

< PreviousNext >

Search ...

CGI using C++ on the BeagleBone (Ggicc)

Introduction

In Chapter 10 of my book (pg. 388-393), Exploring BeagleBone, I describe how you can build web-based CGI applications that can interface with electronics hardware that is attached to the BeagleBone using Bash scripts that call C/C++ programs. The solution works well for very straightforward applications, but this discussion investigates more advanced solutions for applications where there are more complex interactions — for example, the use of web forms to pass data between your web browser and the application that is executing on the BeagleBone. In this discussion I begin by explaining how you can use a C/C++ program, rather than a CGI script, to display a web page. I then investigate the use of the GNU Cgicc library for more structured and complex interactions.

The approach describe here will work on any Linux machine, including other embedded platforms such as the Raspberry Pi; however, the steps are structured for the BeagleBone platform and the examples that interact with the hardware are specific to the BeagleBone platform — they can however be easily modified. In this example, the Apache server is used to serve the CGI applications, so having Apache installed is the only prerequisite software required. Apache runs on port 8080 of the BeagleBone by default when using the recommended Debian images — you can test that by opening a web browser and entering the IP address of your Beaglebone in the address bar using the following format: `http://192.168.7.2:8080/` (replace the IP address with your BeagleBone's IP address — the one listed is the default IP address for Internet-over-USB).

It could be argued that this is a "dated approach" to solving the problem of having an embedded system web server interact with a web browser client — it has been around since the 1990's. To some extent that is true. There are powerful alternatives available such as Java servlets, node.js, Dart, PHP etc. However, this approach:

- has a low overhead on the BeagleBone, as the code is compiled rather than interpreted,
- permits access to system calls,
- can interface readily with hardware using the code libraries that I provide in the book.

The downside is that it is not really suitable for novice programmers, the output format syntax can be verbose and session management is complex. Even with that, it is worth pointing out that some large-scale web applications, including those by Google and Amazon, do use C++ on their servers for performance-critical systems. The BeagleBone is not a high-end server, so any performance optimizations are always welcome, perhaps even at the cost of complexity.

Page Contents [hide]

1 Introduction

2 Source Code for this Discussion

3 A Simple C++ Dynamic Web Page

3.1 Hello World (with uptime!)

4 GNU Cgicc (CGI for C++)

4.1 Installing GNU Cgicc

5 C++ Cgicc HTTP GET

5.1 The HTTP GET LED Controller

6 C++ Cgicc POST LED Example

7 Conclusions

New, June 2016!

The image shows the front cover of the book 'Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux' by Derek Molloy. The cover is primarily purple with a central photograph of a Raspberry Pi board. Text on the cover includes 'Exploring Raspberry Pi is THE book to go to if you are interested in learning about the impressive physical computing capabilities of the Raspberry Pi platform. Derek Molloy imparts the electronics, programming, and embedded Linux skills that are vital to today's innovators in building the next generation of Internet of Things applications.' and 'DEREK MOLLOY'. The publisher's name 'WILEY' is at the bottom right.

My new book on the Raspberry Pi. See: www.exploringrpi.com for further information. Buy on Amazon: (USA) (Canada) (Brazil) (UK) (Germany) (France) (Italy) (Spain) (China) (India) (Japan)

Source Code for this Discussion

GET SOURCE CODE

All of the code for this discussion is available in the GitHub repository for the book Exploring BeagleBone. The code can be viewed publicly at: the ExploringBB GitHub Chapter 10 directory, and/or you can clone the repository on your BeagleBone (or other Linux device) by typing:

```
1 molloyd@beaglebone:~$ sudo apt-get install git
2 molloyd@beaglebone:~$ git clone https://github.com/derekmolloy/exploringBB.git
```

A Simple C++ Dynamic Web Page

The first step is to write a simple C or C++ program that is capable of displaying a dynamically generated web page. This task is performed by a bash script in the book chapter, which is perfectly suitable but limiting, both in capability and performance.

Common Gateway Interface (CGI) is a straightforward approach for building dynamic web applications — effectively it allows a web server to share more than just HTML files and/or static images. It does this by allowing executable scripts/programs in a certain file system locations (e.g., `/usr/lib/cgi-bin/`) to be executed by the web server, and for the output from the scripts/programs to be passed, via the web server, to the web browser of the user that made the request. CGI allows the user's web browser to pass information (environment and application information) to the script/program using HTTP POST or GET requests. Almost all programming languages can be used to build CGI applications, as their only role in the transaction is to parse the input that is sent to them by the server, and to construct a suitable HTML output response.

On the BeagleBone, the `cgi-bin` directory requires root access permissions. There are a number of different ways of solving this problem and that is for another discussion. Since the BeagleBone is not typically a multi-user server, I am going to be liberal in my approach to security so that we do not get bogged down in detail. If you are planning to make your BeagleBone publicly visible on the Internet, and place it in control of your home automation system, then you must investigate the topic of server security.

Hello World (with uptime!)

To create a simple C++ CGI application the application can be deployed to the `/usr/lib/cgi-bin/` directory on the BeagleBone by default. This directory requires superuser permissions in order to add a script/program — that issue is dealt with shortly.

You can use the `nano` editor to create the C++ program. Remember that the command `"nano -c"` displays row/column numbering at the bottoms of the editor display — this is always useful for locating compilation errors. The following code example in Listing 1 can then be entered — it is also available in the exploringBB GitHub repository in the directory `/chp10/gcicc/` with the file name `hello.cpp`:

Listing 1: The Hello.cpp code that results in the hello.cgi CGI application

```
1 /* C++ CGI BeagleBone uptime example -- Written by Derek Molloy (www.derekmolloy.ie) */
2
3 #include <iostream>           // for the input/output
4 #include <stdlib.h>           // for the getenv call
5 #include <sys/sysinfo.h>      // for the system uptime call
6 using namespace std;
7
8 int main(){
9     struct sysinfo info;           // A structure that contains system stats
10    sysinfo(&info);                // retrieve the data
11    char *value = getenv("REMOTE_ADDR"); // The remote address CGI environment variable
12    cout << "Content-type:text/html\r\n\r\n"; // Generate the HTML output
13    cout << "<html><head>\n";
14    cout << "<title>EBB C++ Uptime</title>\n";
15    cout << "</head><body>\n";
16    cout << "<h1>BeagleBone System Uptime</h1>\n";
17    int mins = info.uptime / 60;    // the uptime comes from the sysinfo struct
18    int ram = info.freeram / 1024 / 1024; // the available memory in Mb
19    cout << "<div> The BBB system uptime is " << mins << " minutes.\n";
20    cout << "There is " << ram << " Mb of memory available.</div>\n";
21    cout << "<div> The CGI REMOTE_ADDR environment variable is " << value << "</div>";
22    cout << "</body></html>\n";
23    return 0;
24 }
```

This example uses the `sysinfo` structure to obtain the system uptime and available memory for the BeagleBone single-board computer. It also uses the environment variables to determine the IP address of the client browser

kernel LED LEDs linux LKM

Logic Analyzer makefile module nmap

opencv RTP SOURCE stepper motor sysfs

Tutorial UDP Video VLC Wordpress x264

Categories

> Analog

> Beaglebone

> Blog

> Digital Electronics

> Embedded Systems

> General

> Linux

> Main Blog

> Raspberry PI

> Tools

> Uncategorized

Recent Posts

> Writing a Linux Kernel Module — Part 3: Buttons and LEDs

> Writing a Linux Kernel Module — Part 2: A Character Device

> Writing a Linux Kernel Module — Part 1: Introduction

> Introduction to CMake by Example

> CGI using C++ on the BeagleBone (Ggicc)

Archives

> April 2015

> March 2015

> June 2014

> January 2014

> December 2013

> November 2013

> October 2013

> July 2013

> June 2013

that made the request to execute the CGI program. The following environment variables are available (sourced from CgiEnvironment.cpp):

> May 2013

> April 2013

CGI Environment Variables			
variable	variable	variable	variable
SERVER_SOFTWARE	SERVER_NAME	GATEWAY_INTERFACE	SERVER_PROTOCOL
SERVER_PORT	REQUEST_METHOD	PATH_INFO	PATH_TRANSLATED
SCRIPT_NAME	QUERY_STRING	REMOTE_HOST	REMOTE_ADDR
AUTH_TYPE	REMOTE_USER	REMOTE_IDENT	CONTENT_TYPE
CONTENT_LENGTH	HTTP_ACCEPT	HTTP_USER_AGENT	REDIRECT_REQUEST
REDIRECT_URL	REDIRECT_STATUS	HTTP_REFERER	HTTP_COOKIE

You can compile the code using the following steps:

```
molloyd@beaglebone:~/exploringBB/chp10/cgicc$ ./build
Building the hello.cgi C++ CGI Program
...
molloyd@beaglebone:~/exploringBB/chp10/cgicc$ ls -al
total 24
drwxr-xr-x  2 molloyd molloyd 4096 Mar 25 00:00 .
drwxr-xr-x 15 molloyd molloyd 4096 Mar 24 23:44 ..
-rwxr-xr-x  1 molloyd molloyd   85 Mar 24 23:46 build
-rwxr-xr-x  1 molloyd molloyd 7204 Mar 25 00:00 hello.cgi
-rw-r--r--  1 molloyd molloyd 1193 Mar 25 00:00 hello.cpp
molloyd@beaglebone:~/exploringBB/chp10/cgicc$ sudo cp hello.cgi /usr/lib/cgi-bin/
```

The build script executes the command 'g++ hello.cpp -o hello.cgi'. Please note that the copy command is executed with superuser permissions. This is required as otherwise it would not have the access level required to create the binary executable (**hello.cgi**) in the **/usr/lib/cgi-bin/** directory.

This CGI program can then be called remotely from the web browser on the desktop computer, using the URL <http://192.168.7.2:8080/cgi-bin/hello.cgi>, as illustrated in Figure 1 below.

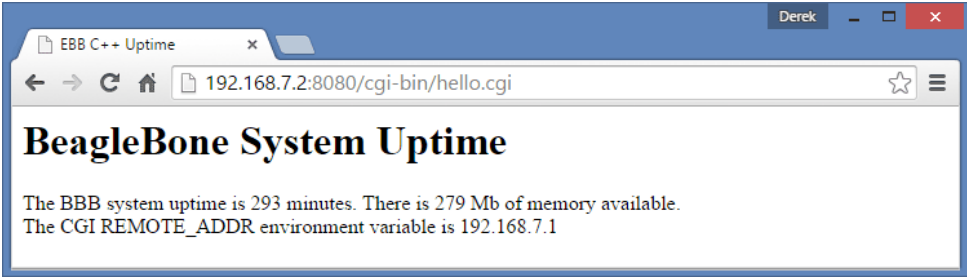


Figure 1: A simple C++ CGI application display

The **.cgi** extension is not a requirement for the executable CGI program, but it is useful here for clarity. For example, the executable could have been named **/usr/lib/cgi-bin/hello** and the address that is used in the web browser client would be: **http://192.168.7.2:8080/cgi-bin/hello**



GNU Cgicc (CGI for C++)

Unfortunately, the approach that is described above is only really suitable for programs that make information available to the Internet. By default, C/C++ do not have the built-in libraries required to easily and effectively build interactive CGI applications — for this we need the Cgicc C++ class library.

The GNU Cgicc is a C++ library for building CGI applications. It is powerful and it greatly simplifies the process of building common gateway interface (CGI) applications — those applications that allow you to interact with the BeagleBone over the Internet using a simplified interface within a web browser. As stated, there are other, more recent, approaches to building web applications, but GNU Cgicc offers an efficient high-performance solution.

Installing GNU Cgicc

To avoid complexity, this discussion assumes that you are building the C++ CGI program directly on the BeagleBone itself. Therefore, the first task is to download, compile and install the GNU Cgicc library on the BeagleBone:

Step 1: Download the Ggicc library

The first step is to download the source code for the Cgicc library. You can find the latest version at: <ftp://ftp.gnu.org/gnu/cgicc/>. At the time that this discussion was written the most recent version is 3.2.16. It is advisable to choose the most recent version and download it to a user account on the BeagleBone, as follows (performing these steps as root is fine too):

```
molloyd@beaglebone:~$ mkdir cgicc
molloyd@beaglebone:~$ cd cgicc
molloyd@beaglebone:~/cgicc$ wget ftp://ftp.gnu.org/gnu/cgicc/cgicc-3.2.16.tar.gz
...
2015-03-24 00:53:55 (509 KB/s) - 'cgicc-3.2.16.tar.gz' saved [1409037]
molloyd@beaglebone:~/cgicc$ tar xvf cgicc-3.2.16.tar.gz
...
cgicc-3.2.16/doc/lib-overview.tpl
cgicc-3.2.16/doc/COPYING.LIB
cgicc-3.2.16/doc/COPYING
cgicc-3.2.16/doc/Makefile.am
```

Step 2. Configure the source code makefiles

The final installation directory (for **make install**) should be configured to be **/usr**. This will ensure that the compiled library is made available to all users.

```
molloyd@beaglebone:~/cgicc$ cd cgicc-3.2.16
molloyd@beaglebone:~/cgicc/cgicc-3.2.16$ ./configure --prefix=/usr
...
config.status: creating doc/Makefile
config.status: creating doc/Doxyfile
config.status: creating cgicc.pc
config.status: creating cgicc/config.h
config.status: executing depfiles commands
config.status: executing libtool commands
```

Step 3 Make the project

This step takes approximately five minutes in order to build the library on the BeagleBone. It is initiated by typing **make** in the download directory:

```
molloyd@beaglebone:~/cgicc/cgicc-3.2.16$ make
Making all in cgicc
make[1]: Entering directory '/home/molloyd/cgicc/cgicc-3.2.16/cgicc'
make all-am
make[2]: Entering directory '/home/molloyd/cgicc/cgicc-3.2.16/cgicc'
/bin/bash ../libtool --tag=CXX --mode=compile g++ -DHAVE_CONFIG_H -I. -x c++ -Wall -W -pedantic -g -O2 -MT libcgicc_la-CgiEnvironment.lo -MD -MP -MF .deps/libcgicc_la-CgiEnvironment.Tpo -c -o libcgicc_la-CgiEnvironment.lo 'test -f 'CgiEnvironment.cpp' || echo './'CgiEnvironment.cpp'
...
make[1]: Leaving directory '/home/molloyd/cgicc/cgicc-3.2.16/contrib'
make[1]: Entering directory '/home/molloyd/cgicc/cgicc-3.2.16'
make[1]: Nothing to be done for 'all-am'.
make[1]: Leaving directory '/home/molloyd/cgicc/cgicc-3.2.16'
```

Step 4. Install the library on the BeagleBone

The final step is to install the compiled library for all users on the BeagleBone. Ensure that you execute **make install** with superuser permissions, otherwise it will fail to place the library files correctly.

```
molloyd@beaglebone:~/cgicc/cgicc-3.2.16$ sudo make install
[sudo] password for molloyd:
Making install in cgicc
...
Libraries have been installed in: /usr/lib
...
molloyd@beaglebone:~/cgicc/cgicc-3.2.16$ cd /usr/lib
molloyd@beaglebone:/usr/lib$ ls libcgi*
libcgicc.a libcgicc.so libcgicc.so.3.2.10 libcgicc.so.5.0.2
libcgicc.la libcgicc.so.3 libcgicc.so.5
```



C++ Cgicc HTTP GET

The first example that uses the Cgicc library is a simple HTTP GET request-response example. A GET is used to request data from the server resource, whereas a POST is used to submit data to be processed by the server resource. The general structure of a GET request is to use name/value pairs to identify the functionality that you would like to action. For example, a generalized GET request could be described as:

```
http://my.beaglebone.com/cgi-bin/mygetapp?name1=value1&name2=value2
```



This would invoke the application "mygetapp" on the BeagleBone and pass the two variables, *name1* and *name2*, that have the values *value1* and *value2* respectively. This approach is quite straightforward and it is easy to construct request strings to be actioned from scripts or other applications. However, this approach should never be used to send sensitive information (e.g., passwords), as the URLs will be visible in server logs, browser histories etc.

The HTTP GET LED Controller

The test application is an LED controller that is very similar to the one that is described in Chapter 5 of the book. The on-board system LEDs can be controlled using the C++ LED class that is available to support that chapter.

Figure 2 illustrates the application that results from the code in Listing 2. You can see in the address bar that the URL has the form ".../getLED.cgi?command=on" — the request URL consists of one *name* "command" that has the *value* "on". By changing this value we can control the behaviour of the USR3 LED that is on the top left-hand corner of the BeagleBone PCB. After this request is sent, the USR3 LED lights constantly.



Figure 2: A C++ CGI GET application (on command)

The code for this example is presented in Listing 2 below. The code example uses the LED.h code that is used in Chapter 5. For convenience it is replicated in this project directory.

Listing 2: The Ggicc HTTP GET LED example code

```
1 /* C++ CGI BeagleBone GET example -- Written by Derek Molloy (www.derekmolloy.ie)
2 You must set the setuid bit for this script in order that it can
3 access the on-board LED sysfs file system. See the web page for instructions.
4 */
5
6 #include <iostream>           // for the input/output
7 #include <stdlib.h>           // for the getenv call
8 #include <sys/sysinfo.h>      // for the system uptime call
9 #include <cgicc/Cgicc.h>      // the cgicc headers
10 #include <cgicc/CgiDefs.h>
11 #include <cgicc/HTTPHTMLHeader.h>
12 #include <cgicc/HTMLClasses.h>
13 #include "LED.h"              // the LED class from Chapter 5 of the book
14 using namespace std;
15 using namespace cgicc;
16
17 int main(){
18     Cgicc form;                // The Cgicc object
19     LED *led3 = new LED(3);    // The LED object -- USR3
20
21     // Generate the response HTML page
22     char *value = getenv("REMOTE_ADDR"); // the remote address CGI env. variable
23     cout << "Content-type:text/html\r\n\r\n"; // generate the HTML output
24     cout << "<html><head>\n";
25     cout << "<title>EBB C++ GET Example</title>\n";
26     cout << "</head><body>\n";
27     cout << "<h1>BeagleBone GET Example</h1>\n";
28
29     form_iterator it = form.getElement("command"); // read the URL get command string
30     if (it == form.getElements().end() || it->getValue()==""){
31         cout << "<div> The LED command is missing or invalid.</div>";
32         cout << "<div> Valid commands are on, off, flash, and status </div>";
33     }
34     else{
35         string cmd(**it);
36         cout << "<div> The LED command is " << cmd << ".</div>";
37         /** This code sets the USR3 LED state using the LED class **/
38         if(cmd=="on") led3->turnOn();
39         else if(cmd=="off") led3->turnOff();
40         else if(cmd=="flash") led3->flash("100");
41         else if(cmd=="status"){
42             cout << "<div>";
43             led3->outputState();
44             cout << "</div>";
45         }
46         else cout << "<div> Invalid command! </div>";
47     }
48     cout << "<div> The CGI REMOTE_ADDR environment variable is " << value << "</div>";
49     cout << "</body></html>\n";
50     return 0;
51 }
```

The code has some interesting features. The `form_iterator` `it` is used to determine the *value* associated with the **command** *name* of the name/value pair. A check is performed to test that the element is not missing or null. If it is then a usage message is presented to the user. Otherwise, a string object `cmd` is created in place of `**it` in order to tidy up the syntax — this is not strictly necessary but it makes the code more legible. The `cmd` is then compared to the four possible options and the correct LED state is triggered. The program can be compiled using the call:

```
g++ getLED.cpp LED.cpp -o getLED.cgi -lcgicc
```

where the `getLED.cpp` and `LED.cpp` files are passed to the compiler, along with the `-l` flag to link with the `cgicc` library. To deploy the resulting binary (`getLED.cgi`) the `deploy` script uses the following calls:

```
sudo cp getLED.cgi /usr/lib/cgi-bin/
sudo chmod +s /usr/lib/cgi-bin/getLED.cgi
```

The `chmod +s` call sets the setuid bit — effectively the `getLED.cgi` binary is now executed with root permissions by Apache. If you perform a `ls -l` in the `/usr/lib/cgi-bin` directory you will see the `"s"` bit set on the binary to indicate that the **setuid** mode is enable.

```
molloyd@beaglebone:/usr/lib/cgi-bin$ ls -l
total 64
-rwsr-sr-x 1 root root 19924 Mar 26 2015 getLED.cgi
```

This is necessary because the LED code requires root access in order to modify the state of the on-board LEDs. Importantly, if this were a CGI script, I would be quite concerned at this point about script code injection. However, this is a CGI program that does not parse input fields. That said, it is not ideal and udev rules (see page 247 in the book) are a better solution for non-root resource access.

The code in Listing 2 does not take full advantage of the `Cgicc` library — for example, it still manually generates the header output. That will be further improved in the next example — so please keep reading, even if you only require HTTP GET functionality.

Figures 3 below illustrates the other functionality that is available in this application. You can trigger the LED to flash at a fixed frequency, or you can request information from the application about the current status of the LED that is made available via `sysfs`. All of these changes have an immediate effect on the `USR3` LED and it behaves as you would expect.

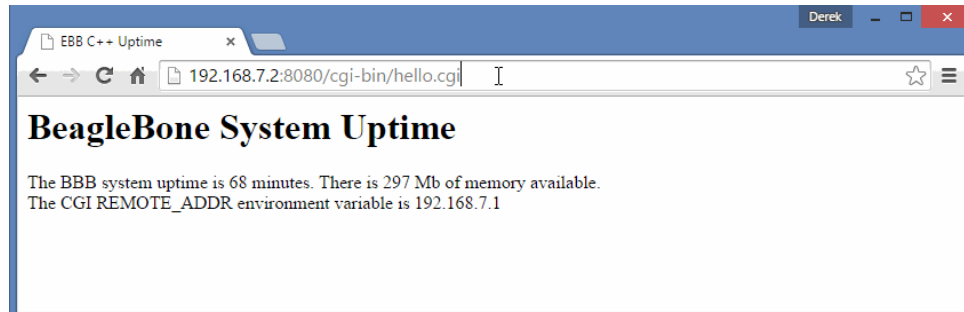


Figure 3: A C++ CGI GET application (animation of all commands)



C++ Cgicc POST LED Example

The second example that uses the `Cgicc` library is a HTTP POST example. The POST example allows you to interact with a form that contains checkboxes, radio components, buttons, text fields etc. For example you could use the HTML code in Listing 3 to display a form that passes data to the `Cgicc` C++ application. That is not required in this example, as the `postLED.cgi` application generates this code dynamically and adapts the rest of the page to represent the current state of the LED. The operation of this example is best understood by watching the animation in Figure 4.

Listing 3: Example HTML code (this code is not used, rather it is generated dynamically by the CGI C++ application)

```
1 <html><head><title>EBB C++ Post LED Example</title></head>
2 <body><h1>BeagleBone POST LED Example</h1>
3 <form action="/cgi-bin/postLED.cgi" method="POST">
4 <div>LED state:
5 <input type="radio" name="command" value="on"/> On
6 <input type="radio" name="command" value="off" checked="" /> Off
7 <input type="radio" name="command" value="flash"/> Flash
8 <input type="checkbox" name="status" /> Display Status </div>
9 <div>Flash period: <input type="text" name="period" size="6" value="100"> ms
10 <input type="submit" value="Execute Command" /></div></form>
11 </body></html>
```

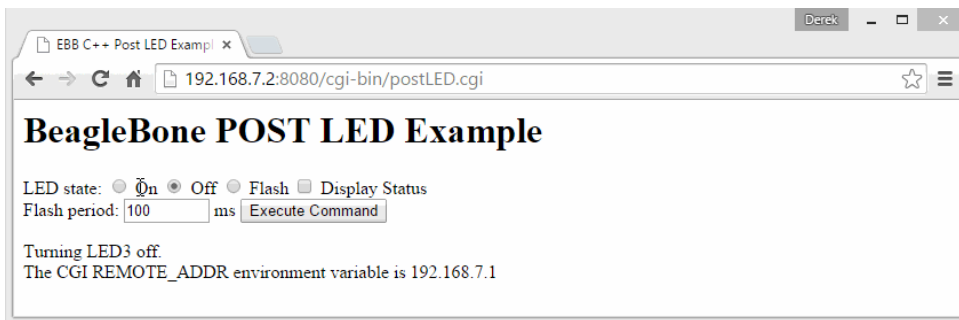



Figure 4: A C++ CGI POST application (animation of all functionality)

You can see that this application changes its output to display the state of the LED, and therefore it appears to be *stateful*. That is only possible in this example because the HTML form "stores" the "application" state — on each occasion you click the **Execute Command** button the **postLED.cgi** application executes afresh and runs to completion, thus losing its state. The expected state is only retained by the HTML form. The full source code for the Cgicc HTTP POST example is presented in Listing 4 below. It is important to note that this is the only source code required! The application generates the HTML form, which calls itself, over and over again. Just like the **getLED.cgi** application, the LED3 output changes instantly and appears as you would expect.

Listing 4: The Cgicc HTTP POST example code

```
1  /* C++ CGI BeagleBone POST example -- Written by Derek Molloy (www.derekmolloy.ie)
2  You must set the setuid bit for this script in order that it can
3  access the on-board LED sysfs file system. This example integrates the form
4  for ease of use -- i.e., it is a form that calls itself. It could just as
5  easily receive the input from a regular HTML page.
6  */
7
8  #include <iostream>           // for the input/output
9  #include <stdlib.h>           // for the getenv call
10 #include <sys/sysinfo.h>      // for the system uptime call
11 #include <cgicc/Cgicc.h>      // the cgicc headers
12 #include <cgicc/CgiDefs.h>
13 #include <cgicc/HTTPHTMLHeader.h>
14 #include <cgicc/HTMLClasses.h>
15 #include "LED.h"              // the LED class from Chapter 5 of the book
16 using namespace std;
17 using namespace cgicc;
18
19 int main(){
20     Cgicc form;                // the CGI form object
21     LED *led3 = new LED(3);     // the LED object -- USR3
22     string flashPeriod, command; // default values
23
24     bool isStatus = form.queryCheckbox("status"); // get the state of the status checkbox
25     form_iterator it = form.getElement("period"); // get the period text value
26     if (it == form.getElements().end() || it->getValue()==""){
27         flashPeriod = "100"; // if it is invalid use 100
28     }
29     else { flashPeriod = it->getValue(); } // otherwise use submitted value
30     it = form.getElement("command"); // get the radio command chosen
31     if (it == form.getElements().end() || it->getValue()==""){
32         command = "off"; // if it is invalid use "off"
33     }
34     else { command = it->getValue(); } // otherwise use submitted value
35     char *value = getenv("REMOTE_ADDR"); // The remote address CGI env. variable
36
37     // Generate the form but use states that are set in the form that was submitted
38     cout << HTTPHTMLHeader() << endl; // Generate the HTML form using cgicc
39     cout << html() << head() << title("EBB C++ Post LED Example") << head() << endl;
40     cout << body() << h1("BeagleBone POST LED Example") << endl;
41     cout << "<form action=\"cgi-bin/postLED.cgi\" method=\"POST\">\n";
42     cout << "<div>LED state: <input type=\"radio\" name=\"command\" value=\"on\" \"
43     << ( command==\"on\" ? \"checked\":\"\") << \"/> On "; // is the command="on"?
44     cout << "<input type=\"radio\" name=\"command\" value=\"off\" \"
45     << ( command==\"off\" ? \"checked\":\"\") << \"/> Off ";
46     cout << "<input type=\"radio\" name=\"command\" value=\"flash\" \"
47     << ( command==\"flash\" ? \"checked\":\"\") << \"/> Flash ";
48     cout << "<input type=\"checkbox\" name=\"status\" \" << (isStatus ? \"checked\":\"\")
49     << \"/> Display Status </div>";
50     cout << "<div>Flash period: <input type=\"text\" name=\"period\" size=\"6\" value=\"\" \"
51     << flashPeriod << "\> ms "; // populate the text field
52     cout << "<input type=\"submit\" value=\"Execute Command\" />";
53     cout << "</div></form>";
54
55     // Process the form data to trigger the LED state
56     if (command=="on") led3->turnOn(); // turn the LED on?
57     else if (command=="off") led3->turnOff(); // off?
58     else if (command=="flash") led3->flash(flashPeriod); // flash with the period above
59     else cout << "<div> Invalid command! </div>"; // not possible at the moment
60     // If the Display Status checkbox is checked then display the status now
61     // this should happen after the command is set, otherwise it is old data
62     if (isStatus){
63         cout << "<div>";
64         led3->outputState();
65         cout << "</div>";
66     }
```

```

67     cout << "<div> The CGI REMOTE_ADDR environment variable is " << value << "</div>";
68     cout << body() << html();
69     return 0;
70 }

```

This program has some interesting features. In particular, it uses functions such as HTTPHTMLHeader(), html(), body() etc. to generate the HTML content for the output. This is much less verbose than in Listings 1 and 2 and is less prone to error. There are many more such functions that can be used to further clean up the code. The code example also demonstrates how to interact with radio buttons (**command**), checkboxes (**status**), and text inputs (**period**) within HTML forms.

It is important that the form data is parsed at the top of the program. This is because the form data that was previously submitted needs to be propagated into the new output. Clearly, the first time this form is requested there will be no data present and the code at the beginning of the program assigns default value (e.g., **flashPeriod="100"**, **command="off"**). If this is not performed then the program will suffer from segmentation faults. From that point onwards, the form output needs to maintain the state and that is why these values appear in the HTML generation code. For example,

```

1 (command=="on" ? "checked":"" )

```

is a clever piece of code that compares the command string to the string "on". If they are equal then the word "checked" appears in the HTML form at that point in the code (thanks to the output stream operator <<); however, if they are not equal then the string "" (nothing) appears at that point in the HTML code. This allows a radio item or checkbox item to remain checked or unchecked. Finally towards the end of the program the command is processed and the code to display the status is placed.

The ? is called the *conditional operator* or often the *ternary operator* (the latter is slightly incorrect as it is a ternary operator, as it takes three operands, but it may not be the only ternary operator in all languages). An example of this operator in C++ is:

```
cout << (s.length() < t.length()) ? s : t) << endl;
```

This line of code will display the shorter of the strings *s* and *t*.

Conclusions

This discussion has just scratched the surface on what can be performed using CGI and C++ on the BeagleBone. For very complex applications you may be better placed to examine other frameworks, but for simple high-performance web interfaces, the GNU Cgicc library provides a very appropriate solution. There are several limitations with the current solution. It is a single session solution -- If two users, on two different machines, access the **postLED.cgi** script at the same time then very strange things will happen! For example, each browser will store an independent state that is likely to be contradictory. However, given that there is only one USR3 LED on the BeagleBone PCB that is not a very significant issue. For more complex applications, session management is important.

For more information on the Cgicc library please see the GNU cgicc library documentation. You will see that the library is capable of handling cookies, file transfers and much more by browsing the Class List.

By Derek | March 27th, 2015 | Beaglebone, Embedded Systems, Linux, Raspberry PI | 4 Comments

Share This Story, Choose Your Platform!



About the Author: Derek



Dr. Derek Molloy is a senior lecturer in the School of Electronic Engineering, Faculty of Engineering and Computing, Dublin City University, Ireland. He lectures at undergraduate and postgraduate levels in object-oriented programming with embedded systems, digital and analog electronics, and 3D computer graphics. His research contributions are largely in the fields of computer and machine vision, 3D graphics, embedded systems, and e-Learning. This is his personal blog site.

Related Posts

<

>

4 Comments

**foikei** July 8, 2015 at 10:17 pm - Reply

hi Derek,

you taught me much things thru your videos. Your book shows very good the capabilities of this technology. Extending the book by a website with further information is great concept. I don't want to know, how much time this project took and will take.

I never thought that i am able to understand how to turn on a led thru a mobile on my BBB or to write code on my desktop and deploy it automatically to another. I have to thank you very, very much!

greetings

foikei

**Derek** July 13, 2015 at 1:21 am - Reply

Thanks for your support Foikei, It took a long time to put everything in place! Great to hear that it has made a difference! Derek.

**Amir Nouri Nia** September 17, 2015 at 5:25 pm - Reply

Hi Derek, I just wanted to say, Thank you! 😊 These stuffs are brilliant. Not that I learned to implement my C++ programming skills in server side, I was able to develop a Java program that runs on Android, which I will use it in my thesis for my B.Sc..

You Helped me a lot with your book and your website and videos. Frankly speaking, you helped me to become the Geek (Person) I wanted to be!

Again, adore your book and your website and your youtube videos.

Best Wishes

Amir Nourinia

**Amritha** September 13, 2016 at 2:41 pm - Reply

Hi,

I followed the steps for configuring cgicc libraries. But while doing the make step i end up with the following error:

Making install in cgicc

make[1]: Entering directory '/path/to/cgicc-3.0.2/cgicc'

/bin/sh ../libtool --mode=compile c++ -DHAVE_CONFIG_H -I. -I. -I. -I. -g -O2 -c CgiUtils.cc

rm -f .libs/CgiUtils.lo

c++ -DHAVE_CONFIG_H -I. -I. -I. -I. -g -O2 -c -fPIC -DPIC CgiUtils.cc -o .libs/CgiUtils.lo

CgiUtils.cc:111:30: error: 'string' does not name a type

CGICCNString(unescapeString(const string& src)

^

CgiUtils.cc:111:41: error: 'std::_cxx11::string cgicc::unescapeString(const int&)' should have been declared inside 'cgicc'

CGICCNString(unescapeString(const string& src)

^

CgiUtils.cc: In function 'std::_cxx11::string cgicc::unescapeString(const int&):'

CgiUtils.cc:117:18: error: request for member 'begin' in 'src', which is of non-class type 'const int'

for(iter = src.begin(); iter != src.end(); ++iter) {

^

CgiUtils.cc:117:39: error: request for member 'end' in 'src', which is of non-class type 'const int'

```
for(iter = src.begin(); iter != src.end(); ++iter) {  
  ^  
  Makefile:229: recipe for target 'CgiUtils.lo' failed  
  make[1]: *** [CgiUtils.lo] Error 1  
  make[1]: Leaving directory '/path/to/cgicc-3.0.2/cgicc'  
  Makefile:161: recipe for target 'install-recursive' failed  
  make: *** [install-recursive] Error 1
```



James Dixon June 19, 2017 at 6:30 am - Reply

Hi,

I tried the first example (the "hello.cgi" one) and seemed to be stuck.

When trying to connect to "IPADDRESS/cgi-bin/hello.cgi" it would not be able to connect and I would get a 404 error.

I feel like I am missing out a step with the configuration of the Apache2 server. Was there some pre configuring that was required?

Thanks!

Leave A Comment

Name (required)

Email (required)

Website

Comment...

The "monster" image that is associated with your comment is auto-generated -- it makes it easier to follow the conversation threads. If you wish to replace this image with a less (or perhaps more) monstrous version, add an image at Gravatar.com against the e-mail address that you use to submit your comment. Your image will henceforth be used on most WordPress sites. Please note that I will remove any messages that contain blatant advertisement or that refer to illegal software, content etc. I may tidy up some messages if they contain code dumps etc. E-mail addresses are used only to notify you of any responses, and to authenticate your future comments on this website -- they are not made public nor used for any other purpose. See the Privacy and Cookie Policy for a full description. I manually approve all new posts in order to keep the website spam free, but once your post is approved, all future posts should be automatically approved. Please let me know if your messages do not appear. I really appreciate it when you answer the questions of others on the page, as it is difficult for me to do so and continue to produce new content. Thanks for your understanding, Derek.

Post Comment

ABOUT:

This site brings together all of the video content on the Derek Molloy YouTube channel and structures it so that you can follow the videos as lessons. It also integrates associated documentation, datasheets and tools to allow you to get the best from the video series. It also has a blog to allow me to post new videos, articles and useful information that may not be in video form

RECENT COMMENTS

Viking on [Resize a VirtualBox guest Linux VDI Disk under Windows Host](#)

Steve on [Resize a VirtualBox guest Linux VDI Disk under Windows Host](#)

Roger on [Beaglebone: Video Capture and Image Processing on Embedded Linux using OpenCV](#)

Alexey on [Resize a VirtualBox guest Linux VDI Disk under Windows Host](#)

DEREK MOLLOY YOUTUBE







Derek Molloy

YouTube45K

RECENT TWEETS

- 

RT @mullrheimhne: Major congrats to @Team_Eirloop on their fantastic performance at the @SpaceX @Hyperloop Pod Competition, taking 5th plac...
2 months ago
- 

RT @DCUEngineering: We were delighted to welcome Minister of State for Training, Skills, Innovation, Research and Development @JohnHalligan...
2 months ago

Follow @DerekMolloyIE847 followers