Microprocessor-Based Systems (E155)

D. Harris and M. Spencer **Fall 2014**

Lab 6: Internet of Things

Requirements

Build a internet accessible light sensor. Use a Raspberry Pi to run a web server (we recommend Apache2) and to measure a light sensing circuit using the analog to digital converter on your MuddPi board. The analog to digital converter must be controlled using the Pi's SPI peripheral, and the measured ADC voltage should be displayed on the internet using the web server on the Pi.

The web page displayed by the Pi must be able to blink MuddPi LEDs using the Pi (using the code given below) and also display the measured ADC voltage. Extra credit is available for improving your Pi's web site in some significant way. Examples of acceptable improvements include adding the ability to blink arbitrary LEDs on the MuddPi board, displaying the current state of the MuddPi LEDs, or enabling SSL in your Apache configuration to make your Pi more secure.

At heart, this lab requires you to directly control the memory mapped peripherals on the Pi, so refrain from finding libraries on the internet which have already spare you that work. Also note that the Pi runs an operating system, so software that runs on top of the operating system only has very weak guarantees about when each instruction will be run. This precludes software implementations of timing sensitive tasks like communications protocols: those are best left to hardware peripherals.

Apache2 Web Server

Broadly speaking, everything that you see on the internet is the product of one computer presenting text to another. The text is often formatted in a special, internet-specific, way that includes information about how to display it which is referred to as *hypertext*. (Forgive the early internet engineers this indulgences, I'm sure it sounded really cool at the time.) Hypertext is specified using a compact programming language called hypertext markup language or HTML. It is transferred over the internet based on a predefined set of agreements between all computers which is referred to as the hypertext transfer protocol or HTTP. The latter most of these acronyms should be familiar: whenever you type http:// into a web browser you are informing your computer that you are attempting to retrieve hypertext from the address that follows.

HTTP is complicated and servicing web service requests takes many steps. Fortunately, the tools necessary to do that are very mature. There are two common tools that interact

with HTTP: the web browser, which lives on a receiving computer, sends internet requests, and renders the received hypertext, and the web server, which listens for requests from the internet and sends out hypertext in response. You will be using a browser on a computer of your choice to access a web server that you install on your Raspberry Pi. If you are an expert, then feel free to use a web server of your choice. If not, then install Apache2 using the following command:

```
sudo apt-get install apache2
```

When the program finishes installing, restart your Pi for the changes to take effect. Then type your Pi's IP address into a browser. You should see Apache2's default page, which consists of the words "It Worked!" displayed. Note that the network infrastructure at Mudd is a bit weird, and it is likely that you won't be able to access your Pi's web server over Claremont-WPA even if you can SSH into it. We recommend using Ethernet connections.

There is a directory on your Pi located at the path `/var/www/html`, and this directory is referred to as you web root. Pages that you want your Pi to display can be put here. If you create an HTML (or plain text) file in this directory then you can access it at the web page `http://your.ip.address.here/filename.txt`. Note that `index.html` is the file which is displayed by default when there is no filename given after the IP address.

In order to display a webpage properly it is common to include a content-type and document type header at the top of the page to declare whether it is plain text, HTML of a particular version, or something else. Common examples of content headers can be found at the W3C website. The W3C stands for The World Wide Web Consortium, which is the governing body for internet standards. We mention the content-type and document-type headers because they feature in example code later.

It may seem unusual to access web pages using your IP address since we're used to typing human recognizable words into our browsers to find web pages. That luxury is provided by a piece of internet infrastructure called a domain name server, or DNS, which translates from human readable website names into the IP addresses that computers use to send information to each other. Your Pi does not have a DNS entry, so we enter the IP address of the device directly.

As a brief aside, all it takes to turn a computer into a device that provides a web service is web server software and a fixed IP address. If you have an old laptop or desktop, you can install Linux (Ubuntu is relatively easy to use), Apache2, and an IP/DNS manager (check out duckdns.org) in order to build a website of your very own.

The Common Gateway Interface

The web requests we've described so far, where a text file is returned from a distant computer to be displayed on your screen, only barely begin to provide the rich functionality available on today's internet. One key piece of functionality that is missing is the ability to provide dynamic content – content that varies from user to user or moment to moment – to users accessing your web server.

The modern internet provides many ways to do this, some of which even run on the user's computer. We'll be looking at a tried and true method of providing dynamic content which is called the common gateway interface or CGI. When using CGI, the web server doesn't deliver a static file in response to a web request as it did in our example above, instead it returns the output of a program which the server runs. We'll be writing programs in C to interface with the Pi, and those programs will return HTML formatted text that will be displayed on the internet.

CGI is not enabled by default in your Apache2 installation, so you need to adjust Apache2's configuration files. Specifically, the file `/etc/apache2/apache2.conf` controls most of the functionality of your web server. This master configuration file incorporates snippets of many lesser files, which have a conveniently modular organization. You can look at the modules and configuration options by observing `/etc/apache2/mods-available` and `/etc/apache2/conf-available`. We need to enable a CGI module, which we could do by writing a snippet of code in apache2.conf, creating symbolic links in the mods-enabled directory, or leveraging built-in apache commands to handle this for us. We'll choose the lattermost option because it is the highest level of abstraction; run the command:

```
sudo a2enmod,
```

and type "cgi" when it gives you a prompt. The changes to the Apache configuration won't take effect until the service is restarted with

```
sudo service restart apache2
```

(The old school way to manage services involves interacting with programs stored in `/etc/init.d/`. For instance, `/etc/init.d/apache2` restart has the same effect as that last command)

The default CGI root directory, the directory where CGI executables are stored, is `/usr/lib/cgi-bin`. Note that this is different than your web root. Keeping the web and CGI roots separate is an important security consideration. Since CGI programs are run by your computer, they can give a hacker a great deal of control over your system if he/she gains the ability to execute them. It is easier for a hacker to access your web root directory than your CGI root directory since your web root is publically accessible (by design), so storing CGI scripts there is dangerous. It is also wise to only put compiled executable files in the CGI directory for similar reasons: source code stored in the CGI directory could get manipulated to convince the computer to perform arbitrary tasks.

Some example code has been provided so that we can see this in action. They are on the class webpage, and one easy way to fetch them to your Pi is the `wget` command. `wget` accepts a URL as an argument and downloads that URL to the current directory.

The file `ledcontrol.html` is should be placed into your web root, and the compiled LEDON and LEDOFF binaries should be placed in your CGI root. `ledcontrol.html` contains elements called forms, which are used to activate the files LEDON and LEDOFF

through the CGI interface. LEDON and LEDOFF set GPIO pin 21 to be an output, raise it high or low, and then produce a tiny stub webpage using `printf` that redirects your browser back to `ledcontrol.html`. `printf` is a C function that prints strings to the "standard output" (STDOUT), which is usually a command line. It isn't in this case: the CGI protocol specifies that the web server should listen to STDOUT for the final web page to display. Connecting GPIO pin 21 to an LED (the LEDs on the MuddPi board are convenient since they already have series resistors) should allow you to control the LED from the internet when the programs are run.

However, when the CGI interface runs a program it does so using the username ascribed to Apache2: `www-data`. The `www-data` user has particularly restrictive permissions in order to prevent a hijacked web server from doing other things with your computer. This has ramifications for the kind of software that can be run using the CGI interface. In particular, the memory addresses used to control GPIO pins (and timers and other peripherals) are protected so that only the root user can access them. To bypass this protection, we are going to give the LEDON and LEDOFF binaries special permission to run as if a root user were invoking them by adjusting one of bits associated with the file's permissions: the superuser identity or SUID bit. Run the commands

```
sudo chown root:www-data /usr/lib/cgi-bin/{LEDON,LEDOFF}
sudo chmod 010 /usr/lib/cgi-bin/{LEDON,LEDOFF}
sudo chmod u+s /usr/lib/cgi-bin/{LEDON,LEDOFF}
```

The first of these commands makes sure that the `root` user and `www-data` group own the LEDON and LEDOFF files in the CGI root, the second grants the `www-data` group (and only the `www-data` group) permission to run them, and the third sets the SUID bit on those files so that the run with `root` user priveliges.

This is a security risk. Any user on your Pi with the right to run the programs in the CGI root directory now has the ability to control at least one GPIO pin. You can imagine that this is a terrifying prospect if that GPIO pin is connected to a million Watt piece of smelting equipment or a missile guidance system. This technique is suitable for this lab, but be sure to study your protocols and security vulnerabilities carefully before making anything into a product. Also note that most CGI scripts don't need the SUID bit set: if the script is just creating a webpage or accessing a `www-data` controlled database then the SUID bit is strictly a liability.

Note that compiling LEDON and LEDOFF depends on the GPIO.h header file, which sets up the GPIO memory map like you did in lab 5. Since you added extra functionality to your GPIO control library in lab 5 (timers, etc.), you should convert the LEDON.c and LEDOFF.c files to use your own library.

It is very possible that these programs won't work the first time due to permissions issues or other problems. Debugging these kinds of mistakes can be tremendously frustrating since the web server doesn't provide particularly transparent error information to your browser (again, by design and for security reasons). In order to find that error information, you should access the `/var/log/apache2/error.log` file: its entries are often

helpful in diagnosing problems with CGI scripts. It's also worth becoming familiar with common HTTP error codes: "403 Forbidden" suggests that the user has insufficient permission to run file and indicates a problem with your file permissions, "404 Not Found" suggests that the file name you're entering is wrong or the file has not been created, "500 Internal Server Error" usually means that there's a bug in your CGI script and that you should take a trip to the Apache error log.

Light Sensors, ADCs and the Internet of Things

The next step in finishing this lab is to create a new CGI script which accesses the ADC on the MuddPi board (the MCP3002, datasheet on the website). The ADC control pins are directly connected to the FPGA in anticipation of many possible final projects, but they can also be accessed at the female header on the bottom of the board. This female header connection allows you to access to the ADC power and control pins, which in turn allows you to connect the control pins to the Pi and the power pins wherever you see fit. The ADC has an SPI interface, so the Pi's SPI peripheral will need to be controlled by the CGI script you produce. We recommend using the SPI0 peripheral because that's what we tested.

Other features of HTML, either language constructs or more details about forms, may be helpful in creating aspects of your final webpage, you can find more information about them at the websites below:

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

The ADC should be attached to a light sensistive circuit in order to sense light in the room. One convenient tool for making this light sensitive circuit is the LPT2023 phototransistor. A light sensitive circuit can be created by creating a small circuit comprised of the LPT2023, a power source, and a resistor. You must design the circuit, so feel free to consult any resources you think will help. As usual, we recommend starting with the datasheet.

The final product of this lab is a crude member of an emerging class of devices called the Internet of Things. Proponents of these devices argue that everything – from your washing machine to your car to giant factories – should be connected to the internet so that the shared data can be used to optimize and improve societal functions. Energy distribution and monitoring is one domain where this is especially promising, so building a light sensor is a natural step into exploring this field.

Credits

This lab was originally developed in 2015 by Alex Alves '16.