_index < (REQ_BUF_SZ - 1)) {
                    HTTP_req[req_index] = c;          // save HTTP request character
                    req_index++;
                }
                // print HTTP request character to serial monitor
                Serial.print(c);
                // last line of client request is blank and ends with \n
                // respond to client only after last line received
                if (c == '\n' && currentLineIsBlank) {
                    // open requested web page file
                    if (StrContains(HTTP_req, "GET / ")
                                 || StrContains(HTTP_req, "GET /index.htm")) {
                        client.println("HTTP/1.1 200 OK");
                        client.println("Content-Type: text/html");
                        client.println("Connnection: close");
                        client.println();
                        webFile = SD.open("index.htm");        // open web page file
                    }
                    else if (StrContains(HTTP_req, "GET /page2.htm")) {
                        client.println("HTTP/1.1 200 OK");
                        client.println("Content-Type: text/html");
                        client.println("Connnection: close");
                        client.println();
                        webFile = SD.open("page2.htm");        // open web page file
```

```
            }
            else if (StrContains(HTTP_req, "GET /pic.jpg")) {
                webFile = SD.open("pic.jpg");
                if (webFile) {
                    client.println("HTTP/1.1 200 OK");
                    client.println();
                }
            }
            if (webFile) {
                while(webFile.available()) {
                    client.write(webFile.read()); // send web page to client
                }
                webFile.close();
            }
            // reset buffer index and all buffer elements to 0
            req_index = 0;
            StrClear(HTTP_req, REQ_BUF_SZ);
            break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
            // last character on line of received text
            // starting new line with next character read
            currentLineIsBlank = true;
        }
        else if (c != '\r') {
            // a text character was received from client
            currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
  for (int i = 0; i < length; i++) {
    str[i] = 0;
  }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
```

```
{
  char found = 0;
  char index = 0;
  char len;

  len = strlen(str);

  if (strlen(sfind) > len) {
    return 0;
  }
  while (index < len) {
    if (str[index] == sfind[found]) {
      found++;
      if (strlen(sfind) == found) {
        return 1;
      }
    }
    else {
      found = 0;
    }
    index++;
  }

  return 0;
}
```

The sketch works the same way as the sketch from the previous part of this tutorial, except for the following code which handles the JPEG image:

```
else if (StrContains(HTTP_req, "GET /pic.jpg")) {
    webFile = SD.open("pic.jpg");
    if (webFile) {
        client.println("HTTP/1.1 200 OK");
        client.println();
    }
}
```

This code checks to see if the HTTP request from the web browser is requesting the JPEG image pic.jpg.

If the request for the image is received and it can be opened from the SD card, a OK response is sent back to the web browser. The JPEG file is then sent using the same code that sends back an HTML page.

Again, as in the previous part of this tutorial, the code was made very basic for teaching purposes. It does not handle cases where the resource (HTML file or image file) can't be found on the SD card. It also specifically only handles an image with the name "pic.jpg".

For practical use, it would be better to obtain the requested HTML page name or image file name from the HTTP request and then try to find it on the SD card. Code

should be in place to handle the case where the file can not be found on the SD card.

CSS Introduction

Part 12 of the Arduino Ethernet Shield Web Server Tutorial

We first looked at HTML in this tutorial which has to do with structuring the content of a web page into paragraphs, headings, form controls, etc.

We now look at CSS (Cascading Style Sheets). CSS controls the appearance of the content on a web page. CSS acts on the HTML tags to change attributes of the text or elements between the HTML tags. For example, the font type, colour and size of a paragraph of text can be changed. CSS can also be used to position HTML elements on a web page.

In this part of the tutorial, we will look at only the basics of CSS so that the newcomer to CSS can get an idea of what CSS can do and what CSS looks like. Further CSS will be explained as it is used in this tutorial.

A CSS Example

In this example, CSS markup is used to style a page so that it appears as follows:

Arduino SD Card Page with CSS

Welcome to the Arduino web page with CSS styling.

This text is red.

This paragraph has one word that uses red text.

A Web Page Styled with CSS

HTML and CSS Markup

The HTML and CSS markup that produces the above web page can be seen below.

Including the CSS

In this example the actual CSS markup is included in the HTML page. The CSS is inserted between opening and closing <style> tags. The style tags are placed in the <head> part of the HTML file.

There are two other methods of including CSS in an HTML file: 1) In-line – the CSS is inserted into the HTML tags. 2) An external style sheet – all the CSS is written in an external file and included at the top of the HTML file.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            h1 {
                font-family: courier, courier-new, serif;
                font-size: 20pt;
                color: blue;
                border-bottom: 2px solid blue;
            }
            p {
                font-family: arial, verdana, sans-serif;
                font-size: 12pt;
                color: #6B6BD7;
            }
            .red_txt {
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with CSS</h1>
        <p>Welcome to the Arduino web page with CSS styling.</p>
        <p class="red_txt">This text is red.</p>
        <p>This paragraph has one word that uses <span class="red_txt">red</span> text.</p>
    </body>
</html>
```

Examining the CSS

The CSS that is used in this example is shown on its own here:

```
h1 {
    font-family: courier, courier-new, serif;
    font-size: 20pt;
    color: blue;
    border-bottom: 2px solid blue;
}
p {
    font-family: arial, verdana, sans-serif;
    font-size: 12pt;
    color: #6B6BD7;
}
.red_txt {
    color: red;
}
```

Header Text

The first part of the CSS applies styles to the <h1> part of the HTML code. The styles between the opening and closing braces after **h1** in the above listing will be applied to every h1 header on the web page.

The line of CSS code below sets the font type to courier. If courier is not found on

the system, then courier-new will be used. If courier-new is not found on the system then any serif font will be used.

```
font-family: courier, courier-new, serif;
```

The next two lines set the size of the h1 font and the font colour.

The last style that is applied to h1 headers is to put a two pixel wide (2px) solid blue line below the header text.

Paragraph Text

The next style in the CSS markup is applied to paragraph text (<p>). The styles between the opening and closing braces after **p** are applied to all paragraph text on the web page.

A font type and size are specified first. The colour of the paragraph text is specified using a RGB hexadecimal number.

Overriding a Style

In the HTML, a paragraph has been given a class name. This name is any name that the person writing the HTML and CSS chooses:

```
<p class="red_txt">This text is red.</p>
```

By creating a CSS class style called red_txt, the paragraph style can be overridden for all paragraphs that are marked as of the red_txt class.

```
.red_txt {
    color: red;
}
```

In this case, only the colour of the paragraph is overridden because it is the only style specified in the CSS red_txt class.

When writing a CSS class, the name must start with a full-stop as in **.red_txt** and as shown above.

Changing the Style of a Single Word

To change the style of a single word in a paragraph, the word must first be isolated by using HTML code. The following line of HTML uses the <span> tag to isolate a single word. It then applies the same style from the red_txt class to the single word.

```
<p>This paragraph has one word that uses <span
class="red_txt" >red </span> text.</p>
```

Further CSS Learning

This has been a very brief introduction to CSS and was intended only to show you what CSS is, what it does and what it looks like.

There are very many more styles that can be applied to a large range of HTML tags. In fact there are whole books dedicated to CSS.

If you would like to learn more about CSS, then search for a more in-depth CSS tutorial on the Internet or pick up a good CSS book.

Running the CSS Example

To load the above CSS example to the Arduino web server, just put the HTML and CSS code into a file called index.htm and copy it to a micro SD card. Insert the micro SD card into the Arduino Ethernet shield card socket and then load the sketch from the Arduino SD card web server.

To save you from having to type out the above code, it has been included here so that it can be copied and pasted:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            h1 {
                font-family: courier, courier-new, serif;
                font-size: 20pt;
                color: blue;
                border-bottom: 2px solid blue;
            }
            p {
                font-family: arial, verdana, sans-serif;
                font-size: 12pt;
                color: #6B6BD7;
            }
            .red_txt {
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with CSS</h1>
        <p>Welcome to the Arduino web page with CSS styling.</p>
        <p class="red_txt">This text is red.</p>
        <p>This paragraph has one word that uses <span class="red_txt">red</span> text.</p>
    </body>
</html>
```

Arduino SD Card Ajax Web Server

Part 13 of the Arduino Ethernet Shield Web Server Tutorial

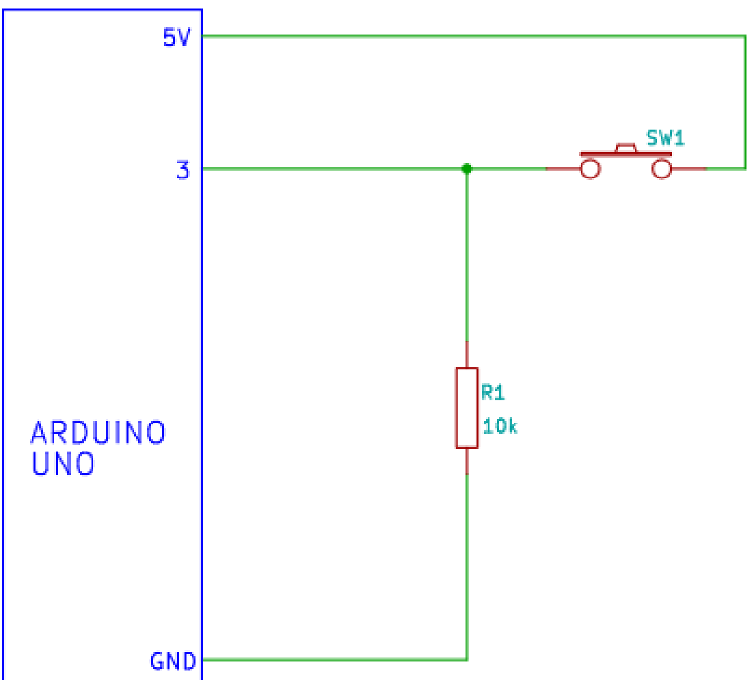
An Arduino Ethernet shield web server that hosts a web page on the SD card. The web page displays the status of a switch and uses Ajax to update the status of the switch.

In previous parts of this tutorial, an SD card hosted web page was never used to display the status of Arduino inputs – all the web pages displaying I/O were part of the Arduino sketch.

This part of the tutorial now displays an Arduino input on an SD card hosted web page.

Circuit Diagram

A switch is interfaced to pin 3 of the Arduino for this example.



Code and Web Page

The code (Arduino sketch) and web page for this part of the tutorial are basically a combination of part 4 (Arduino SD card web server) and part 8 (reading switch status automatically using Ajax) of this tutorial.

No video has been included with this tutorial as the output will look the same as part 8 of this tutorial, but with the title and heading text changed.

Web Page

The web page consisting of HTML and JavaScript (to implement Ajax) is shown below:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page using Ajax</title>
        <script>
        function GetSwitchState()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseText != null) {
                            document.getElementById("switch_txt").innerHTML = this.responseText;
                        }
                    }
                }
            }
            request.open("GET", "ajax_switch" + nocache, true);
            request.send(null);
            setTimeout('GetSwitchState()', 1000);
        }
    </script>
    </head>
    <body onload="GetSwitchState()">
        <h1>Arduino Switch State from SD Card Web Page using Ajax</h1>
        <p id="switch_txt">Switch state: Not requested...</p>
    </body>
</html>
```

Web Page Hosted on SD Card (index.htm)

This web page is saved to the micro SD card as **index.htm** – it is basically the same HTML/JavaScript that was produced by the Arduino sketch in part 8 (reading switch status automatically using Ajax), but with the title and heading text changed.

Copy and paste the web page from the listing below.

Arduino Sketch

The Arduino sketch for this part of the tutorial is shown below. It requires the above HTML/JavaScript to be available on the micro SD card in the index.htm file.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>

// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   40

// MAC address from Ethernet shield sticker under board
```

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20); // IP address, may need to change depending on
network
EthernetServer server(80);  // create a server at port 80
File webFile;
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null
terminated string
char req_index = 0;              // index into HTTP_req buffer

void setup()
{
   // disable Ethernet chip
   pinMode(10, OUTPUT);
   digitalWrite(10, HIGH);

   Serial.begin(9600);       // for debugging

   // initialize SD card
   Serial.println("Initializing SD card...");
   if (!SD.begin(4)) {
      Serial.println("ERROR - SD card initialization failed!");
      return;    // init failed
   }
   Serial.println("SUCCESS - SD card initialized.");
   // check for index.htm file
   if (!SD.exists("index.htm")) {
      Serial.println("ERROR - Can't find index.htm file!");
      return;  // can't find index file
   }
   Serial.println("SUCCESS - Found index.htm file.");
   pinMode(3, INPUT);        // switch is attached to Arduino pin 3

   Ethernet.begin(mac, ip);  // initialize Ethernet device
   server.begin();           // start to listen for clients
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
      while (client.connected()) {
         if (client.available()) {   // client data available to read
            char c = client.read(); // read 1 byte (character) from client
            // buffer first part of HTTP request in HTTP_req array (string)
            // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
            if (req_index < (REQ_BUF_SZ - 1)) {
```

```
          HTTP_req[req_index] = c;          // save HTTP request character
          req_index++;
        }
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
           client.println("Connection: keep-alive");
          client.println();
          // Ajax request
          if (StrContains(HTTP_req, "ajax_switch")) {
            // read switch state and send appropriate paragraph text
            GetSwitchState(client);
          }
          else {  // web page request
            // send web page
            webFile = SD.open("index.htm");        // open web page file
            if (webFile) {
              while(webFile.available()) {
                client.write(webFile.read()); // send web page to client
              }
              webFile.close();
            }
          }
          // display received HTTP request on serial port
          Serial.println(HTTP_req);
          // reset buffer index and all buffer elements to 0
          req_index = 0;
          StrClear(HTTP_req, REQ_BUF_SZ);
          break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
          // last character on line of received text
          // starting new line with next character read
          currentLineIsBlank = true;
        }
        else if (c != '\r') {
          // a text character was received from client
          currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
```

```
}

// send the state of the switch to the web browser
void GetSwitchState(EthernetClient cl)
{
  if (digitalRead(3)) {
    cl.println("Switch state: ON");
  }
  else {
    cl.println("Switch state: OFF");
  }
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
  for (int i = 0; i < length; i++) {
    str[i] = 0;
  }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
{
  char found = 0;
  char index = 0;
  char len;

  len = strlen(str);

  if (strlen(sfind) > len) {
    return 0;
  }
  while (index < len) {
    if (str[index] == sfind[found]) {
      found++;
      if (strlen(sfind) == found) {
        return 1;
      }
    }
    else {
      found = 0;
    }
    index++;
  }
```

```
    return 0;
}
```

How the Sketch Works

This sketch works in the same way as the sketch from part 8 (reading switch status automatically using Ajax), except that instead of sending the web page line by line from the Arduino sketch code, the web page is sent from the index.htm file on the SD card.

Because we are using Ajax, the web page (when loaded in the web browser) sends the same request for the switch status as the sketch in part 8 of this tutorial. If we were not using Ajax, then the Arduino would need to read the index.htm file from the SD card and modify the part that shows the switch status, then send back the whole web page with the modified part – depending on if the switch is on or off.

Arduino Inputs using Ajax with XML on the Arduino Web Server

Part 14 of the Arduino Ethernet Shield Web Server Tutorial
The Arduino web server hosts a web page (stored on the SD card) that displays the status of two push button switches and an analog (analogue) input.

The status of the two switches and the analog input are updated on the web page using Ajax. An XML file containing the switch statuses and the analog value is sent from the Arduino to the web browser.

This example produces the same output on the web page (with only the text changed) as part 9 of this tutorial – Analog Inputs and Switches using AJAX, with the following changes:

Switch statuses and an analog value are sent in an XML file and not as a block of HTML.
JavaScript responseXML is used instead of responseText to get the received values from the Arduino out of the XML file.

The values from the XML file (Arduino inputs) are inserted into HTML paragraphs in the web page instead of replacing the entire paragraph.

The web page is stored on the micro SD card of the Ethernet shield.
Why use Ajax with XML?

The advantage of using Ajax with an XML file is that individual values can easily be extracted by JavaScript on the web page, rather than having to write JavaScript code to extract values from a text file.

XML File Structure

An XML file uses tags like HTML or XHTML. The file has an initial tag that identifies it as an XML file. A main user defined tag follows that contains all other tags for the file.

This listing shows the structure of the XML file used in this example:

```
<?xml version = "1.0" ?>
<inputs>
    <button1></button1>
    <button2></button2>
    <analog1></analog1>
</inputs>
```

The **inputs** tag and all other tags contained in it are user defined names. The above XML could also be created as follows:

```
<?xml version = "1.0" ?>
<inputs>
    <button></button>
```

```
    <button></button>
    <analog></analog>
</inputs>
```

This file shows a button type and analog type that can be used to contain any button state or analog value. By adding more <button> or more <analog> tags, the state of additional buttons or analog inputs can be added.

The difference between the above two files is that the first uses unique names for all tags, whereas the second uses the tags to identify an input type.

Arduino XML File

In this example the Arduino creates an XML file and inserts the status of the switches and the analog input between the tags. The XML file is sent to the web browser in response to an Ajax request for data.

The image below shows an example of the XML file sent from the Arduino.

```
<?xml version = "1.0" ?>
<inputs>
    <button1>ON</button1>
    <button2>OFF</button2>
    <analog1>394</analog1>
</inputs>
```

XML File Sent by Arduino

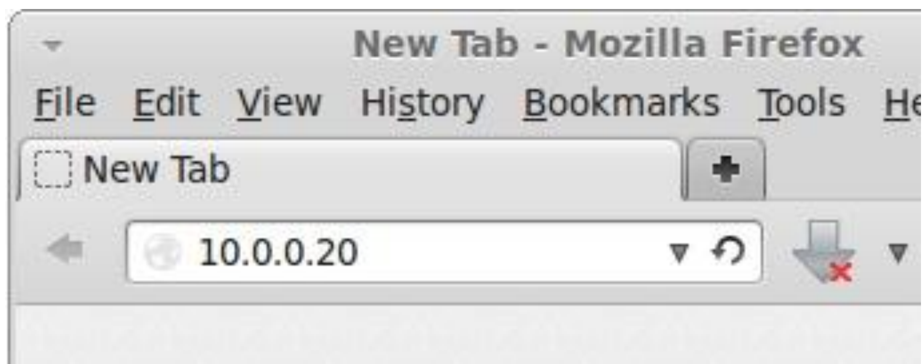
How Ajax with XML Works

If you have been following each part of this tutorial, then a lot of this will look familiar.

To update the Arduino input values on the web page, the following must occur:

1. Requesting a web page

As usual, the web browser is used to access the Arduino web server at the IP address that it has been set at in the Arduino sketch.



Connecting to the Arduino Web Server

This causes the web browser to sent an HTTP request:

```
GET / HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:19.0)
Gecko/20100101 Firefox/19.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

2. Sending the web page

The Arduino web server receives the above request and responds with an HTTP header followed by the web page:

```
HTTP/1.1 200 OK
Content-Type: text/html
Connection: keep-alive
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page using Ajax with XML</title>
        <script>
        function GetArduinoInputs()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseXML != null) {
                            // extract XML data from XML file (containing switch states and analog value)
                            document.getElementById("input1").innerHTML =
                                this.responseXML.getElementsByTagName('button1')[0].childNodes[0].nodeValue
                            document.getElementById("input2").innerHTML =
                                this.responseXML.getElementsByTagName('button2')[0].childNodes[0].nodeValue
                            document.getElementById("input3").innerHTML =
                                this.responseXML.getElementsByTagName('analog1')[0].childNodes[0].nodeValue
                        }
                    }
                }
            }
            request.open("GET", "ajax_inputs" + nocache, true);
            request.send(null);
            setTimeout('GetArduinoInputs()', 1000);
        }
    </script>
    </head>
    <body onload="GetArduinoInputs()">
        <h1>Arduino Inputs from SD Card Web Page using Ajax with XML</h1>
        <p>Button 1 (pin 7): <span id="input1">...</span></p>
        <p>Button 2 (pin 8): <span id="input2">...</span></p>
        <p>Analog (A2): <span id="input3">...</span></p>
    </body>
</html>
```

The Arduino reads the web page from the SD card and sends it to the web browser.

After receiving the web page, it will be displayed in the web browser.

The web page contains JavaScript that is used as part of the Ajax process.

Note that the content type in the HTTP header for the HTML web page is text/html.

3. Ajax request

The JavaScript code on the web page sends an Ajax request to the Arduino (and continues to send a request every second).

```
GET /ajax_inputs&nocache=299105.2747379479 HTTP/1.1
Host: 10.0.0.20
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:19.0) Gecko/20100101 Firefox/19.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0
.8
Accept-Language: en-ZA,en-GB;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip
```

4. The Arduino responds to the Ajax request

After receiving the request for the XML file, the Arduino responds with an HTTP header followed by the XML file which contains input values from the Arduino.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Connection: keep-alive
```

```
<?xml version = "1.0" ?>
<inputs>
    <button1>ON</button1>
    <button2>OFF</button2>
    <analog1>394</analog1>
</inputs>
```

Note that the content type in the HTTP header is now text/xml.

5. Displaying the data

Finally the JavaScript in the web page extracts the three values from the Arduino from the XML file and displays them on the web page.

Arduino Sketch and Web Page

Web Page

The Arduino hosts the following web page on the SD card:

This is basically the same web page as sent by the Arduino in part 9 of this tutorial, but with the following changes (besides the text changes):

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page using Ajax with XML</title>
        <script>
        function GetArduinoInputs()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseXML != null) {
                            // extract XML data from XML file (containing switch states and analog value)
                            document.getElementById("input1").innerHTML =
                                this.responseXML.getElementsByTagName('button1')[0].childNodes[0].nodeValue
                            document.getElementById("input2").innerHTML =
                                this.responseXML.getElementsByTagName('button2')[0].childNodes[0].nodeValue
                            document.getElementById("input3").innerHTML =
                                this.responseXML.getElementsByTagName('analog1')[0].childNodes[0].nodeValue
                        }
                    }
                }
            }
            request.open("GET", "ajax_inputs" + nocache, true);
            request.send(null);
            setTimeout('GetArduinoInputs()', 1000);
        }
    </script>
    </head>
    <body onload="GetArduinoInputs()">
        <h1>Arduino Inputs from SD Card Web Page using Ajax with XML</h1>
        <p>Button 1 (pin 7): <span id="input1">...</span></p>
        <p>Button 2 (pin 8): <span id="input2">...</span></p>
        <p>Analog (A2): <span id="input3">...</span></p>
    </body>
</html>
```

Arduino Web Page index.htm - click for a bigger image

Function

The JavaScript function in the web page has been renamed to GetArduinoInputs().

The function still sends out an Ajax request every second. It now sends ajax_inputs with the GET request.

Because an XML file is being sent back from the Arduino, the function now checks if responseXML contains data instead of responseText:

```
if (this.responseXML != null) {
```

The data is extracted from the received XML as explained shortly.

HTML

The HTML is modified to display three paragraphs of text, one each for each value sent from the Arduino. Each paragraph contains an HTML span, each span has a unique ID.

The JavaScript function will insert the extracted values from the XML file into each span. This will replace only the default text (...) in each paragraph with the value from the Arduino.

The function uses the following code to get hold of each span for inserting data (code for getting "input1" shown here):

```
document.getElementById("input1").innerHTML =
```

Extracting the XML Data

The XML data is extracted from the received XML file using the following line of code:

```
this.responseXML.getElementsByTagName('button1')[0].childNode
s[0].nodeValue;
```

In this code, this.responseXML is used instead of this.responseText as used in previous examples.

Now every tag in the XML can be accessed using this.responseXML.getElementsByTagName('button1') as can be seen in the JavaScript function.

If you refer back to the top of this part of the tutorial under XML File Structure, and the second XML file example, you will see that there can be tags with the same name. If we used this for the button tags, then each button tag value can be accessed as follows:

```
this.responseXML.getElementsByTagName('button')[0].childNodes
[0].nodeValue;
this.responseXML.getElementsByTagName('button')[1].childNodes
[0].nodeValue;
```

This is usefull if there were a number of buttons that you did not want to give unique tags to. The values can then also be accessed in the JavaScript by using a loop.

The button values will then be extracted in the order that they have been inserted into the file.

The number of buttons in the XML file can then be obtained by using:

```
this.responseXML.getElementsByTagName('button').length
```

Arduino Sketch

The Arduino sketch for this example is shown below.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>
```

```
// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   50

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80
File webFile;              // the web page file on the SD card
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null terminated string
char req_index = 0;              // index into HTTP_req buffer

void setup()
{
   // disable Ethernet chip
   pinMode(10, OUTPUT);
   digitalWrite(10, HIGH);

   Serial.begin(9600);       // for debugging

   // initialize SD card
   Serial.println("Initializing SD card...");
   if (!SD.begin(4)) {
      Serial.println("ERROR - SD card initialization failed!");
      return;    // init failed
   }
   Serial.println("SUCCESS - SD card initialized.");
   // check for index.htm file
   if (!SD.exists("index.htm")) {
      Serial.println("ERROR - Can't find index.htm file!");
      return;  // can't find index file
   }
   Serial.println("SUCCESS - Found index.htm file.");
   pinMode(7, INPUT);        // switch is attached to Arduino pin 7
   pinMode(8, INPUT);        // switch is attached to Arduino pin 8

   Ethernet.begin(mac, ip);  // initialize Ethernet device
   server.begin();           // start to listen for clients
}

void loop()
{
   EthernetClient client = server.available();  // try to get client

   if (client) {  // got client?
      boolean currentLineIsBlank = true;
      while (client.connected()) {
```

```
if (client.available()) {   // client data available to read
   char c = client.read(); // read 1 byte (character) from client
   // buffer first part of HTTP request in HTTP_req array (string)
   // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
   if (req_index < (REQ_BUF_SZ - 1)) {
      HTTP_req[req_index] = c;          // save HTTP request character
      req_index++;
   }
   // last line of client request is blank and ends with \n
   // respond to client only after last line received
   if (c == '\n' && currentLineIsBlank) {
      // send a standard http response header
      client.println("HTTP/1.1 200 OK");
      // remainder of header follows below, depending on if
      // web page or XML page is requested
      // Ajax request - send XML file
      if (StrContains(HTTP_req, "ajax_inputs")) {
         // send rest of HTTP header
         client.println("Content-Type: text/xml");
         client.println("Connection: keep-alive");
         client.println();
         // send XML file containing input states
         XML_response(client);
      }
      else {  // web page request
         // send rest of HTTP header
         client.println("Content-Type: text/html");
         client.println("Connection: keep-alive");
         client.println();
         // send web page
         webFile = SD.open("index.htm");        // open web page file
         if (webFile) {
            while(webFile.available()) {
               client.write(webFile.read()); // send web page to client
            }
            webFile.close();
         }
      }
      // display received HTTP request on serial port
      Serial.print(HTTP_req);
      // reset buffer index and all buffer elements to 0
      req_index = 0;
      StrClear(HTTP_req, REQ_BUF_SZ);
      break;
   }
   // every line of text received from the client ends with \r\n
   if (c == '\n') {
      // last character on line of received text
```

```
                // starting new line with next character read
                currentLineIsBlank = true;
            }
            else if (c != '\r') {
                // a text character was received from client
                currentLineIsBlank = false;
            }
        } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

// send the XML file with switch statuses and analog value
void XML_response(EthernetClient cl)
{
  int analog_val;

  cl.print("<?xml version = \"1.0\" ?>");
  cl.print("<inputs>");
  cl.print("<button1>");
  if (digitalRead(7)) {
    cl.print("ON");
  }
  else {
    cl.print("OFF");
  }
  cl.print("</button1>");
  cl.print("<button2>");
  if (digitalRead(8)) {
    cl.print("ON");
  }
  else {
    cl.print("OFF");
  }
  cl.print("</button2>");
  // read analog pin A2
  analog_val = analogRead(2);
  cl.print("<analog1>");
  cl.print(analog_val);
  cl.print("</analog1>");
  cl.print("</inputs>");
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
```

```
  for (int i = 0; i < length; i++) {
    str[i] = 0;
  }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
{
  char found = 0;
  char index = 0;
  char len;

  len = strlen(str);

  if (strlen(sfind) > len) {
    return 0;
  }
  while (index < len) {
    if (str[index] == sfind[found]) {
      found++;
      if (strlen(sfind) == found) {
        return 1;
      }
    }
    else {
      found = 0;
    }
    index++;
  }

  return 0;
}
```

This sketch is basically a modified version of the sketch from the previous part of this tutorial.

Creating the XML File

The XML_response() function takes care of generating and sending the XML file in the format that has already been explained.

The switches and analog values are inserted into the XML file and sent to the web browser.

HTTP Response

Because the HTTP response must send a different content type for the HTML page and XML file (text/html or text/xml), it has been split up in the sketch to send the correct file type in each HTTP header.

As with the previous part of this tutorial, the web page is stored on the SD card as index.htm and sent when the browser accesses the Arduino web server.

Running the Sketch

Wire up the push button switches and the potentiometer as shown in the circuit diagram from part 9 of this tutorial.

Copy the index.htm file to a micro SD card and insert it into the micro SD card socket of the Ethernet shield. The index.htm file can be copied below.

Load the above sketch to the Arduino and connect to the Arduino with Ethernet shield using a web browser.

You will not seen any significant difference between this part of the tutorial and part 9 of the tutorial, but we now have an easy way of extracting values sent from the Arduino to be used on a web page.

Web Page Source Code

The web page can be copied here and pasted to a file called index.htm:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page using Ajax with
XML</title>
        <script>
        function GetArduinoInputs()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseXML != null) {
                            // extract XML data from XML file
(containing switch states and analog value)

document.getElementById("input1").innerHTML =

this.responseXML.getElementsByTagName('button1')[0].childNode
s[0].nodeValue;

document.getElementById("input2").innerHTML =

this.responseXML.getElementsByTagName('button2')[0].childNode
s[0].nodeValue;

document.getElementById("input3").innerHTML =
```

```
this.responseXML.getElementsByTagName('analog1')[0].childNode
s[0].nodeValue;
                        }
                    }
                }
            }
            request.open("GET",   "ajax_inputs"   +   nocache,
true);
            request.send(null);
            setTimeout('GetArduinoInputs()', 1000);
        }
    </script>
    </head>
    <body onload="GetArduinoInputs()">
        <h1>Arduino Inputs from SD Card Web Page using Ajax
with XML</h1>
        <p>Button 1 (pin 7): <span id="input1">...</span></p>
        <p>Button 2 (pin 8): <span id="input2">...</span></p>
        <p>Analog (A2): <span id="input3">...</span></p>
    </body>
</html>
```

Arduino Web Server Gauge Displaying Analog Value

Part 15 of the Arduino Ethernet Shield Web Server Tutorial

A gauge component is used to display the analog value from one of the Arduino's analog pins on a web page. The dial gauge is updated using Ajax.

The gauge is written in JavaScript and uses the HTML5 canvas. The gauge is used as a component (unmodified) and is simply set up to display the analog value of one of the Arduino analog pins. The value is updated every 200ms.

Getting the Gauge Component

The gauge component is written by Mykhailo Stadnyk and can be downloaded from Mikhus at github. Also see a blog article on the gauge.

The JavaScript code from the file gauge.min.js (downloaded from the above github link) was used on the web page in this part of the tutorial.

Arduino Sketch and Web Page

Arduino Sketch

The Arduino sketch for this part of the tutorial is shown here.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>

// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   50

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20); // IP address, may need to change depending on
network
EthernetServer server(80);  // create a server at port 80
File webFile;               // the web page file on the SD card
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null
terminated string
char req_index = 0;              // index into HTTP_req buffer

void setup()
{
   // disable Ethernet chip
```

```
  pinMode(10, OUTPUT);
  digitalWrite(10, HIGH);

  Serial.begin(9600);       // for debugging

  // initialize SD card
  Serial.println("Initializing SD card...");
  if (!SD.begin(4)) {
    Serial.println("ERROR - SD card initialization failed!");
    return;    // init failed
  }
  Serial.println("SUCCESS - SD card initialized.");
  // check for index.htm file
  if (!SD.exists("index.htm")) {
    Serial.println("ERROR - Can't find index.htm file!");
    return;  // can't find index file
  }
  Serial.println("SUCCESS - Found index.htm file.");

  Ethernet.begin(mac, ip);  // initialize Ethernet device
  server.begin();           // start to listen for clients
}

void loop()
{
  EthernetClient client = server.available();  // try to get client

  if (client) {  // got client?
    boolean currentLineIsBlank = true;
    while (client.connected()) {
      if (client.available()) {   // client data available to read
        char c = client.read(); // read 1 byte (character) from client
        // buffer first part of HTTP request in HTTP_req array (string)
        // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
        if (req_index < (REQ_BUF_SZ - 1)) {
          HTTP_req[req_index] = c;          // save HTTP request character
          req_index++;
        }
        // last line of client request is blank and ends with \n
        // respond to client only after last line received
        if (c == '\n' && currentLineIsBlank) {
          // send a standard http response header
          client.println("HTTP/1.1 200 OK");
          // remainder of header follows below, depending on if
          // web page or XML page is requested
          // Ajax request - send XML file
          if (StrContains(HTTP_req, "ajax_inputs")) {
            // send rest of HTTP header
```

```
                client.println("Content-Type: text/xml");
                client.println("Connection: keep-alive");
                client.println();
                // send XML file containing input states
                XML_response(client);
            }
            else {  // web page request
                // send rest of HTTP header
                client.println("Content-Type: text/html");
                client.println("Connection: keep-alive");
                client.println();
                // send web page
                webFile = SD.open("index.htm");        // open web page file
                if (webFile) {
                    while(webFile.available()) {
                        client.write(webFile.read()); // send web page to client
                    }
                    webFile.close();
                }
            }
            // display received HTTP request on serial port
            Serial.print(HTTP_req);
            // reset buffer index and all buffer elements to 0
            req_index = 0;
            StrClear(HTTP_req, REQ_BUF_SZ);
            break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
            // last character on line of received text
            // starting new line with next character read
            currentLineIsBlank = true;
        }
        else if (c != '\r') {
            // a text character was received from client
            currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

// send the XML file containing analog value
void XML_response(EthernetClient cl)
{
  int analog_val;
```

```
    cl.print("<?xml version = \"1.0\" ?>");
    cl.print("<inputs>");
    // read analog pin A2
    analog_val = analogRead(2);
    cl.print("<analog>");
    cl.print(analog_val);
    cl.print("</analog>");
    cl.print("</inputs>");
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
    for (int i = 0; i < length; i++) {
        str[i] = 0;
    }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
{
    char found = 0;
    char index = 0;
    char len;

    len = strlen(str);

    if (strlen(sfind) > len) {
        return 0;
    }
    while (index < len) {
        if (str[index] == sfind[found]) {
            found++;
            if (strlen(sfind) == found) {
                return 1;
            }
        }
        else {
            found = 0;
        }
        index++;
    }

    return 0;
}
```

This sketch is a slightly modified version of the sketch from the previous part of this tutorial (part 14).

The sketch sends a single analog value from Arduino pin A2 to the web browser which is then fed to the dial on the web page. The analog value is updated on the web page using Ajax and the value is sent as part of an XML file from the Arduino.

The web page hosted by the Arduino web server is stored on a micro SD card on the Arduino Ethernet shield.

Web Page

The web page for this example is shown here.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino Ajax Dial</title>
        <script>
        var data_val = 0;
        eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+
        function GetArduinoInputs()
        {
            nocache = "&nocache=" + Math.random() * 1000000;
            var request = new XMLHttpRequest();
            request.onreadystatechange = function()
            {
                if (this.readyState == 4) {
                    if (this.status == 200) {
                        if (this.responseXML != null) {
                            document.getElementById("input3").innerHTML =
                                this.responseXML.getElementsByTagName('analog')[0].childNodes[0].nodeValue;
                                data_val = this.responseXML.getElementsByTagName('analog')[0].childNodes[0].nodeValue;
                        }
                    }
                }
            }
            request.open("GET", "ajax_inputs" + nocache, true);
            request.send(null);
            setTimeout('GetArduinoInputs()', 200);

        }
    </script>
    </head>
    <body onload="GetArduinoInputs()">
        <h1>Arduino Ajax Dial</h1>
        <p>Analog (A2): <span id="input3">...</span></p>
        <canvas id="an_gauge_1" data-major-ticks="0 100 200 300 400 500 600 700 800 900 1000 1023"
                data-type="canv-gauge" data-min-value="0" data-max-value="1023"
                data-onready="setInterval( function() { Gauge.Collection.get('an_gauge_1').setValue(data_val);}, 200);">
        </canvas>
    </body>
</html>
```

The Web Page for Displaying the Gauge with Gauge Code Cut Off – click for a bigger image

The web page is a modified version of the web page used in the previous part of this tutorial.

The Gauge

The gauge is made available to the web page by inserting the gauge code between the script tags in the head of the web page HTML code.

The gauge is displayed by using the HTML5 canvas tag in the body of the HTML.

Updating the Gauge

The analog value from the Arduino is received in the same way as it was in the previous part of this tutorial and displayed at the top of the web page in the same way as the previous part of this tutorial. The analog value is also saved to the JavaScript variable **data_val** so that it can be used by the gauge.

The value in **data_val** is fed to the gauge by using the line of JavaScript in the data-onready property of the canvas tag:

```
setInterval( function() {Gauge.Collection.get('an_gauge_1').setValue(data_val);}, 200);
```

The 200 in the above line tells the gauge to update every 200ms (200 milliseconds).

Timing

The gauge updates every 200ms, and the Ajax request for an analog value is also set to 200ms.

The timing for the Ajax refresh of the analog data is done in this line of JavaScript code from the web page:

```
setTimeout('GetArduinoInputs()', 200);
```

This refresh rate may cause a problem on a busy or slow network. If there are any problems, try changing this value to 1000 to make the analog value refresh every second.

Running the Gauge Sketch

Wire up the potentiometer as shown in the circuit diagram of part 9 of this tutorial – leave the push buttons shown in the diagram off, they will not be used.

Copy the web page (index.htm) to a micro SD card and insert it into the card slot of the Arduino Ethernet shield.

Load the above sketch to the Arduino web server. The web page is available for download below.

Web Page Source Code

Download the web page source code (index.htm) for this part of the tutorial here:

Arduino_web_gauge.zip (6.3 kB)

The license for the gauge is included in the download as a text file.

SD Card Web Server I/O

Part 16 of the Arduino Ethernet Shield Web Server Tutorial

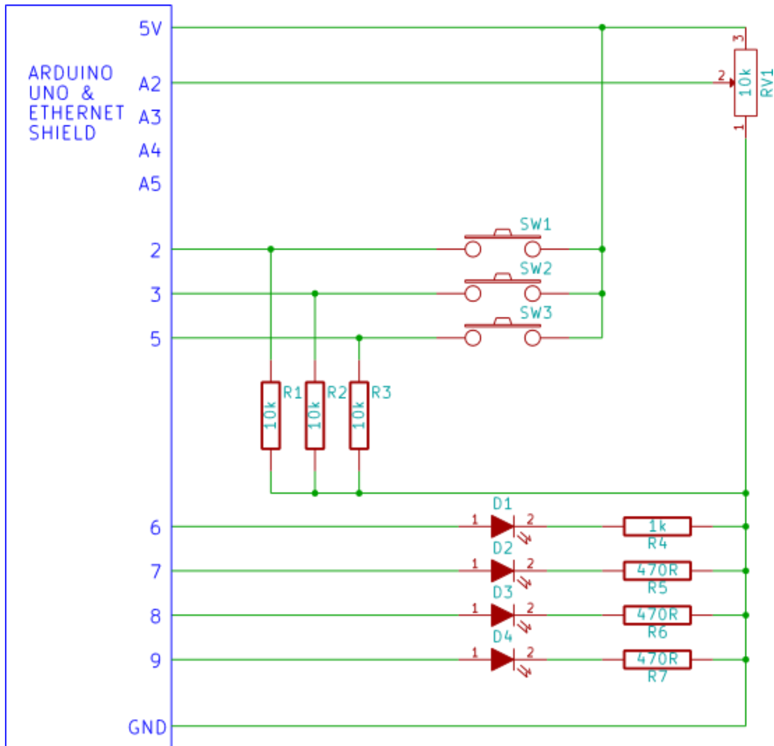
In this part of the tutorial, everything covered so far comes together. HTML, JavaScript, CSS, HTTP, Ajax and the SD card are used to make a web page that displays Arduino analog and digital inputs and allows digital outputs to be controlled.

The Arduino web server hosts a web page that displays four analog input values and the state of three switches. The web page allows four LEDs to be controlled – two LEDs are controlled using checkboxes and two LEDs are controlled using buttons.

When more than one computer (web browser) is connected to the Arduino web server, then outputs (LEDs) switched on using one computer will be updated on the other computer – i.e. when a checkbox is clicked to switch an LED on from one computer, the checkbox will also be checked on the other computer automatically.

Web Server I/O Circuit Diagram

The circuit diagram for this part of the tutorial is shown here:



In the video, A3 was connected to a voltage divider, A4 was connected through a resistor to GND and A5 was connected through a resistor to +5V – not shown in the circuit diagram.

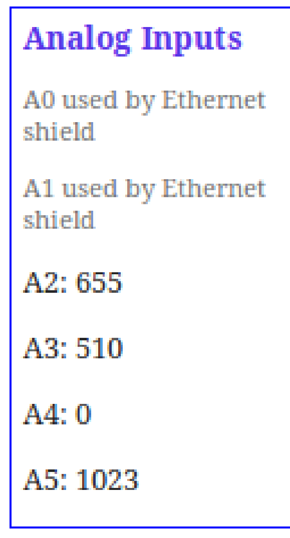
Overview of How the Web Server Works

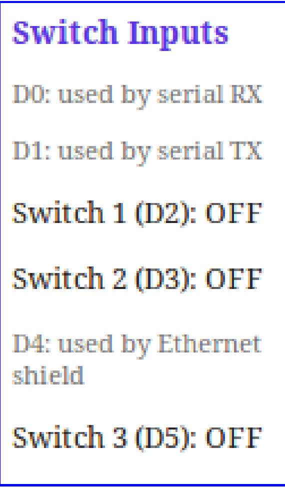
After a web browser has requested and loaded the web page from the Arduino web

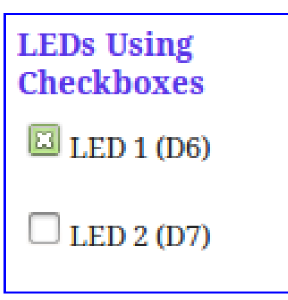
server, the JavaScript in the web page will send an Ajax request for data from the Arduino every second.

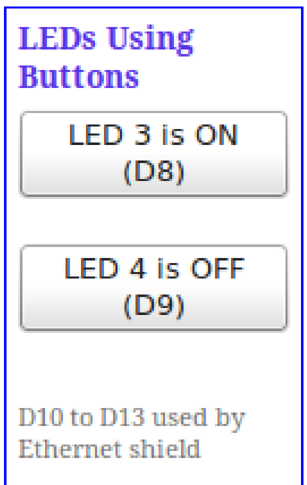
The web page that the Arduino web server hosts is shown here:

Arduino Ajax I/O









Web Page Hosted by the Arduino Web Server

The Arduino will respond to every Ajax request by sending an XML file back to the web browser. The XML file contains the values from the four analog inputs of the Arduino (A2 to A5), the state of three pins (switches on pins 2, 3 and 5) and the state of the four LEDs.

```xml
<?xml version = "1.0" ?>
<inputs>
    <analog>655</analog>
    <analog>510</analog>
    <analog>0</analog>
    <analog>1023</analog>
    <switch>OFF</switch>
    <switch>OFF</switch>
    <switch>OFF</switch>
    <LED>checked</LED>
    <LED>unchecked</LED>
    <LED>on</LED>
    <LED>off</LED>
</inputs>
```

XML File Sent by Arduino

When an LED is switched on from the web page by checking a checkbox, the JavaScript will send the state of the checkbox (send an instruction to switch the LED on) with the next Ajax request. The same will occur if the LED is switched off or if one of the buttons is used to switch an LED on or off.

Source Code

The source code for both the Arduino sketch and web page are a bit big to include on this page. It is suggested to download these files below and open them in your favourite editor when following the explanation below.

Arduino sketch and web page: web_server_IO.zip (4.7 kB)

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>
// size of buffer used to capture HTTP requests
#define REQ_BUF_SZ   60

// MAC address from Ethernet shield sticker under board
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 0, 20); // IP address, may need to change depending on network
EthernetServer server(80);  // create a server at port 80
File webFile;             // the web page file on the SD card
char HTTP_req[REQ_BUF_SZ] = {0}; // buffered HTTP request stored as null terminated string
char req_index = 0;            // index into HTTP_req buffer
boolean LED_state[4] = {0}; // stores the states of the LEDs

void setup()
{
   // disable Ethernet chip
   pinMode(10, OUTPUT);
   digitalWrite(10, HIGH);

   Serial.begin(9600);       // for debugging

   // initialize SD card
   Serial.println("Initializing SD card...");
   if (!SD.begin(4)) {
      Serial.println("ERROR - SD card initialization failed!");
      return;    // init failed
   }
   Serial.println("SUCCESS - SD card initialized.");
   // check for index.htm file
   if (!SD.exists("index.htm")) {
      Serial.println("ERROR - Can't find index.htm file!");
      return;  // can't find index file
   }
   Serial.println("SUCCESS - Found index.htm file.");
   // switches on pins 2, 3 and 5
```

```
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(5, INPUT);
    // LEDs
    pinMode(6, OUTPUT);
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    Ethernet.begin(mac, ip);  // initialize Ethernet device
    server.begin();           // start to listen for clients
}

void loop()
{
    EthernetClient client = server.available();  // try to get client

    if (client) {  // got client?
        boolean currentLineIsBlank = true;
        while (client.connected()) {
            if (client.available()) {   // client data available to read
                char c = client.read(); // read 1 byte (character) from client
                // limit the size of the stored received HTTP request
                // buffer first part of HTTP request in HTTP_req array (string)
                // leave last element in array as 0 to null terminate string (REQ_BUF_SZ - 1)
                if (req_index < (REQ_BUF_SZ - 1)) {
                    HTTP_req[req_index] = c;          // save HTTP request character
                    req_index++;
                }
                // last line of client request is blank and ends with \n
                // respond to client only after last line received
                if (c == '\n' && currentLineIsBlank) {
                    // send a standard http response header
                    client.println("HTTP/1.1 200 OK");
                    // remainder of header follows below, depending on if
                    // web page or XML page is requested
                    // Ajax request - send XML file
                    if (StrContains(HTTP_req, "ajax_inputs")) {
                        // send rest of HTTP header
                        client.println("Content-Type: text/xml");
                        client.println("Connection: keep-alive");
                        client.println();
                        SetLEDs();
                        // send XML file containing input states
                        XML_response(client);
                    }
                    else {  // web page request
                        // send rest of HTTP header
```

```
                client.println("Content-Type: text/html");
                client.println("Connection: keep-alive");
                client.println();
                // send web page
                webFile = SD.open("index.htm");        // open web page file
                if (webFile) {
                    while(webFile.available()) {
                        client.write(webFile.read()); // send web page to client
                    }
                    webFile.close();
                }
            }
            // display received HTTP request on serial port
            Serial.print(HTTP_req);
            // reset buffer index and all buffer elements to 0
            req_index = 0;
            StrClear(HTTP_req, REQ_BUF_SZ);
            break;
        }
        // every line of text received from the client ends with \r\n
        if (c == '\n') {
            // last character on line of received text
            // starting new line with next character read
            currentLineIsBlank = true;
        }
        else if (c != '\r') {
            // a text character was received from client
            currentLineIsBlank = false;
        }
      } // end if (client.available())
    } // end while (client.connected())
    delay(1);      // give the web browser time to receive the data
    client.stop(); // close the connection
  } // end if (client)
}

// checks if received HTTP request is switching on/off LEDs
// also saves the state of the LEDs
void SetLEDs(void)
{
  // LED 1 (pin 6)
  if (StrContains(HTTP_req, "LED1=1")) {
    LED_state[0] = 1;  // save LED state
    digitalWrite(6, HIGH);
  }
  else if (StrContains(HTTP_req, "LED1=0")) {
    LED_state[0] = 0;  // save LED state
    digitalWrite(6, LOW);
```

```
  }
  // LED 2 (pin 7)
  if (StrContains(HTTP_req, "LED2=1")) {
    LED_state[1] = 1;  // save LED state
    digitalWrite(7, HIGH);
  }
  else if (StrContains(HTTP_req, "LED2=0")) {
    LED_state[1] = 0;  // save LED state
    digitalWrite(7, LOW);
  }
  // LED 3 (pin 8)
  if (StrContains(HTTP_req, "LED3=1")) {
    LED_state[2] = 1;  // save LED state
    digitalWrite(8, HIGH);
  }
  else if (StrContains(HTTP_req, "LED3=0")) {
    LED_state[2] = 0;  // save LED state
    digitalWrite(8, LOW);
  }
  // LED 4 (pin 9)
  if (StrContains(HTTP_req, "LED4=1")) {
    LED_state[3] = 1;  // save LED state
    digitalWrite(9, HIGH);
  }
  else if (StrContains(HTTP_req, "LED4=0")) {
    LED_state[3] = 0;  // save LED state
    digitalWrite(9, LOW);
  }
}

// send the XML file with analog values, switch status
//  and LED status
void XML_response(EthernetClient cl)
{
  int analog_val;            // stores value read from analog inputs
  int count;                 // used by 'for' loops
  int sw_arr[] = {2, 3, 5};  // pins interfaced to switches

  cl.print("<?xml version = \"1.0\" ?>");
  cl.print("<inputs>");
  // read analog inputs
  for (count = 2; count <= 5; count++) { // A2 to A5
    analog_val = analogRead(count);
    cl.print("<analog>");
    cl.print(analog_val);
    cl.println("</analog>");
  }
  // read switches
```

```
for (count = 0; count < 3; count++) {
    cl.print("<switch>");
    if (digitalRead(sw_arr[count])) {
        cl.print("ON");
    }
    else {
        cl.print("OFF");
    }
    cl.println("</switch>");
}
// checkbox LED states
// LED1
cl.print("<LED>");
if (LED_state[0]) {
    cl.print("checked");
}
else {
    cl.print("unchecked");
}
cl.println("</LED>");
// LED2
cl.print("<LED>");
if (LED_state[1]) {
    cl.print("checked");
}
else {
    cl.print("unchecked");
}
 cl.println("</LED>");
// button LED states
// LED3
cl.print("<LED>");
if (LED_state[2]) {
    cl.print("on");
}
else {
    cl.print("off");
}
cl.println("</LED>");
// LED4
cl.print("<LED>");
if (LED_state[3]) {
    cl.print("on");
}
else {
    cl.print("off");
}
cl.println("</LED>");
```

```
  cl.print("</inputs>");
}

// sets every element of str to 0 (clears array)
void StrClear(char *str, char length)
{
  for (int i = 0; i < length; i++) {
    str[i] = 0;
  }
}

// searches for the string sfind in the string str
// returns 1 if string found
// returns 0 if string not found
char StrContains(char *str, char *sfind)
{
  char found = 0;
  char index = 0;
  char len;

  len = strlen(str);

  if (strlen(sfind) > len) {
    return 0;
  }
  while (index < len) {
    if (str[index] == sfind[found]) {
      found++;
      if (strlen(sfind) == found) {
        return 1;
      }
    }
    else {
      found = 0;
    }
    index++;
  }

  return 0;
}

//////////////////////////////////
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino Ajax I/O</title>
        <script>
```

```
          strLED1 = "";
          strLED2 = "";
          strLED3 = "";
          strLED4 = "";
          var LED3_state = 0;
          var LED4_state = 0;
          function GetArduinoIO()
          {
               nocache = "&nocache=" + Math.random() * 1000000;
               var request = new XMLHttpRequest();
               request.onreadystatechange = function()
               {
                    if (this.readyState == 4) {
                         if (this.status == 200) {
                              if (this.responseXML != null) {
                                   //   XML   file   received   -
contains analog values, switch values and LED states
                                   var count;
                                   // get analog inputs
                                   var          num_an          =
this.responseXML.getElementsByTagName('analog').length;
                                   for   (count   =   0;   count   <
num_an; count++) {

     document.getElementsByClassName("analog")[count].innerHTM
L =

     this.responseXML.getElementsByTagName('analog')[count].ch
ildNodes[0].nodeValue;
                                   }
                                   // get switch inputs
                                   var          num_an          =
this.responseXML.getElementsByTagName('switch').length;
                                   for   (count   =   0;   count   <
num_an; count++) {

     document.getElementsByClassName("switches")[count].innerH
TML =

     this.responseXML.getElementsByTagName('switch')[count].ch
ildNodes[0].nodeValue;
                                   }
                                   // LED 1
                                   if
(this.responseXML.getElementsByTagName('LED')[0].childNodes[0
].nodeValue === "checked") {

     document.LED_form.LED1.checked = true;
                                   }
                                   else {
```

```
     document.LED_form.LED1.checked = false;
                                    }
                                    // LED 2
                                    if
(this.responseXML.getElementsByTagName('LED')[1].childNodes[0
].nodeValue === "checked") {

     document.LED_form.LED2.checked = true;
                                    }
                                    else {

     document.LED_form.LED2.checked = false;
                                    }
                                    // LED 3
                                    if
(this.responseXML.getElementsByTagName('LED')[2].childNodes[0
].nodeValue === "on") {

     document.getElementById("LED3").innerHTML = "LED 3 is ON
(D8)";
                                         LED3_state = 1;
                                    }
                                    else {

     document.getElementById("LED3").innerHTML = "LED 3 is OFF
(D8)";
                                         LED3_state = 0;
                                    }
                                    // LED 4
                                    if
(this.responseXML.getElementsByTagName('LED')[3].childNodes[0
].nodeValue === "on") {

     document.getElementById("LED4").innerHTML = "LED 4 is ON
(D9)";
                                         LED4_state = 1;
                                    }
                                    else {

     document.getElementById("LED4").innerHTML = "LED 4 is OFF
(D9)";
                                         LED4_state = 0;
                                    }
                               }
                          }
                     }
                }
                // send HTTP GET request with LEDs to switch
on/off if any
```

```
                request.open("GET", "ajax_inputs" + strLED1 +
strLED2 + strLED3 + strLED4 + nocache, true);
                request.send(null);
                setTimeout('GetArduinoIO()', 1000);
                strLED1 = "";
                strLED2 = "";
                strLED3 = "";
                strLED4 = "";
            }
            // service LEDs when checkbox checked/unchecked
            function GetCheck()
            {
                if (LED_form.LED1.checked) {
                    strLED1 = "&LED1=1";
                }
                else {
                    strLED1 = "&LED1=0";
                }
                if (LED_form.LED2.checked) {
                    strLED2 = "&LED2=1";
                }
                else {
                    strLED2 = "&LED2=0";
                }
            }
            function GetButton1()
            {
                if (LED3_state === 1) {
                    LED3_state = 0;
                    strLED3 = "&LED3=0";
                }
                else {
                    LED3_state = 1;
                    strLED3 = "&LED3=1";
                }
            }
            function GetButton2()
            {
                if (LED4_state === 1) {
                    LED4_state = 0;
                    strLED4 = "&LED4=0";
                }
                else {
                    LED4_state = 1;
                    strLED4 = "&LED4=1";
                }
            }
        </script>
        <style>
            .IO_box {
```

```
            float: left;
            margin: 0 20px 20px 0;
            border: 1px solid blue;
            padding: 0 5px 0 5px;
            width: 120px;
        }
        h1 {
            font-size: 120%;
            color: blue;
            margin: 0 0 10px 0;
        }
        h2 {
            font-size: 85%;
            color: #5734E6;
            margin: 5px 0 5px 0;
        }
        p, form, button {
            font-size: 80%;
            color: #252525;
        }
        .small_text {
            font-size: 70%;
            color: #737373;
        }
    </style>
    </head>
    <body onload="GetArduinoIO()">
        <h1>Arduino Ajax I/O</h1>
        <div class="IO_box">
            <h2>Analog Inputs</h2>
            <p   class="small_text">A0   used   by   Ethernet
shield</p>
            <p   class="small_text">A1   used   by   Ethernet
shield</p>
            <p>A2: <span class="analog">...</span></p>
            <p>A3: <span class="analog">...</span></p>
            <p>A4: <span class="analog">...</span></p>
            <p>A5: <span class="analog">...</span></p>
        </div>
        <div class="IO_box">
            <h2>Switch Inputs</h2>
            <p class="small_text">D0: used by serial RX</p>
            <p class="small_text">D1: used by serial TX</p>
            <p>Switch         1         (D2):           <span
class="switches">...</span></p>
            <p>Switch         2         (D3):           <span
class="switches">...</span></p>
            <p   class="small_text">D4:   used   by   Ethernet
shield</p>
            <p>Switch         3         (D5):           <span
```

```
class="switches">...</span></p>
         </div>
         <div class="IO_box">
              <h2>LEDs Using Checkboxes</h2>
              <form id="check_LEDs" name="LED_form">
                   <input      type="checkbox"      name="LED1"
value="0" onclick="GetCheck()" />LED 1 (D6)<br /><br />
                   <input      type="checkbox"      name="LED2"
value="0" onclick="GetCheck()" />LED 2 (D7)<br /><br />
              </form>
         </div>
         <div class="IO_box">
              <h2>LEDs Using Buttons</h2>
              <button         type="button"         id="LED3"
onclick="GetButton1()">LED 3 is OFF (D8)</button><br /><br />
              <button         type="button"         id="LED4"
onclick="GetButton2()">LED 4 is OFF (D9)</button><br /><br />
              <p   class="small_text">D10   to   D13   used   by
Ethernet shield</p>
         </div>
    </body>
</html>
////////////////////////////////////////
```

Running the Example

To run this example, first copy the web page (index.htm) from the above download to a micro SD card. Insert the micro SD card in the card slot of the Ethernet shield. The Ethernet shield must be plugged into an Arduino board, connected to your network and powered by USB or an external power supply.

Build as much of the circuit as you want. If you don't connect anything to the input pins, they may just toggle randomly as they pick up noise. Inputs can always be pulled high or low through a resistor if you do not want to connect them to switches or a potentiometer.

Load the Arduino sketch (eth_websrv_SD_Ajax_in_out) from the above download to the Arduino.

Surf to the IP address set in the sketch using a web browser – you should see the web page loaded and the analog values and switch values updated every second.

How the Example Works

The Arduino sketch and web page are basically modified versions of code from previous examples in this multi-part tutorial. The files are modified versions from the examples that use Ajax with XML (part 14 and part 15).

Analog and Digital Inputs

The analog inputs (analog values) and digital inputs (switch states) are requested and received the same ways as explained in part 14 of this tutorial, except that the

analog values are all enclosed in <analog> tags and the switch states are all enclosed in <switch> tags – i.e. they do not use unique tag names for each input.

When the JavaScript in the web page receives the XML file from the Arduino, it uses one **for** loop to extract the analog values and a second **for** loop to extract the switch states.

The JavaScript code for extracting the analog values from the XML file is shown here:

```
var num_an = this.responseXML.getElementsByTagName('analog').length;
for (count = 0; count < num_an; count++) {

document.getElementsByClassName("analog")[count].innerHTML =

this.responseXML.getElementsByTagName('analog')[count].childNodes[0].nodeValue;
}
```

The first line of the above code gets the number of items in the XML file that are enclosed in <analog> tags and stores it in the variable **num_an** which is then be used in the **for** loop to get the correct number of analog values.

In the **for** loop, the following code gets hold of each HTML **span** that has the class **analog**:

```
document.getElementsByClassName("analog")[count].innerHTML
```

Which gets hold of each **span** element in the following HTML:

```
<p>A2: <span class="analog">...</span></p>
<p>A3: <span class="analog">...</span></p>
<p>A4: <span class="analog">...</span></p>
<p>A5: <span class="analog">...</span></p>
```

This line of code then gets each analog value from the XML file in turn:

```
this.responseXML.getElementsByTagName('analog')[count].childNodes[0].nodeValue;
```

So the above two lines of JavaScript code working together get each analog value from the XML file in the order that they appear in the file and insert these values into each **span** in the order that they appear in the HTML.

The switch values work the same way, but each HTML **span** with the class name **switches** has a value from the XML file with the tag **switch** inserted into it in the order that they appear.

LEDs

The Arduino keeps track of which LEDs are on and which LEDs are off. This information is sent back as part of the XML file in response to the Ajax request that occurs every second.

In the Arduino sketch, the following array stores the LED states (stores 1 for 'on' LED and 0 for 'off' LED):

```
boolean LED_state[4] = {0}; // stores the states of the LEDs
```

This array is initialized with zeros – all LEDs off. Each element of the array corresponds to an LED in order from LED1 to LED4, i.e. LED_state[0] stores the state of LED 1, LED_state[1] stores the state of LED 2, etc.

LEDs Using Checkboxes

In the JavaScript code, the following lines of code show how the state of LED 1 is extracted from the received XML file and used to update the checkbox – i.e. to mark it as checked or unchecked, depending on the state of the LED.

```
// LED 1
if
(this.responseXML.getElementsByTagName('LED')[0].childNodes[0
]. nodeValue === "checked") {
    document.LED_form.LED1.checked = true;
}
else {
    document.LED_form.LED1.checked = false;
}
```

In the above code, if the first value (i.e. for LED 1) from the XML file in an <LED> tag contains the text **checked** then the corresponding checkbox is checked using this line of JavaScript:

```
document.LED_form.LED1.checked = true;
```

Otherwise the checkbox is unchecked.

This ensures that every browser that connects to the Arduino web server displays the correct states of the LEDs because they are receiving the LED states every second.

When a user checks or unchecks a checkbox, then the next Ajax request will include the LED name and state with the next HTTP GET request.

The JavaScript function **GetCheck()** is run whenever a checkbox is clicked. If the checkbox for LED 1 is checked, then the value of a string for LED 1 is set:

```
strLED1 = "&LED1=1";
```

If the checkbox is unchecked, the the string is set as follows:

```
strLED1 = "&LED1=0";
```

The next time that **GetArduinoIO()** function is called, which occurs every second, the **strLED1** string will now not be empty. This string will now be included with the GET request:

```
request.open("GET", "ajax_inputs" + strLED1 + strLED2 +
strLED3 + strLED4 + nocache, true);
```

All the other LED strings will be empty if a checkbox or button has not been clicked, so will not be included with the GET request.

After the get request has occured, all LED strings are set back to empty:

```
strLED1 = "";
strLED2 = "";
strLED3 = "";
strLED4 = "";
```

When the Arduino receives the Ajax GET request, the **SetLEDs()** function in the Arduino sketch checks for a change of LED state in the received GET request and if it finds one will switch the corresponding LED on or off and then save the state of the LED to the **LED_state[]** array.

LEDs Using Buttons

The buttons that control the LEDs work in the same way as the checkboxes, except that each button has its own JavaScript function that runs when a button is clicked.

The text on the buttons is updated to show the state of the LEDs that they control.

Buttons do not have a 'checked' property, so the current state of each button's LED is stored in a variable:

```
var LED3_state = 0;
var LED4_state = 0;
```

Storing the button state is necessary so that when the button is clicked again, it knows whether to update the string that is sent with the GET request to switch the LED on or off.

When the Arduino sends the state of the LEDs in the XML file that are controlled by the buttons, it inserts "on" or "off" between the <LED> tags instead of "checked" or "unchecked" as it does for the checkboxes.

CSS

The CSS used in this example puts each set of inputs or outputs in their own boxes with a blue border. It also positions the boxes next to each other. If the browser windows is resized along its width, then the boxes on the right will move below the boxes on the left.

The CSS in this part of the tutorial will not be explained here, but will be explained in the next parts of this tutorial where we look at more CSS.

Accessing HTML Tags with CSS and JavaScript

Part 17 of the Arduino Ethernet Shield Web Server Tutorial

How to access HTML elements (tags) with CSS and JavaScript in order to apply CSS styles to the elements and manipulate the elements using JavaScript. Shows how to reference HTML tags by using ID and class names.

Running the Examples

Each of the examples below can be copied from this page and saved as an HTML file (e.g. index.htm). The file can then be loaded to the Arduino if desired by following the example in part 4 of this tutorial.

Alternatively each example can be opened from the computer in a web browser without using the Arduino.

Accessing HTML Elements

HTML elements need to be accessed so that CSS styles can be applied to them and so that we can get hold of these elements with JavaScript in order to change or manipulate them.

Accessing HTML Elements in CSS

There are three main methods of accessing or referring to HTML elements in CSS:

By referring to the HTML element by its HTML tag name, e.g. **p** to refer to the paragraph HTML tag – <p>

By using an ID, e.g. <p id="red_text">Some text</p>
By using a class, e.g. <p class="red_text">Some text</p>
A combination of the above methods can also be used to access an HTML element.

We will now look at an example of each of the above methods.

Reference by HTML Element Name

This method was already demonstrated in part 12 of this tutorial and was used in part 16 (the previous part).

```
<!DOCTYPE html>
<html>
    <head>
```

```
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            p {
                font-family: arial, verdana, sans-serif;
                font-size: 12pt;
                color: #6B6BD7;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with CSS</h1>
        <p>Welcome to the Arduino web page with CSS styling.</p>
    </body>
</html>
```

In the above HTML with CSS, the CSS part is applied to every paragraph in the web page. This is done by referring to the paragraph by its HTML element name, **p**:

```
p {
    font-family: arial, verdana, sans-serif;
    font-size: 12pt;
    color: #6B6BD7;
}
```

Styles can be applied to other HTML elements in the same way by referring to their HTML tag names, e.g. h1, h2, h3, div, span, b, etc.

This method is used to set the default style for HTML elements on a page. It is possible to override these default styles by giving an HTML element an ID or class name.

Reference by ID

An ID of a specific name can only be used once on a web page. The name of the ID is chosen by the person writing the HTML and CSS. An ID would normally be used for something like a menu that occurs only once per page. Using an ID also allows JavaScript to access the element using the unique ID.

This example code shows the use of an ID:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            h1 {
                font-family: arial, verdana, sans-serif;
                font-size: 16pt;
                color: blue;
            }
```

```
            p {
                font-family: arial, verdana, sans-serif;
                font-size: 12pt;
                color: #6B6BD7;
            }
            #green_small {
                font-size: 9pt;
                color: green;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with CSS</h1>
        <p>Welcome to the Arduino web page with CSS
styling.</p>
        <p id="green_small">This is a second paragraph.</p>
        <p>This is a third paragraph.</p>
    </body>
</html>
```

The above markup code produces the following text on the web page when loaded in a browser:

Arduino SD Card Page with CSS

Welcome to the Arduino web page with CSS styling.

This is a second paragraph.

This is a third paragraph.

As can be seen in the above markup, when an HTML element is referred to by its ID, the # character is used before its name in the CSS style:

```
#green_small {
    font-size: 9pt;
    color: green;
}
```

This style is then applied to the HTML element with that ID:

```
<p id="green_small">This is a second paragraph.</p>
```

Because it is an ID, it may not be used again on the web page – other IDs may be used, but each must have a unique name.

Also notice that the ID overrides the **p** default style that applies to all paragraphs. The other paragraphs that do not have an ID are then formatted with the default paragraph style.

Reference by Class

A class works the same way as an ID, except that it may be used more than once

on a web page. A class uses a dot (.) in front of the class name in the CSS style to show that it is referring to a class and not an ID or HTML element.

The following example shows how to apply a CSS style to HTML elements that have class names.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            h1 {
                font-family: arial, verdana, sans-serif;
                font-size: 16pt;
                color: blue;
            }
            p {
                font-family: arial, verdana, sans-serif;
                font-size: 12pt;
                color: #6B6BD7;
            }
            .red_big {
                font-size: 14pt;
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with <span
class="red_big">CSS</span></h1>
        <p>Welcome to the <span
class="red_big">Arduino</span> web page
with <span class="red_big">CSS styling</span>.</p>
        <p>This is a second paragraph.</p>
        <p class="red_big">This is a third paragraph.</p>
    </body>
</html>
```

The above markup produces the following web page:

Arduino SD Card Page with CSS

Welcome to the Arduino web page with CSS styling.

This is a second paragraph.

This is a third paragraph.

Notice that the class style has been applied both to the HTML **span** tag as well as the HTML **p** (paragraph) tag. The style applied to the HTML elements that have class names overrides the default style.

The CSS style applied to the HTML elements with the class name **red_big** can be seen here using the dot to show that it is referring to a class:

```
.red_big {
    font-size: 14pt;
    color: red;
}
```

Mixing Access Methods

The same CSS style can be applied to more than one HTML element. HTML elements can also be referred to more specifically, e.g. every paragraph inside a div element. This example demonstrates:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <style type="text/css">
            h1, p {
                font-family: arial, verdana, sans-serif;
                color: cyan;
            }
            h1 {
                font-size: 16pt;
            }
            p {
                font-size: 12pt;
            }
            div p {
                font-style: italic;
                letter-spacing: 5px;
            }
        </style>
    </head>
    <body>
        <h1>Arduino SD Card Page with CSS</h1>
        <p>Welcome to the Arduino web page with CSS
styling.</p>
        <div><p>This is a second paragraph.</p></div>
        <p>This is a third paragraph.</p>
        <div><p>This is a fourth paragraph.</p></div>
    </body>
</html>
```

When a CSS style has HTML tag names, id names or class names separated by a comma, then the specified style applies to each of the elements.

The following CSS style applies to both the h1 tag and the p tag and specifies the font family for both as well as the colour as cyan:

```
h1, p {
    font-family: arial, verdana, sans-serif;
    color: cyan;
}
```

In the above HTML with CSS listing, the sizes of the h1 tag and paragraph are then specified separately.

When the CSS style has HTML tag names, id names or class names that are not separated by a comma, then the style is applied to the elements as they occur inside each other. In the above HTML with CSS listing, a CSS style is applied to every paragarph that occurrs in an HTML div element. The style is shown here:

```
div p {
    font-style: italic;
    letter-spacing: 5px;
}
```

The above style makes the font italic and separates paragraph letters by 5 pixels in every paragraph that occurs inside an HTML div.

Accessing HTML Elements in JavaScript

HTML ID and class names allow JavaScript to get hold of these HTML tags to modify or manipulate them.

If you have been following each part of this tutorial, you will recognize these methods of accessing HTML elements using JavaScript.

Accessing HTML Elements with ID Names in JavaScript

The following example shows how to access an HTML element that has an ID name from JavaScript:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <script>
            var btn_count = 0;
            function ButtonCount()
            {
                btn_count++;

document.getElementById("btn_clicks").innerHTML = btn_count;
            }
        </script>
    </head>
    <body>
        <h1>Arduino SD Card Page with JavaScript</h1>
```

```
        <button  type="button"  onclick="ButtonCount()">Click
to Count</button>
        <p>The    button    has    been    clicked    <span
id="btn_clicks">0</span> times.</p>
    </body>
</html>
```

The JavaScript function ButtonCount() is run every time that the button is clicked. The function adds 1 to the variable **btn_count** every time that the function is run.

The JavaScript gets hold of the HTML span tag with the ID **btn_clicks** by using the following code:

```
document.getElementById("btn_clicks").innerHTML = btn_count;
```

Which accesses the span in this line of HTML:

```
<p>The button has been clicked <span id="btn_clicks">0</span>
times.</p>
```

The current value of **btn_count** is then inserted into the HTML span.

Remember that only one element on the page can use the ID with the name **btn_clicks**.

Accessing HTML Elements with Class Names in JavaScript

The following code accesses HTML tags that have the same class name:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino SD Card Web Page</title>
        <script>
            var btn_count = 0;
            var btn_count_0 = 0;
            var num_classes = 0;
            function ButtonCount()
            {
                btn_count++;

document.getElementsByClassName("btn_clicks")[0].innerHTML   =
btn_count;

document.getElementsByClassName("btn_clicks")[1].innerHTML   =
btn_count_0;
                btn_count_0++;
                //  get  the  number  of  btn_clicks  classes  on
the page

document.getElementById("btn_classes").innerHTML =
```

```
document.getElementsByClassName("btn_clicks").length;
            }
        </script>
    </head>
    <body>
        <h1>Arduino SD Card Page with JavaScript</h1>
        <button  type="button"  onclick="ButtonCount()">Click
to Count</button>
        <p>The  button  has  been  clicked  <span
class="btn_clicks">0</span> times.</p>
        <p>Or  counting  from  0  clicked  <span
class="btn_clicks">?</span> times.</p>
        <p>Number  of  "btn_clicks"  classes  on  page:  <span
id="btn_classes"></span></p>
    </body>
</html>
```

Two HTML span tags have the same class name (btn_clicks) and are accessed in the JavaScript code as shown here:

```
document.getElementsByClassName("btn_clicks")[0].innerHTML   =
btn_count;
document.getElementsByClassName("btn_clicks")[1].innerHTML   =
btn_count_0;
```

The number of HTML tags on the page that use this class name can be obtained using this code:

```
document.getElementsByClassName("btn_clicks").length;
```

Each tag with this class name can then be accessed in a loop if needed.

The HTML tags with the same class name are accessed on the page in the order that they occur from top to bottom.

CSS for Positioning, Sizing and Spacing

Part 18 of the Arduino Ethernet Shield Web Server Tutorial

Part 16 of this tutorial uses CSS to position HTML div elements that contain Arduino inputs and outputs. The CSS also sizes and spaces the div elements. This part of the tutorial explains the CSS used for positioning, sizing and spacing as used in part 16.

Making a HTML div Element Visible

The following basic web page consists of a paragraph of text inside a div element. The div is given a class name of **txt_block** and a CSS style is applied to the class.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
        .txt_block {
            border: 1px solid red;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 1</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
    </body>
</html>
```

To make the div visible, a 1 pixel wide, solid red border is drawn around the div using the following CSS style:

```
border: 1px solid red;
```

This produces the following output in a web browser:

Another way to make the div visible is to change its background colour, e.g. to change the div to green:

```
background-color: green;
```

As can be seen in the above image, the div extends to the edge of the web browser. If the web browser is resized, the div will always extend the width of the browser with a small margin of empty space on either side.

Sizing the div

The div can be sized by applying CSS **width** and **height** styles to it. The HTML/CSS listing below adds sizing to the div.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
        .txt_block {
            border: 1px solid red;
            width: 140px;
            height: 120px;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 2</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
    </body>
</html>
```

The above markup produces the following in a web browser:

The size of the div is now 140 pixels wide and 120 pixels high. In CSS px means pixels.

Padding the div

Padding is now applied to the left of the div so that there is a space between the left of the div and the text inside it.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
        .txt_block {
            border: 1px solid red;
            width: 140px;
            height: 120px;
            padding-left: 10px;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 3</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
    </body>
</html>
```

In the above markup, CSS padding of 10 pixels is applied as follows:

```
padding-left: 10px;
```

The result of adding the padding can be seen in this image:

Arduino CSS Example 3

A paragraph of text for this example.

Padding can be added to all sides of the div. Padding makes a space between the edge of the div and the inside of the div.

Padding for each side can be specified in CSS as follows:

```
padding-top: 5px;
padding-right: 3px;
padding-bottom: 7px;
padding-left: 10px;
```

There is a shorter method of specifying padding shown here.

```
padding: 5px 3px 7px 10px;
```

When specifying the padding using the above method, the order of the padding sizes from left to right always apply to the div or other element being padded by starting at the top and moving clockwise.

In other words, the above line of code applies padding to the div in this order:

```
padding: top right bottom left;
```

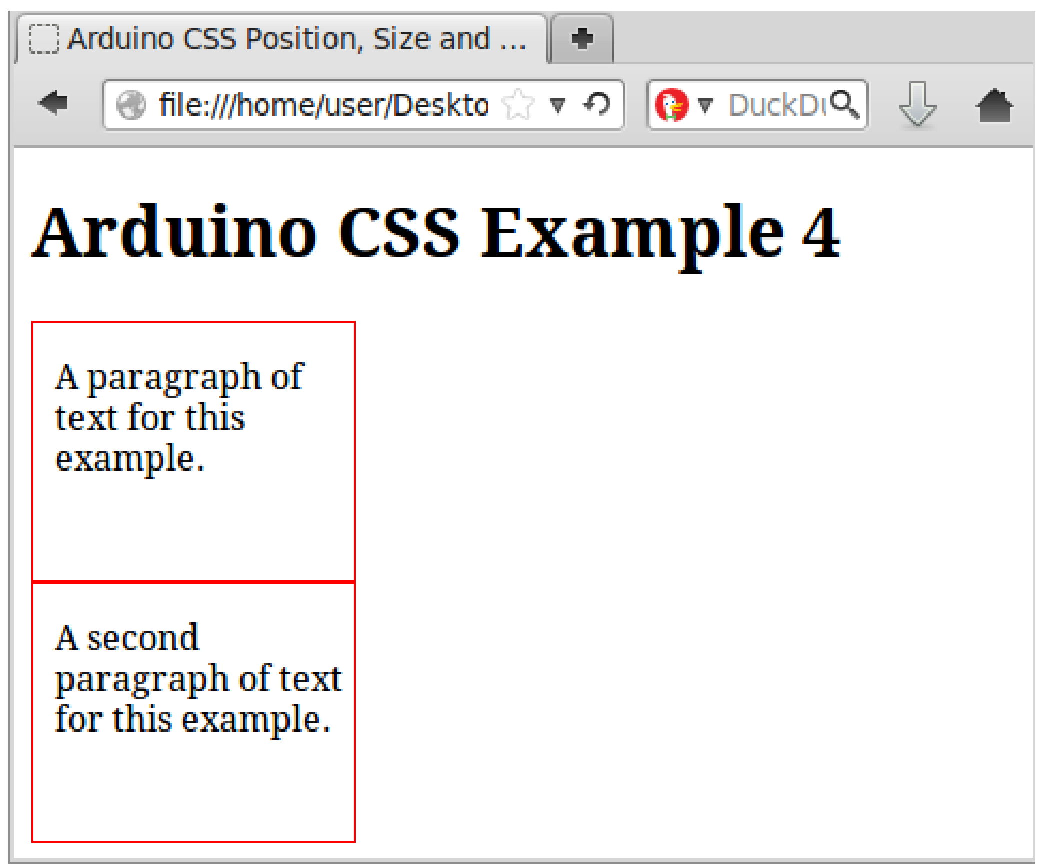
Adding a Second div

We now add a second div that is formatted the same as the first div by using the same class name for the second div.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
```

```
        .txt_block {
            border: 1px solid red;
            width: 140px;
            height: 120px;
            padding-left: 10px;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 4</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
        <div class="txt_block">
            <p>A    second    paragraph    of    text    for    this
example.</p>
        </div>
    </body>
</html>
```

The above markup produces the following web page:



Positioning the divs

There are a number of different methods for positioning HTML elements on a web page using CSS. We will look at one method here.

To position the two divs next to each other, we will use the CSS **float** style as

shown in this next markup lising.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
        .txt_block {
            border: 1px solid red;
            width: 140px;
            height: 120px;
            padding-left: 10px;
            float: left;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 5</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
        <div class="txt_block">
            <p>A   second   paragraph   of   text   for   this
example.</p>
        </div>
    </body>
</html>
```

By adding the CSS float style, the two divs are now floating to the left of the web page. The causes the first div to be placed on the left of the page and the second div placed on the left of the page next to the first div.

```
float: left;
```

The above markup produces the following web page:

Spacing the divs

To add some space between the divs, we finally add a 20 pixel margin to the right of the divs as shown in the following listing.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Arduino CSS Position, Size and Space</title>
    </head>
    <style>
        .txt_block {
            border: 1px solid red;
            width: 140px;
            height: 120px;
            padding-left: 10px;
            float: left;
            margin-right: 20px;
        }
    </style>
    <body>
        <h1>Arduino CSS Example 6</h1>
        <div class="txt_block">
            <p>A paragraph of text for this example.</p>
        </div>
        <div class="txt_block">
            <p>A    second    paragraph    of    text    for    this
example.</p>
        </div>
    </body>
</html>
```

The above markup produces the following web page.

The margin uses the same format as the padding, except that the margin puts space between the edge of the div to the outside of the div.

```
margin-right: 20px;
```

Margins can be applied individually to each side of the div or other HTML element:

```
margin-top: 5px;
margin-right: 3px;
margin-bottom: 7px;
margin-left: 10px;
```

And can also use the short format for specifying margins:

```
margin: 5px 3px 7px 10px;
```

Which applies to the HTML element in a clockwise order starting from the top:

margin: ***top right bottom left*****;**

Further Reading

CSS is a big topic and there is a lot more to learn, in fact there are whole books dedicated to CSS.

If you would like to learn more about CSS or need to find out how to do something specific for your own Arduino web server project, then read a good CSS book or search for more information on the Internet.

Arduino Ethernet Shield Web Server Tutorial Summary and Conclusion

Final part of the Arduino Ethernet Shield Web Server Tutorial

In this final part of the multi-part Arduino Ethernet shield web server tutorial, we summarise the technologies covered by the tutorial and provide some extra information. We also look at where to get further information on the various technologies and how to progress from here.

HTTP

The first technology to get familiar with when writing code for a web server that hosts HTML web pages is the Hypertext Transfer Protocol (HTTP). HTTP is normally invisible to the user of a web browser, except for **http://** in the browser address bar. HTTP is happening behind the scenes for the web surfer.

Because we are writing web server code on our Arduino, we need to know how HTTP works. When a web browser connects to an Arduino web server by a user entering the IP address in the address bar of the web browser (the IP address is set in the Arduino sketch), the web browser sends an HTTP request for the default or home page of the web server.

The Arduino web server must answer the HTTP request with the text that the protocol requires and then send the actual web page.

A good resource for learning more about HTTP is HTTP Made Really Easy by James Marshall.

Part 2 of this Arduino web server tutorial shows how HTTP is handled in the first Arduino web server sketch.

HTML and XHTML

Every web page consists of a text file written in Hypertext Markup Language (HTML). HTML structures the web page, marking which portion of the text are paragraphs, headers, checkboxes, buttons, etc.

There are actually several versions of HTML and also several versions of XHTML. XHTML is also used for creating web pages, but has a stricter syntax than HTML. For example, all (X)HTML tags must use lower-case letters in XHTML, but HTML can use upper or lower-case letters.

A web page can be written in any of the standards and could be HTML or XHTML – the person browsing a website would normally not know which standard the website is using unless he looked at the source code of a web page from the site.

HTML 5

We have actually been using HTML 5 in this tutorial. The version of HTML or XHTML used in a web page is specified by the very first line of markup in the web page. All the examples in this tutorial have used the following line at the top of the HTML page:

```
<!DOCTYPE html>
```

This lets the browser know that the web page that it is loading is written using HTML 5.

XHTML 1.0

Other HTML and XHTML standards use different markup for the first line of the web page. For example, strict and transitional XHTML use the following lines respectively (these are shown on two lines each here to fit on the page, the two lines are normally written on one line and separated by a space):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

HTML 4.01

The first line of markup for two of the HTML standards for HTML 4.01 are shown here (strict and transitional):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
```

```
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

CSS

Cascading Style Sheets (CSS) are used to style the HTML elements on a web page. This means that CSS can be used to change the font style, size and colour. The size position and colour of various HTML elements can be changed using CSS.

In our examples in this tutorial, the CSS was included as part of the HTML file and put between opening and closing **style** tags.

Normally a website will have a separate style sheet file (e.g. style.css) containing all the CSS. This allows the same style sheet to be used to apply styles to all the pages on the website.

Because we have mostly been using single web pages on the Arduino web server, it was easier to include the style in the web page. If more than one web page is needed using this method, then the same styles would need to be copied to the top of each page. The advantage of using a separate style sheet means that the style sheet only needs to be included in the top of each HTML page (the style sheet is included using a single line of markup).

In the relatively slow Arduino, a faster response will be obtained by including the CSS styles in the top of the web page when hosting a single web page. This is because only one HTTP request is then needed to get the page and CSS.

JavaScript and Ajax

We used JavaScript to implement Ajax for sending data to the Arduino from the web page and getting data back from the Arduino behind the scenes.

Ajax enables parts of the web page to be updated. This reduces the amount of data that needs be sent from the Arduino making the updates faster because the entire web page does not need to be reloaded every time new data is to be displayed on it.

Another advantage of using Ajax is that the web page does not flicker when data is updated as occurs when the entire page is refreshed.

HTML 5 Canvas

HTML 5 introduces the **canvas** tag. The canvas HTML element allows JavaScript to draw to the canvas area of the web page.

We saw the HTML canvas with JavaScript being used by the Gauge component written by Mykhailo Stadnyk in part 15 of this tutorial.

Only modern web browsers that support the relatively new HTML 5 standard will be able to run the gauge example.

Conclusion

Most of the technologies used to create a web page are rather large topics and have entire books dedicated to each one.

We have used a small subset of each of these technologies that have hopefully given you a start in creating your own Arduino web server and web pages.

How far you wish to pursue each of these subjects depends on what you want to implement on your web server and how good you want to get at each technology.

There are many resources available for web page design covering HTML, CSS, JavaScript and Ajax such as books and web sites – just do some searching on the Internet to find more detailed tutorials and references.