

Barracuda Embedded Web-Server whitepapers

Please note that the PDF version is for printing a hardcopy. Please see the online whitepapers for the interactive examples.

Introduction to the Barracuda Embedded Web-Server.....	3
Introduction to HTTP.....	3
Using a Web Server in an embedded device	4
Using a Web Server for regression tests	5
Using a browser to control an embedded device	5
Introduction to CSP.....	6
Creating dynamic user interfaces using CSP.....	8
Sending data from a browser to the server.....	12
"Hacking" a Web Server using telnet	14
Rich Client Interface.....	15
The EventHandler	17
Conclusion	17
References.....	17
Device control with Barracuda	18
Use Barracuda to remotely set and get the system time	18
Assembling the web-server.....	19
The Barracuda SMX task.....	19
Creating the set/get time page class	22
Dynamically create the web-interface	23
Setting a new time in the device	25
Conclusion	26
The CSP Hangman game.....	28
Designing the hangman framework	29
Designing the C code infrastructure	31
Writing the C code	32
Replacing the HTML comments with CSP tags.....	34
Creating the form and maintaining the state variables.....	36
How it works.....	37
Conclusion	38
Exercise.....	38
Introduction to web-security and the Barracuda Virtual File System ...	39
Preventing eavesdropping and modification of data.....	39

Authenticating users.....	40
Authorizing users	41
The virtual file system	43
The Barracuda security manager	45
Conclusion	46
A Trace Tool Using the EventHandler.....	47
Designing Rich Client User Interfaces	48
Writing and testing the initial code without using Barracuda.....	49
Creating the interface definition file (IDL).....	50
Adding the client side EventHandler code.....	50
ANSI C and object-oriented programming.....	51
Downloading the example code.....	51
The server side implementation of the Simple Debugger.....	52
Conclusion	53

Introduction to the Barracuda Embedded Web-Server

This paper covers fundamental concepts of HTTP and how the Barracuda Embedded Web Server can be used in an embedded device.

Depending on your knowledge of the various technologies covered, it may not be necessary to read each section, although each section also presents some of Barracuda's unique features. This paper also assumes a basic understanding of HTML, C and C++.

This article is a PDF version of our online whitepaper at:

<http://barracudaserver.com/WP/intro/>

Introduction to HTTP

The Hyper Text Transfer Protocol(HTTP) is a text-based Remote Procedure Call (RPC) protocol that can transfer any type of data between the client and server. An HTTP client opens a connection and sends a request message to an HTTP server. The server then returns a response message.

The HTTP header contains an initial line and a number of header lines followed by an optional body. The initial line of the request and the response are different. In a request, the initial line contains the HTTP method and the requested resource, while in a response, it contains the response protocol version, a status code, and a status message. The most common HTTP methods, the RPC type, are GET and POST, where GET usually means "Server, please give me this resource" and POST usually means "Server, here is my data", though this should be decided by the resource.

A good Web Server should treat the requested resource as an executing unit and let the executing unit decide what to do with the request command. A good web-server should also allow the resource to announce the HTTP methods supported by the resource when upon a client request.

Example HTTP command, setting the refrigerator and freezer temp:

```
POST https://embeddedwebserver.net/refrigerator/temperature/ HTTP/1.1
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

fridgeTemp=5&freezerTemp=-30
```

The Barracuda embedded Web Server treats all resources as executing units. A Barracuda executing unit can be a physical resource such as a page. A virtual resource can also be a container that can contain other executing units. An executing unit is just a plugin to the web-server. The Barracuda embedded web-server comes with a number of executing units such as a directory unit that can directly read from a ZIP file and present the content to a browser client. Other plugins, such as the EventHandler, can be installed as well. The above HTTP command example sends two parameters, fridgeTemp and freezerTemp to an executing unit in the Barracuda Virtual File System at location "/refrigerator/temperature/". A Directory Executing Unit is called HttpDir and a Page Executing Unit is called HttpPage. An HttpPage and HttpDir can be extended, or in Java terminology, you can extend the base classes and implement your own "live resource".

The Barracuda Web Network Management plugin, WNMP, would have been impossible to design without the Barracuda Directory Executing Unit. The Barracuda Web Server platform with its support for Executing Resources makes it easy to design advanced plugins.

Using a Web Server in an embedded device

A Web Server in one device is typically used for remote management of one or more devices. A client does not necessarily have to be a browser. Any client implementing the HTTP protocol stack can be used to control the device. The HTTP protocol is good for sending anything to the device, from control data to exchanging user data, and even uploading new software releases. A client HTTP library can be linked into a C or C++ client application or a scripting with native HTTP support, such as Python, can be used.

There are many reasons for using HTTP as a general protocol between a client and a server, the device. An application using the HTTP protocol benefits from services provided by HTTP such as:

- authentication and authorization support
- Encryption by using SSL
- Bypass firewall restrictions

A resource is addressed by its URL, or the path element of the URL. For example, we used the path "/refrigerator/temperature/" above to address the temperature object in the server. One can think of a resource as an object with a number of methods. The methods can be the standard HTTP methods such as GET, POST, PUT etc, but HTTP is very flexible and you have no such constraints when designing a Barracuda executing unit. You are free to interpret the HTTP methods to anything you like as long as your client and server understand the data being exchanged.

Using a Web Server for regression tests

The HTTP protocol is becoming increasingly popular for exchanging data. You can even find Open Source HTTP client libraries that can be link with C or C++ host applications. Most scripting languages also come with native HTTP support. Python is one such scripting language. During development, internal objects and methods in your embedded system can be exposed using a Barracuda Executing Unit. Each Executing Unit is installed as a resource in the Barracuda Virtual File system, thus each object you want to expose can have its own URL.

A scripting language such as python simplifies writing advanced regression tests. Data sent to the server can be encoded as "application/x-www-form-urlencoded" data as shown above. URL encoded data, is so common in HTTP that the Barracuda Web Server automatically decodes all of the data in the URL encoded stream. URL encoded data in HTTP is much like the parameters passed to a C function.

However, exchanging complex data structures is difficult with URL-encoded data, so for these some serialization layer must be used. Fortunately, there are platform and language independent standards available that do this. SOAP is one method used to serialize an object into XML. Barracuda supports a lighter and more common XML protocol called XML-RPC.

Many scripting languages natively support XML-RPC. Python, for example, can utilize any object in the embedded device exposed as an XML-RPC object from a script running on a host computer. Another common regression test environment is Expect, based on TCL, which also has support for XML-RPC.

Using a browser to control an embedded device

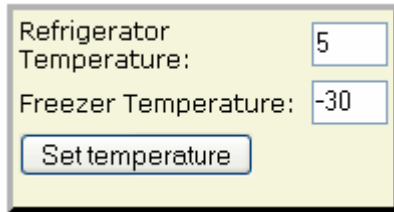
Most people associate a Web Server as a way of serving HTML content to a browser. An advanced Web Server such as Barracuda can do much more than serving HTML files. We briefly discussed above how one can use Barracuda as a general purpose HTTP stack and how one can do regression tests by exporting internal objects in a device to script programs running on a host computer.

A naive Web Server treats resources as files and either has no support for dynamically generated content or forces the Web-Designer to use certain predefined directories for generating dynamic content. For example, a server supporting CGI forces the web-designer to use the /cgi/ directory for dynamic content generation and data exchange.

Barracuda supports Executing Units, thus a resource is a "live resource" i.e. it is a mini program one can plug into the server. In Barracuda, a resource is typically derived from the HttpPage type. A more complex design can use the HttpDir type to design a collection of virtual resources. A Barracuda resource can dynamically generate anything from binary data such as streaming audio to plain text. A resource is typically generating HTML when the client is a browser.

A typical scenario when using a browser to control a device is for the browser to initially request a certain resource in the server. The resource might respond by sending back an HTML page containing information about that particular resource when receiving an HTTP GET command. For example, a resource at "refrigerator/temperature/" sends back information about the current temperature in the refrigerator and freezer. The HTML sent to the browser can for example contain an HTML form, which the user can use for setting a new temperature in the device.

HTML form example



Refrigerator Temperature:

Freezer Temperature:

The HTML form source code

```
<form method="post">
<table>
<tr>
<td>Refrigerator Temperature:</td>
<td><input type="text" name="fridgeTemp" value="5"/></td>
</tr>
<tr>
<td>Freezer Temperature:</td>
<td><input type="text" name="freezerTemp" value="-30"/></td>
</tr>
<tr>
<td colspan="2">
<input type="Submit" value="Set temperature"/>
</td>
</tr>
</table>
</form>
```

If the user wants to change the temperature, the user can use the form displayed in the browser to change the values and then press the submit button. When the user presses the submit button, the browser collects the data in the form and sends it as URL-encoded data to the server. The Barracuda server decodes the URL-encoded data and starts the resource specified in the POST command. In Barracuda, this will typically be the same resource as the resource that produced the initial HTML page. A naive web-server forces the web-developer to POST the data to a predefined fixed URL, for example, a CGI Web Server forces the web-developer to POST data to the "cgi/" directory.

GET tells the resource to dynamically create HTML and return it to the browser.

POST tells the resource that the user wants to set new data.

Introduction to CSP

A Barracuda resource is typically derived from the `HttpPage` type. When the user requests a resource, the Barracuda Web Server locates the resource and delegates the request to the resource. A resource gets two objects from the server, a request object and a response object. The request object contains information about the client and any data sent from the client. The response object is used when dynamically creating the response data, such as an HTML response.

The following shows a code fragment from a resource object serving a HTTP GET request.

```
response->write("<html>");
response->write(" <body>");
response->write(" <p>Greetings</p>");
response->write(" </body>");
response->write("</html>");
```

As you can see from the above fragment, the resource object creates a simple HTML page containing a greeting message.

Most Web-Developers want to separate content from logic. The above HttpPage resource code example shows a tight integration of code and HTML content. Maintaining such a resource is difficult and time consuming. The HttpPage resource makes it possible to do advanced HTTP communication. A way of separating logic and content, while giving developers the flexibility of the services provided by a HttpPage resource is needed.

The Barracuda platform solves this problem by providing a special TAG compiler that takes an HTML file and translates the file to an HttpPage. The TAG language is called CSP. CSP, short for C Server Pages, is a modern reimplementaion of the CGI standard. CSP extends HTML with new tags that are only visible on the server side.

The following shows a very simple CSP page.

```
<html>
  <body>
    <p>Greetings</p>
  </body>
</html>
```

The above example shows that a CSP page looks like a regular HTML file. This page contains no special CSP tags and behaves like static HTML file.

CSP also makes it easy to create dynamic pages. A dynamic page is a resource that creates content either from user input or from some information in the device.

Let us now change the above simple CSP page to the following:

```
<html>
  <body>
    <p><%= "Greetings" %></p>
  </body>
</html>
```

CSP tags start with <% and end with %>. The <%= tag tells the CSP runtime to take the following string and send it to the client that requested the resource -- i.e. to the browser.

We have now added a CSP tag to the page, but the result produced is identical to our first CSP page. The string "Greetings" is constant and will never change.

Continuing with our CSP example page:

```
<html>
  <body>
    <p>Greetings user of <%=request->getHeaderValue("User-Agent")%></P>
  </body>
</html>
```

Now we added some C++ code directly into the HTML page. The method getHeaderValue in the request object returns the value for the "User-Agent" HTTP request header value. The Barracuda server decodes all the header values and keeps them in an internal table, thus getHeaderValue("User-Agent") returns the identity of the client doing the request.

Barracuda is a C library written in ANSI C compatible code, but provides both a C++ and a C interface. The above C++ code can be written as the following C code

```
<%=HttpRequest_getHeaderValue(request, "User-Agent")%>
```

The Barracuda framework provides a rich API to the developer. A CSP resource can use all of the C or C++ methods provided by this API. A CSP resource has two implicit objects available when executing: the request and response object. The request object provides information sent from the client to the server, while the response object is used when sending data back to the client. The response object is implicitly used in the above code by the `<%=` tag.

Creating dynamic user interfaces using CSP.

The first phase of developing a CSP application is creating the HTML interface, and is usually handled by a professional web interface developer. The second phase involves a C programmer or web developer with a small amount of C experience to add the CSP tags to the HTML code.

The final step is testing the new interface in the Barracuda Embedded Web Server. This can be an iterative process, and will show places where adjustments should be made to the HTML or other interface components. HTML editors such as Dreamweaver will not corrupt or remove the CSP tags with the C/C++ code.

The following examples are simple so that they can easily be understood. In most web applications, the HTML would be much more extensive, and would also likely involve CSS.

First, we'll add an interface to display the temperature reported by a new controller unit on our refrigerator.

```
<html>
  <body>
    <h1>My refrigerator</h1>
    <table>
      <tr>
        <td>Refrigerator Temperature:</td>
        <td><%= "%d" rand() %10 %></td>
      </tr>
      <tr>
        <td>Freezer Temperature:</td>
        <td><%= "%d" - rand() %35 %></td>
      </tr>
    </table>
  </body>
</html>
```

The `<%=` tag by default assumes that the data is a string. All other types must be explicitly declared. The `<%=` tag accepts the same format flags as `printf`, so `<%= "%d"` means "print an integer value". The above code uses the ANSI C function `rand` to generate a random temperature value. A real implementation would call a function that returns the actual temperature. The modulus operator is used such that we get a refrigerator temperature

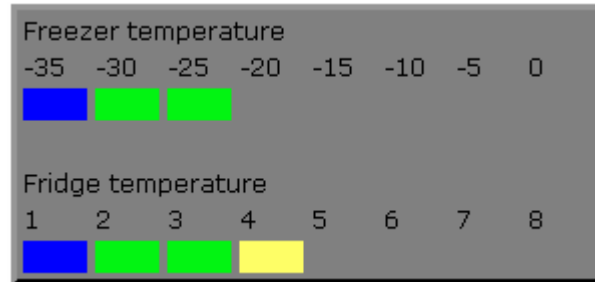
between 0 and 10 degrees Celsius, and a freezer temperature between 0 and -35 degrees Celsius.

Most of the content in this example page is static. This is typical when working with CSP. The HTML framework is always the same, but the temperature data is dynamically presented to the user.

An improvement to this, which just prints out the temperature of the freezer and refrigerator in an HTML table, would be to present the temperature graphically as a bar chart.

There are a number of ways to create a bar chart using HTML. This example will dynamically create a HTML table containing small images.

The figure to the right shows a bar chart representation of a freezer temperature of -25 degrees Celsius and a refrigerator temperature of 4 degrees Celsius:



The colors blue, green, yellow, orange and red are used. Also, a transparent image is used when hiding the temperature element. The above freezer bar contains 5 transparent images for the temperature -20, -15, -5 and 0.

To keep the example as simple as possible, the refrigerator temperature is not included in our CSP example. The table in the CSP code below contains three table row elements (<tr>). The first row contains a table data element, which spans across 8 elements, the second row contains the temperatures, and the last table row contains the images. The getFreezerImg function returns the string for the current image to insert into the table element.

Refrigerator CSP code:

```
<html>
  <body>
    <h1>My refrigerator</h1>
    <table>
      <tr>
        <td colspan="8">Freezer temperature</td>
      </tr>
      <tr>
        <% /* print out the temperatures -35, -30 etc in the first table row */
          for(i = -35 ; i <= 0; i+=5)
            response->printf("<td>%d</td>", i);
        %>
      </tr>
      <tr>
        <% /* Print out the images in the second table row */
          for(i = -35 ; i <= 0; i+=5)
            response->printf("<td><img src=\"%s.gif\"/></td>",
              getFreezerImg(freezerTemp, i));
        %>
      </tr>
    </table>
  </body>
</html>
<%p
  int i;
  int freezerTemp = - rand()%35;
%>
<%g
#include <stdlib.h>
const char* getFreezerImg(int freezerTemp, int i)
{
  if(freezerTemp < i)
    return "transparent";
  switch(i)
  {
    case -35: return "blue";

    case -30:
    case -25: return "green";

    case -20:
    case -15: return "yellow";

    case -10:
    case -5: return "orange";

    case 0: return "red";

  }
  assert(0); /* Should not get here */
  return "";
}
%>
```

The above CSP example demonstrated several new tags.

The `<%g` tag means global. Anything declared here goes outside of the generated `HttpPage`. The `getFreezerImg` function is inserted into the global area by the CSP compiler.

The `<%p` is the prolog tag. Anything declared here is inserted at the beginning of the `HttpPage` service function. The `<%p` is a special form of a code fragment tag.

The `<%` is a code fragment tag. Anything in a code fragment tag is executed when the `HttpPage` service function runs. For example, the following code fragment tag from the example above prints out table data elements containing the images.

```
<%
  for(i = -35 ; i <= 0; i+=5)
    response->printf("<td><img src=\"%s.gif\"/></td>",
                    getFreezerImg(freezerTemp, i));
%>
```

The following will be printed out by the above code fragment if the freezer temperature is -25 degrees:

```
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
```

We use the `printf` function in the response object to print out the table data elements. Another possibility is shown below.

```
<% //Begin loop
  for(i = -35 ; i <= 0; i+=5) {
%>
  <td></td>
%>
  } //End loop
%>
```

This produces the same output as the original code fragment.

Sending data from a browser to the server

A client application does not necessarily have to be a web-browser. When a browser sends data to a server, the content type of the data normally is "application/x-www-form-urlencoded". Browsers don't normally send other content types without executing JavaScript code.

URL-encoded data is sent as key/value pairs as "key1=value1&key2=value2&key3=value3".

Some characters must also be "quoted" to prevent them from being handled as control characters. This "quoting" is transparent to the CSP designer when using the Barracuda framework.

Sending data from a browser to the server normally works like this:

- The browser sends a GET request to a resource in the web-server.
- The web-server delegates the request to the resource.
- The resource responds by sending an HTML page containing one or a number of HTML forms to the client.
- The user fills in the form and submits the form to the web-server.
- The browser collects the form data and sends the information as URL-encoded data to the resource.
- The web-server delegates the request to the resource, and this time the resource detects that this is a POST command. The resource fetches the pre-parsed URL-encoded data from the request object by using methods in the request object.

The following code will make the refrigerator resource also able to change the temperature settings.

```
<html>
  <body>
    <h1>My refrigerator</h1>
    <form method="post">
      <table>
        <tr>
          <td>Refrigerator Temperature:</td>
          <td><input type="text" name="fridgeTemp" value="<%= "%d" fridgeTemp%>" /></td>
        </tr>
        <tr>
          <td>Freezer Temperature:</td>
          <td><input type="text" name="freezerTemp" value="<%= "%d" freezerTemp%>" /></td>
        </tr>
        <tr>
          <td colspan="2"><input type="Submit" value="Set temperature" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
<%!
  // Declare two variables in the HttpPage object
  int fridgeTemp;
  int freezerTemp;
%>
<%!!
  // Initialize the two variables at system startup.
  fridgeTemp = 5; // Default refrigerator temp
  freezerTemp = -30; // Default freezer temp
%>
<%p
  if(request->getMethodType() == HttpMethod_Post)
  { // The following code is executed if the user press the submit button
    int newFridgeTemp = (int)U32_atoi(request->getParameter("fridgeTemp"));
    int newFreezerTemp = (int)U32_atoi(request->getParameter("freezerTemp"));
    // Are the new values within the tolerated limits.
    if(newFridgeTemp < 1 || newFridgeTemp > 7 ||
       newFreezerTemp < -35 || newFreezerTemp > -18)
    { /* No, the user obviously does not know enough about food
      safety. Send user to a page that can educate the user
      on food safety.
      */
      response->sendRedirect( /* Send HTTP code 302 to browser */
        "http://lancaster.unl.edu/food/ciqxx.htm");
      return; // Abort
    }
    //Set the new temperature
    fridgeTemp = newFridgeTemp;
    freezerTemp = newFreezerTemp;
  }
%>
```

If you look at the HTML section, you will see that we have now added a HTML form element to the page. The original table data elements printing out the temperature are now changed to a table data element containing a HTML form input type.

```
<td><input type="text" name="fridgeTemp" value="<%= "%d" fridgeTemp%>" /></td>
```

The default value for fridgeTemp is 5 degrees, so what is initially sent to the browser is:

```
<td><input type="text" name="fridgeTemp" value="5"/></td>
```

When the user clicks the submit button, the browser sends the data to the web-server. The initial HTTP POST command in the HTTP section above shows what this form data looks like during transmission. The URL-encoded data "fridgeTemp=5&freezerTemp=-30" corresponds to the two field names in the CSP example.

The CSP code extracts the fridgeTemp and freezerTemp in the prolog section, i.e., the tag starting with <%p.

```
request->getParameter("fridgeTemp")  
request->getParameter("freezerTemp")
```

The code example above uses the keys "fridgeTemp" and "freezerTemp" in order to fetch the value part of the data.

The server will perform a sanity check on the data. This is an important step and is typically where web applications fail. If the sanity check fails, the user is forwarded to a page explaining food safety. This is something we added for fun. A web application would normally respond with an error page.

Another implicit sanity check is performed by the U32_atoi function. The U32 is an unsigned 32 bit type in the Barracuda web-server. The U32_atoi function works like the standard atoi function, but it also can tolerate a NULL string without crashing. Even though the U32 function returns an unsigned integer, the function can decode and negate negative numbers.

```
//The following returns NULL if "fridgeTemp" is not in the URL-encoded data.  
request->getParameter("fridgeTemp")
```

Hackers are constantly trying to penetrate and crash web servers by using a number of methods, and they aren't connecting to web servers with browsers for these attacks. They won't always be sending proper form data, so the server must check that all required fields are present.

"Hacking" a Web Server using telnet

As a simple example, open a command window (DOS window). In windows 2000 and XP a command window can be started by pressing the start button and thereafter selecting run. Type cmd in the pop-up window and press return. Copy the text below by marking it and press control-C.

```
telnet embeddedwebserver.net 80
POST /refrigerator/temperature/ Http/1.0
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

fridgeTemp=2&freezerTemp=-25
```

Now paste the text into the DOS window. Text can be pasted into a DOS window by pressing the top left icon in the DOS window, the small "C:\\" icon. Scroll down to edit and select paste. The resource should respond with a message telling you that you successfully set the refrigerator temperature to 2 degrees and the freezer temperature to -25 degrees Celsius. You can also [download](#) two DOS batch files, which automatically run the telnet commands.

Now do the same thing, but without the URL-encoded data.

```
telnet embeddedwebserver.net 80
POST /refrigerator/temperature/ Http/1.0
User-Agent: I-typed-this-connecting-a-telnet-client-to-the-server
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
```

Copy the above text and paste it into the same DOS window, and observe the different response message.

The path in the HTTP POST example requests a directory since the path ends with "/". The web server automatically adds "index.html" to the end of all directory requests. If you change the path to "/refrigerator/temperature/index.html" in the above examples, the results will be the same.

Barracuda supports request delegation, which is typically used in Model View Controller design. We use request delegation to construct a simple filter in index.html. The actual temperature resource is in "ex4.html". The index.html CSP resource is just a filter that validates a POST request and responds by sending a different message to the client if the Content-Length is 0.

The index.html resource delegates (forwards) the request to the ex4.html resource if the Content-Length is valid. This can be seen by changing the path in the first (the valid) telnet example above from "/refrigerator/temperature/" to "/refrigerator/temperature/ex4.html". You should get the same response message from the server.

Rich Client Interface

There are a lot of buzzwords on the internet these days about Rich Client applications, like "Rich Client Interface", "DHTML client", "Web Services", "Ajax", etc. A Rich Client application uses the browser as an environment to create a fully interactive graphical user interface rich and interactive user interface, and lies between a standard lightweight HTML form and a full native client-side OS-based application. A Rich Client can also be a standard Windows application that communicates with the server using HTTP, although such an application must be installed on the host computer before interacting with the server.

Modern browsers natively support a programming language called JavaScript. They also have a standard API, the Document Object Model, that JavaScript code can use for creating the user interface. This API allows the JavaScript code to create interactive GUI applications that resemble native applications. This basically means that you can design Dynamic Web-Applications using DHTML without the need of refreshing or reloading pages when you send data to the server.

A DHTML client communicates with the server using an object called XMLHttpRequest which, despite the name, has nothing to do with XML. The XMLHttpRequest allows JavaScript code to send any type of HTTP request to the server.

Below we have created a Rich Client Interface for the simple fridge controller we discussed above.

The popular "Web Services" is basically a method of serializing and de-serializing objects over HTTP. Barracuda supports XML-RPC Web Services, even a standard Web Services client can be used when communicating with Barracuda. Typically, Web Services won't be used in a DHTML client when URL encoded data is sufficient. The original resource already accepts URL encoded data which is much easier to manage for most applications.

The resource `"/refrigerator/temperature/ex4.html"` is designed to accept URL encoded data. Our original resource wasn't configured to handle Web Services requests, so a new temperature object should be constructed specifically for Web-Services.

The response from the XMLHttpRequest object is sent asynchronously to a JavaScript callback function. The original service function responds by sending a HTML page to the client. A DHTML client could potentially parse the HTML to get the new temperature settings, but this requires a lot of work. A better solution is to tell the server resource that the client accepts JavaScript code as a response message by setting the correct "Accept" header in the request.

```
xmlhttp.setRequestHeader('Accept', 'JavaScript');
```

The server resource can simply check for this header.

```
if( strcmp("JavaScript", request->getHeaderValue("Accept")) == 0)
    Send JavaScript object
else
    Send HTML page
```

If the client sent the header, then a JavaScript object is sent as response message.

Ajax is an example of this, and can be found by searching on the internet. You can also take a look at Google Suggest, which is using an XMLHttpRequest object in the background as you type your search string.

The EventHandler

The above examples show how to design basic user interfaces using CSP technology and Rich Client Interfaces that can communicate with the server without refreshing the page.

Both the server generated CSP interface and the Rich Client interface must explicitly ask the server for a new temperature reading. A CSP-generated page could be sent to the browser that automatically refreshes the page after a number of seconds, and the Rich Client interface can have a JavaScript timer periodically call the XMLHttpRequest.

Polling the server for data adds overhead to the network traffic, and can even make the client interface behave strangely. It would be better if the server could inform the client asynchronously of changes to the temperatures without waiting for the client to ask again.

The Barracuda EventHandler plugin makes this possible as described in our EventHandler White Paper. This White Paper shows Rich Client Interface designs that aren't possible to implement using traditional RPC mechanisms.

The Barracuda EventHandler plugin gives Web developers the ability to add server-based notification functions and two-way, real-time data exchange to browser interfaces. This improves the functionality of a wide range of Web applications, and makes them useful for functions like alarm handling and live monitoring of devices.

Conclusion

We have covered a lot of ground in this paper, but we have barely scratched the surface of the endless possibilities with the Barracuda platform. The following White Papers are for users interested in a deeper understanding of the Barracuda platform:

- [Asynch trace tool](#) shows how the EventHandler can be used to construct an embedded device trace tool by having a printf like function output its data asynchronously to a browser window.
- [Device Control](#) demonstrates how to design a resource without using the CSP compiler.
- [Hangman](#) implements the classic Hangman Game using CSP.
- [Web security](#). A device that is constantly connected to the internet needs to be protected from unauthorized use. This paper gives an introduction to the security mechanisms in Barracuda.

References

- [Http Made really Easy](#)
- [Introduction to XMLHttpRequest](#)
- [Asynchronous JavaScript + XML = Ajax](#)
- [Google Suggest = example Ajax application](#)
- [Representational State Transfer \(REST\)](#)

Device control with Barracuda

Barracuda, which is an object oriented C library, provides a C code API and a C++ code API. We have two versions of this white paper, one for [C programmers](#) and one for C++ programmers. This article assumes that you are a C++ programmer with some HTML knowledge. It is suggested that you read the [introduction to Barracuda](#) prior to reading this article.

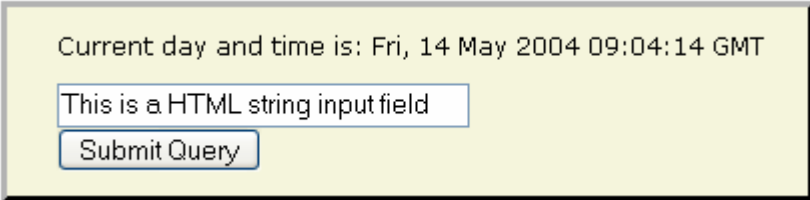
The Barracuda Web-Server comes with a number of sophisticated tools that can automate much of the work in creating web-pages for device control. We have in this article purposely not used any of these tools in order to show you the foundation of the web-server and to keep the article as short as possible.

Use Barracuda to remotely set and get the system time

This article will demonstrate how one can use Barracuda to remotely control a device. The Barracuda Embedded Web-Server will be used to remotely set and get the system time in an embedded device powered by the SMX real-time operating system. The Barracuda Web-Server, which is a C library, is linked with the SMX real-time operating system and the set/get system time code. **Barracuda runs on many operating systems, but we have in this paper used a few primitives from the SMX real-time operating system for setting and getting time etc.**

The objective of this article is to make a simple web-interface, which we can use to obtain the current time in an embedded device or to change the time, if the time is incorrect.

The web-interface should have the following layout:



Current day and time is: Fri, 14 May 2004 09:04:14 GMT

This is a HTML string input field

Submit Query

When the user makes the browser send a HTTP GET request to the web-server, the C++ code we are going to design should get the current time from SMX, format the time into a human readable string and send the string to the browser. A HTML form will also be created and sent to the browser. The user can use this form for setting a new time in the embedded device.

The [HTML form](#) contains two input fields:

- A string input field, which is where the day and time is entered.
- A submit button, which sends the form as a POST request to the server.

The Hyper Text Transfer Protocol (HTTP) is the network protocol of the Web. It is a protocol designed to transmit data from the client to the server and vice versa. The GET command sent from a browser (the client) to the server is the most common HTTP method; it says "give me this resource". This is the initial command sent to the server for getting the above web-interface. The set/get time logic we will design will get this request and dynamically create the above web-interface. Every time the user presses the refresh button, the browser sends a new GET request to the server and a new HTML page is dynamically created with the correct time formatted as a human readable string. You will therefore see the time change every time you

press the refresh button. A POST command is used to send data to the server. The browser will automatically send a POST command when the user presses the "submit" button. The browser extracts the user data entered in the HTML form and sends this data embedded inside the POST command to the server. The set/get time logic will receive this data and set the new date and time based on the data received from the browser.

Assembling the web-server

As aforementioned, Barracuda is a C library, which you link with your application and you therefore have to write some support code in order to make a web-server. Barracuda is designed to run in its own SMX task, where it uses the TCP/IP interface and waits for client requests. When a client sends a command to the server, the server first parses the incoming data and from this data, the server uses the extracted HTTP path to locate the object (the page) that matches the path. When the object (the page) is located, the web-server sends an event to the object and it is this object's responsibility to handle the request from the client and to present the information requested by the client.

The Barracuda web-server uses a virtual file system. The path extracted from the client request is used when searching the virtual file system for the object (the page) requested by the user. If the object is found the web-server sends an event to this object, by calling a registered callback function in this object.

The web-server we design in this article will only have one resource. The set/get time resource is installed as "index.html" and the web-server knows that if a user requests a path to a directory and that directory contains an index.html, the web-server will forward the request to that page. The resource will be installed in the root directory thus by typing the IP address to the device in a browser, the resource will be activated. For example, assuming that the device has IP address 192.168.1.1, the URL <http://192.168.1.1:8080> will activate our set/get time resource. The 8080 is the port number to the web-server. You have to specify the port number in the URL if the web-server is listening on a port different than the default HTTP port 80.

The Barracuda SMX task

Barracuda, which is run in a SMX task, must first be initialized. After the initialization, the task enters an infinite loop which waits for incoming HTTP requests.

The Barracuda SMX task does the following:

1. Performs host platform initialization if run on Windows.
2. Creates the necessary Barracuda Web-Server objects and bind the objects together.
3. Checks if the initialization of the HTTP server listen object worked.
4. Creates the virtual file system and insert the set/get time resource into the virtual file system.
5. Runs the web-server.

```
116 void
117 barracudaMain()
118 {
```

1

```
119 #ifdef WIN32
120     WORD wVersionRequested;
121     WSADATA wsaData;
122     wVersionRequested = MAKEWORD(1, 1);
123     if(WSAStartup( wVersionRequested, &wsaData ))
124         perror("WSAStartup");
125 #endif
126
127     int port=8080;
128
```

2

```
129     //Create the socket dispatcher object.
130     HttpDispatcher dispatcher;
131     //Create the Web-Server
132     HttpServer server(&dispatcher);
133     //Create a server socket listener that listen on all interfaces.
134     HttpServCon serverCon(&server, port);
135
```

3

```
136     if( !serverCon.isValid() )
137     { //A primitive error handling
138         assert(0);
139         return;
140     }
141
```

4

```
142     HttpDir rootDir; //Root dir takes no parameters i.e. no name
143     SetGetTimePage setGetTimePage;
144     rootDir.insertPage(&setGetTimePage);
145     server.insertRootDir(&rootDir);
146
```

5

```
147 #ifdef WIN32
148     for(;;)
149     {
150         struct timeval t;
151         t.tv_sec=0;
152         t.tv_usec=0;
153         dispatcher.run(&t);
154         Thread::sleep(50); /* 50 msec. */
155     }
156 #else
157     dispatcher.run();//Never returns
158 #endif
159 }
```

- Line 117 is the start of the Barracuda SMX web-server task. You must create and start one instance of this task.
- Line 119 to 125 is used when you are running the web-server using the SMX windows simulator. Barracuda is using Windows sockets when run on windows, and the windows socket library must always be initialized as shown above.
- Line 127, the web-server port number is set to 8080 and not to the default HTTP port 80. This is useful when run on a host computer that might have another web-server running on port 80. You can use any port number with Barracuda.
- Line 130 to 134 is where we assemble our minimal web-server. A Barracuda web-server can be assembled in many different ways.
- Line 136 is where we check if the HttpServCon constructor was successful in opening a TCP/IP listen object on port 8080. This might fail if another service is already using this port number.
- Line 142 to 145 is where we design the minimal virtual file system and insert the virtual file system into the web-server object. The HttpDir type is the base type for a directory node in the virtual file system. We can use the base type directly, but it is sometimes convenient to overload this base class and perform custom operations such as filtering and logging of client requests.
- Line 143 is where we instantiate one instance of our set/get time resource. The remaining part of this article will discuss how this class is designed.
- Line 157 is where we enter the web-server socket dispatcher loop when using SMXNET. This function, which never returns, waits for incoming socket requests from clients and dispatches the events to the web-server. When a user types in the URL http://[IP address of device]:8080, the TCP/IP stack wakes up the barracuda task, which starts processing the request. The path is extracted from the GET request, which will be '/'. The web-server uses this path to locate our set/get time resource and calls the resource's callback function.
- Line 148 to 154 is how we run the socket dispatcher when using windows sockets. The SMX windows simulator is not compatible with a blocking call into the windows socket API and therefore the web-server must run in "poll mode".
- Line 154 is where we make other SMX tasks run. The Thread class is a simple cross-platform thread library that comes with Barracuda.



As you can see from the above code, Barracuda can also run on the SMX windows simulator. The Barracuda C library and advanced host tools facilitate rapid development using a Windows machine. You develop, run and test the code on Windows before you even deploy the code into an embedded device -- obviously, a huge time saver.

Creating the set/get time page class

A resource is an instance of the `HttpPage` class. The `HttpPage` class is in C++ terminology -- an abstract base class. This means that you cannot create an instance of this class without first making a derived class. The set/get time class inherits from the `HttpPage` object and implements the resource's service function. The service function is the callback function the web-server calls when the web-server delegates the request.

The set/get time page class:

```
9 class SetGetTimePage : public HttpPage
10 {
11     public:
12         SetGetTimePage();
13     private:
14         static void service(HttpPage* o,
15                             HttpRequest* request,
16                             HttpResponse* response);
17         void doPost(HttpRequest* request, HttpResponse* response);
18         void doGet(HttpResponse* response);
19         void setRTC(struct tm* tm);
20 };
```

- Line 14 is the declaration of the service callback function. This is a C++ static function and the first argument to the function is therefore a pointer to the object, the "this" pointer in C++ terminology. You might wonder why we use a callback function and not a C++ virtual function. The reason for this is that Barracuda is a C library and a virtual function pointer is not compatible with C code.
- Line 17 to 19 declares some functions used internally by our `SetGetTimePage`. These functions will be explained later.

SetGetTimePage constructor:

```
22 SetGetTimePage::SetGetTimePage() : HttpPage(service, "index.html")
23 {
24 }
```

The `SetGetTimePage` constructor calls the `HttpPage` constructor. The first argument is a pointer to the service function declared on line 14 and the second argument is the name of the page. The page name and the function pointer to the service function are used by the virtual file system when locating and calling the page service function.

The service function:

The web-resource service callback function is called by the web-server (via the virtual file system) when the web-server delegates either a GET or a POST request to an instance of the `SetGetTimePage` class. Recall that we registered the service callback function when we called the `HttpPage` Constructor (line 22).

```

26 void
27 SetGetTimePage::service(HttpPage* o,
28                         HttpRequest* request,
29                         HttpResponse* response)
30 {
31
32     response->write("<html><head>"
33                  "<title>Set or get SMX time</title>"
34                  "</head><body>");
35
36     if(request->getMethodType() == HttpRequest_Post)
37         ((SetGetTimePage*)o)->doPost(request,response);
38     else
39         ((SetGetTimePage*)o)->doGet(response);
40
41     response->write("</body></html>");
42 }

```

The first argument is the "this" pointer, but we have to typecast the "this" object to the derived class. Remember, the service function is a static C++ callback function, which is compatible with a C callback function.

The second argument, the request object, contains all the information you can possibly get from the client. This information is pre-processed and presented as sub-objects within the request object. For example, a client sending data to the server using a POST command is pre-parsed by the web-server and stored as objects in the request object.

The third argument, the response object, is used when sending the response message back to the client. For example, line 32 uses the "write" member function to send the initial part of the HTML text back to the client, and line 41 sends the closing HTML page tags to the client.

Line 36 detects if this is a POST request or a GET request. Recall that GET is used to fetch a new page and POST is used to send data from the client to the server. As you can see, we have made one function to handle the GET request and one function to handle the POST request. This is not necessary, but it sometimes gives a cleaner design.

Dynamically create the web-interface

The doGet function is called when the user types in the device URL in the browser or press the refresh button.

The doGet function does the following:

1. Gets the current time from SMX.
2. Formats the time into a human readable string.
3. Sends the string to the client.
4. Sends the HTML form to the client.

The GET request function:

```
45 void
46 SetGetTimePage::doGet(HttpResponse* response)
47 {
48     static const char* wd[7] = {
49         "Sun","Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
50     };
51     static const char* months[12] = {
52         "Jan","Feb", "Mar", "Apr", "May", "Jun",
53         "Jul","Aug", "Sep", "Oct", "Nov", "Dec"
54     };
55     struct tm tm;
1
56     dword time = get_stime();
57     httpTime2tm(&tm, time);
58     response->write("<p>Current day and time is: ");
2 and 3
59     response->printf("%s, %02d %s %d %02d:%02d:%02d GMT",
60                    wd[tm.tm_wday],
61                    tm.tm_mday,
62                    months[tm.tm_mon],
63                    tm.tm_year+1900,
64                    tm.tm_hour,
65                    tm.tm_min,
66                    tm.tm_sec);
67     response->write("</p>");
4
68     response->write(
69         "<p>"
70         "<form method='post'>"
71         "<input type='text' name='time' size='30' />"
72         "<br />"
73         "<input type='submit' />"
74         "</form>"
75         "</p>");
76
77 }
```

- Line 56 uses the SMX time function to get the current time. This is in the number of seconds since January 1, 1970. The web-server implements and uses a number of time conversion functions. These functions are also available for you to use. Line 57 converts the time in seconds to a tm structure. This structure is documented in time.h.
- Line 59 formats the tm structure into a human readable string and sends the string to the client. The printf member function works just like the regular printf, except for that the output is sent to the client. The printf function is specially designed for embedded devices and uses very little stack.
- Line 70 to 75 emits the HTML form to the client. This form will be used by the user when changing the time in the server.

Setting a new time in the device

The doPost function is called when the user press the "submit" button.

The doGet function does the following:

1. Extracts the form data (the date and time string sent from the client to the server).
2. Converts the date and time string to number of seconds since January 1, 1970.
3. Converts number of seconds to a tm structure.
4. Sets the new time in the device.
5. Sends a command to the client that makes the browser do an automatic GET request.

The POST request function:

```
80 void
81 SetGetTimePage::doPost(HttpRequest* request, HttpResponse* response)
82 {
83     HttpTime time;
84     struct tm tm;
1
85     const char* timeStr = request->getParameter("time");
2
86     if(timeStr == NULL || (time=httpParseDate(timeStr)) == 0)
87     {
88         response->printf(
89             "<p>"
90             "Cannot parse date and time string. "
91             "Date string must be in RFC1123 or RFC850 format.<br/>"
92             "<a href='%s'>Try again</a>"
93             "</p>",
94             getName());
95     }
96     else
97     {
3
98         httpTime2tm(&tm, time);
4
99         setRTC(&tm);
5
100         response->sendRedirect(response->encodeRedirectURL(getName()));
101     }
102 }
```

- Line 85 extracts the time field from the form data sent from the client. The string "time" matches the form input field name on [line 71](#) (name='time'). Barracuda makes it extremely easy to work with complex form data sent from a client to the server. This is just one of many functions you can use for retrieving data sent from the client to the server.
- Line 86 checks if the form data should for any reason be empty and if the Barracuda httpParseDate function could successfully convert the string to time in seconds since

January 1970. The `httpParseDate` function is a complex parser function implemented and used internally by the web-server, but most of these functions are also available to user code.

- Line 88 to 94 sends a response message to the client if the date and time could not be parsed.
- Line 98 uses a Barracuda function and converts the time in seconds to a `tm` structure.
- Line 99 calls the function that sets the system time.
- Line 100 tells the client to redirect the request to the given URL. The `sendRedirect` function can only take an absolute URL, such as `http://mydomain/myPath/myPage.html`. The `encodeRedirectURL` converts the relative path to an absolute URL and the `getName` function returns the name of the page. The `getName` is a member function in `HttpPage` and `SetGetTimePage` inherits from `HttpPage`.



You may have noticed that the `sendRedirect` function, which only sends a HTTP header and no body text, is called after we had already started sending data to the client. For example, we sent the beginning of the HTML body on [line 32](#). This works because the web-server does not immediately send data to the client. The data is buffered internally in the response object and flushed to the client when the buffer is full. Calling the `sendRedirect` member function discards any data in this buffer and immediately sends the redirect HTML header to the client. The `sendRedirect` member function also disables the response write functions. For example, the write on [line 41](#) will have no effect after calling the `sendRedirect` member function. At this point, you might not understand the enormous amount of aid provided by the Barracuda web-server, but you will come to appreciate these features when you design complex user interfaces for device control. In the end, you will not understand how you could ever survive without Barracuda.

You might wonder why we send a HTTP redirect response header to the client instead of sending a response body in the POST request. There are two reasons for this:

1. We already have the logic for sending the web-interface HTML code to the client when the client sends a GET request.
2. A browser might stay in "POST mode" unless the browser is specifically doing a GET after the HTTP POST response. This means that pressing the "refresh" button in the browser would have sent another POST request to the server. The redirect response makes the browser do a GET after the POST request. This is transparent to the user.

You can get the full source code for the set/get time code [here](#).

Conclusion

We have shown you how to implement a resource by sub-classing the `HttpPage` class. A `HttpPage` resource can do all kinds of sophisticated device control. The Barracuda framework, which models the framework in [J2EE](#) makes it extremely easy to design web-interfaces for controlling your device.

A `HttpPage` is similar in functionality to a [Java-servlet](#), except for that a `HttpPage` is written in C or C++. This gives you an enormous advantage since a good Java-servlet book, with its interface resembling the Barracuda interface, can help you leverage the true power of the Barracuda framework.

We also mentioned that one could overload the functionality in an `HttpDir` (the virtual file system directory node). Barracuda comes with many classes that sub-class the `HttpDir` class. These plugins make it possible for the Barracuda web-server to read HTML pages from a regular file system, view a ZIP file as a read only file system, implement authentication and authorization, etc..

Writing a web-page class such as the SetGetTimePage in this article is normally not necessary as the Barracuda web-server comes with a number of development tools that can automate this task. One of these tools is a compiler that can take an HTML file and automatically convert the HTML file to either a C or C++ web-page class. This HTML file can use special server side tags, which we call CSP tags. The CSP tag language, which means C Server Pages, is similar in functionality to ASP. The ASP tag language invented by Microsoft makes it possible to insert Visual Basic within special server side HTML tags in an HTML file. Our CSP tag language makes it possible to insert C or C++ within the same server side tags. [Here is an example of the above code designed in CSP.](#)

The host tools that comes with Barracuda can help you convert an entire web-site with sub-directories, gif images, etc. into a virtual file system, thus you do not have to manually create the virtual directory structures as we did in this example.

Please see our [CSP tag language white paper](#) for more information on how to design and do rapid development of advanced device control logic using the CSP compiler.

The CSP Hangman game

This article will show you how to use the server side tag language integrated into the Barracuda Embedded Web-Server. It is suggested that you read the [introduction to Barracuda](#) prior to reading this article.

Server side scripting, a method used in web-servers to dynamically create HTML, is typically used in embedded devices as a way to dynamically create a user interface. The dynamic HTML rendered in the server may, for example, show the dynamic changes in a power meter in a device.

Barracuda supports a very advanced type of server side scripting, which we call CSP. CSP is similar to Active Server Pages, ASP, which was invented by Microsoft. With CSP, you can embed C or C++ directly into an HTML template page created by an HTML designer.

This article assumes that you are a C or C++ programmer with some HTML knowledge and some experience in server side scripting. An introduction to server side scripting is outside the scope of this article, but since Barracuda is similar to ASP, one can refer to an ASP book. The ASP tag language is similar to CSP and the ASP API is similar to the API provided by Barracuda, except for that Barracuda is using C or C++ for server side scripting.

A good introduction to ASP can be found online: <http://www.w3schools.com/asp/>

We will demonstrate how one can design the classic hangman game using CSP. The objective of the game is to guess the word before the man is hung.

The hangman game is integrated into our [Barracuda standalone windows demo \(11MB\)](#). The demo is a self-extracting archive. It automatically starts when the files are extracted. Audio commentary begins immediately, so turn on your speaker before running.



[Start Example](#)

The computer randomly selects a word from its dictionary. The correct number of blank spaces will appear above the gallows. The gamer should click on a letter of the alphabet to guess that letter. If the gamer is correct, that letter will appear. If the gamer is incorrect, more of the gallows will be drawn.

The hangman game is one CSP page, which is installed into the virtual file system at system startup. The web-browser will send an initial HTTP GET request to the server the first time the gamer requests the page. The server will forward the request to the compiled CSP page, which is now part of the virtual file system. Subsequent requests from the browser will be HTTP POST requests, which transmit data from the browser to the server.

The compiled CSP page renders a new HTML page and sends the page back to the browser for every request. The generated HTML file will therefore look different for every request. It is the C code embedded into the CSP page, which generates the dynamic part.

A CSP page is an ordinary HTML file with added C or C++ code for rendering the dynamic parts of the page. The C or C++ code is surrounded by, for example, the delimiters `<%` and `%>`. The tags are similar to the ASP tags, but there is a difference between ASP and CSP. ASP is compiled inside the web-server, but CSP is pre-compiled into C or C++ code on a host computer. This code is then compiled with a C/C++ compiler and linked with your embedded system.



Designing the hangman framework

The first part of the design is to identify the static components of the game. The static components are the parts that will always look identical. This is typically done by an HTML designer. We have deliberately made a very basic HTML framework. A real-world application, which is designed by an HTML designer, would usually have a more appealing and professional looking user interface.

You can open the complete [hangman game source code](#) in a separate window and use the source code as a reference when we discuss various code snippets from the game.

The HTML framework designed by the HTML designer:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Draft//EN">
<html>
  <head><title> hangman </title></head>
  <body bgcolor='white' text='#000010' link='blue' vlink='#ff00ff'
link='#ffff00'>

    <div align=center>
      <font color='#ff3333' size=+5>H</font>
      <font color='#cc6633' size=+5>a</font>
      <font color='#999900' size=+5>n</font>
      <font color='#33cc33' size=+5>g</font>
      <font color='#339999' size=+5>m</font>
      <font color='#3366cc' size=+5>a</font>
      <font color='#0033ff' size=+5>n</font>
    </div>

    <div align=center>
      <font size=+3 color='#336699'>
<!-- Letters of the correct word and blank spaces here -->
      </font>
    </div>
    <br/>
    <div align=center>
<!-- one of the 8 gallows images here -->
<!-- more dynamic text here -->

<!-- The following form should only be emitted when playing a game -->
      <form method='post' action='hangman.html'>

<!-- The 3 value attributes below must be dynamically updated -->
        <input type='hidden' name='wordDbIndex' value=''/>
        <input type='hidden' name='guess' value=''/>
        <input type='hidden' name='noOfUsedGues' value=''/>

        <input type='submit' name='ch' value='A'/>
        <input type='submit' name='ch' value='B'/>
<!--Letter C to Y omitted, must be added -->
        <input type='submit' name='ch' value='Z'/>
      </form>

    <p>
      <a href = "hangman_src.html" target="_blank">view html source code</a>
    </p>
  </div>
</body>
</html>
```

Designing the C code infrastructure

The HTML designer left a number of HTML comments in the above HTML page. It is now the C programmer's job to replace these comments with C or C++ code enclosed within server side tags.

Before the C programmer can start filling in the blank spaces (replace the comments) in the CSP file, the infrastructure for the game must be designed. The C code for handling this logic should normally be left outside of the CSP file, but we have included all the code into the CSP page for the purpose of this article. Web-designers always talk about separating the presentation and the logic and one should therefore limit the amount of CSP tags with C or C++ code in the CSP file.

One problem with designing the hangman game is that we must have some mechanism that can keep state information in between HTML page requests. HTTP is a stateless protocol and can not help us in maintaining the state information. There are basically two methods for maintaining state information between page requests.

- Persistent session object: Provides a way to identify a user across more than one page request or visit to a Web-Server and to store information about that user.
- Hidden variables: Invisible HTML tags that stores information in between requests.

Barracuda supports persistent session objects, but the HTML designer, which designed the above HTML template, assumed we wanted to use hidden variables. Hidden variables are adequate for the hangman game, but the use of hidden variables can easily get ugly in more advanced applications. This is when session object comes to the rescue and can be used when a more advanced state architecture is required.

Game state information is stored in 3 hidden HTML variables:

```
<!-- The 3 value attributes below must be dynamically updated -->
  <input type='hidden' name='wordDbIndex' value='' />
  <input type='hidden' name='guess' value='' />
  <input type='hidden' name='noOfUsedGues' value='' />
```

The HTML designer assumed we needed 3 hidden variables for storing state information. It is the C programmer's responsibility to update these variables in between page requests. The state variables stored in the HTML page maintains the game's internal state information for the current gamer.

An interesting feature of this game is that it is possible to cheat by pressing the back button in the browser. Say that you almost got hung, but you got some letters right, then all you have to do is to press the back button until you do not see the gallows. You can now restart the game and enter the letters you had right so far. This works because the state variables are stored in the HTML page and the browser cached the old pages. By pressing the back button, you are in fact rewinding the game.

Using a session object for storing the state information would have made it impossible for the gamer to cheat as the state information would have been stored on the server. As you can see, storing state information in hidden variables can lead to unwanted results and a possible security breach in your system.

Writing the C code

We have designed this version of the hangman game in C++. Barracuda is a C library with special C++ wrappers in the header files. The C interface is similar to the C++ interface.

The first we need is the dictionary:

```
86 struct Word
87 {
88     const char* clue;
89     const char* answer;
90 };
91
92 const Word wordDB[] = {
93     {"action", "drool"},
94     {"action", "forecast"},
95     {"action", "forget"}, ----- more code
```

This is declared within the `<%g %>`, the CSP Global tag is typically used for declaring static data in the CSP file.

We can now start looking at what goes into the CSP page's service function. The service function is automatically generated by the CSP compiler. You would not normally see this function unless you look at the C or C++ code generated by the CSP compiler. The web-server calls the service function when the browser requests the page. The C/C++ code in the following CSP tags are executed within the service function: `<%p %>`, `<%e %>`, `<% %>` and `<%= %>`.

The service function has the following prototype:

```
Hangman::service(HttpRequest* request, HttpResponse *response)
```

The request object contains all the information you can possibly get from the client. This information is pre-processed and presented as sub-objects within the request object. For example, a client sending data to the server using POST is pre-parsed by the web-server and stored as objects in the request object.

The response object is used for sending the response message. For example, the CSP compiler extracts all the static HTML data in a CSP file and automatically creates C code for sending the data back to the client using the response object. The embedded C/C++ code within the CSP tags can also use the response object for sending data to the client.

The following code snippet is from within the prolog CSP tag, the <%p > tag. C code within the prolog tag is guaranteed to execute first within the page service function, no matter where in the HTML file you insert the tag:

```
252 unsigned int i;
253 const char* ch=0;
254 const Word* word=0;
255 int wordDbIndex;
256 const char* guess=0;
257 char* newGuess=0;
258 int noOfUsedGues = 0;
259 HttpParameterIterator fIter(request);
260 /* Extract data from form in the above html */
261 for( ; fIter.hasMoreElements() ; fIter.nextElement())
262 {
263     if( ! strcmp(fIter.getName(), "guess" ) )
264         guess = fIter.getValue();
265     else if( ! strcmp(fIter.getName(), "ch" ) )
266         ch = fIter.getValue();
267     else if( ! strcmp(fIter.getName(), "wordDbIndex" ) )
268     {
269         wordDbIndex = atoi(fIter.getValue());
270         word = &wordDB[wordDbIndex];
271     }
272     else if( ! strcmp(fIter.getName(), "noOfUsedGues" ) )
273         noOfUsedGues = atoi(fIter.getValue());
274 }
275 if(word) /* Should be true unless first time or a new game.
276          * This works since the form in the html above is only
277          * emitted if we are playing a game
278          */
279 {
280     /* See hangman game for complete source code */
281 }
282 else /* First time or new game */
283 {
284     wordDbIndex = rand() % (sizeof(wordDB) / sizeof(wordDB[0]));
285     word = &wordDB[wordDbIndex];
286     /* See hangman game for complete source code */
287 }
304 }
```

The above code declares a parameter iterator (line 259), which is used for fetching all of the variables (including the hidden state variables) from the FORM declared in the CSP page. The first request is a GET request and the parameter container will therefore be empty. When the gamer press one of the A to Z letters, the browser will send a POST request and the FORM variables will be retrieved from the parameter container, by using the above HttpParameterIterator.

GET request (Line 298 - 304):

If it is a new game, a new word is randomly selected using "rand()". The current word offset position in the database is later stored in the wordDbIndex hidden HTML variable. This will be explained later.

POST request (Line 279 - 296):

If the gamer presses one of the buttons in the hangman game, the browser sends the FORM data to the server. The loop (line 261 - 274), using the iterator, in the above code extracts the current game's state variables and the A-Z button pressed.

Replacing the HTML comments with CSP tags

It is the C programmer's responsibility to add the dynamic HTML rendering to the HTML page. The C programmer should replace all the comments in the above HTML file with CSP tags.

The first HTML comment in the HTML page is:

```
<!-- Letters of the correct word and blank spaces here -->
```

We replace the above comment with the following code:

```
18         <%  
19         for(i = 0; i < strlen(newGuess) ; i++)  
20             response->printf("%c ", newGuess[i]);  
21         %>
```

The newGuess variable is a C array, which contains the letters guessed so far. The array is initially filled with underscores and the underscores are replaced by the correct letters when the gamer correctly guesses the letter.

The above loop simply loops through all the letters in the array and print out the letters and spaces in between. For example, the following may be emitted when a new game is started:

```
- - - - -
```

The response object is in charge of sending data back to the client. The CSP page automatically emits the static part of the HTML page to the response object. The CSP tags are inserted into the HTML such that we can inject the dynamic content into the page. The printf member function in the response object works like the regular printf function, except for that the data printed is not sent to some console, but to the client browser window.

The comment:

```
<!-- one of the 8 gallows images here -->
```

is replaced with:

```
26      <img src='images/hang<%= "%d" noOfUsedGues%>.gif' />
```

We have a total of 8 gallows images. The above code emits HTML code for selecting one of the 8 images. The variable `noOfUsedGues` is a number between 0 and 7, thus the emitted HTML will be `` if the variable `noOfUsedGues` is zero. The gamer has a total of 6 possible failures before hung, thus the first 6 images (0-5) are images of an appearing gallows. Image 6 is shown if the gamer failed and image 7 is shown if the gamer won.

The next HTML comment is:

```
<!-- more dynamic text here -->
```

is replaced with:

```
31      <p>
32          <font size=+3 color="#ff3333">
33              <%= noOfUsedGues == 6 ? "Game over." : "You Won!!!"%>
34          </font>
35      </p>
36      <p>
37          <font size=+2 color="#3333cc">
38              The correct answer is <B><%=word->answer%></B>.
39          </font>
40      </p>
41      Play another game of <a href='hangman.html'>Hangman</a>
42  <%
43  }
44  else
45  {
46      if(noOfUsedGues == 0)
47      {
48  %>
```

The first condition above is to test for `noOfUsedGues = 6` or `7`, which means that the game is over. The user won if it is `7` and failed if the variable is `6`. A game is still in progress if the variable is between `0` and `5`. Please refer to the `hangman.html` source code for the complete code.

The above CSP code snippets shows how one can make static HTML code conditional by enclosing the HTML code within CSP tags. For example:

```
<%
  if(someCondition)
  {
%>
  <p>
    The condition is true.
  </p>
%>
  }
  else
  {
%>
  <p>
    The condition is false.
  </p>
%>
  }
%>
```

The HTML in the first section will be emitted if the `someCondition` variable is true, and the second HTML section will be emitted if the `someCondition` variable is false.

Creating the form and maintaining the state variables

```
61         <form method='post' action='hangman.html'>
62             <input type='hidden' name='wordDbIndex' value='<%= "%d"
wordDbIndex%>' />
63             <input type='hidden' name='guess' value='<%= newGuess%>' />
64             <input type='hidden' name='noOfUsedGues' value='<%= "%d"
noOfUsedGues%>' />
65             <% for(i = 'A'; i <= 'Z'; i++)
66                 response->printf("\t\t<input type='submit' name='ch'
value='<%= c%>' />\n", i);
67             %>
68         </form>
```

The first 3 hidden variables in the form are dynamically created by the CSP page every time the gamer fetches a new HTML page from the server. This is how we maintain the game's state information. We showed you how to extract the FORM data from a POST request in the above C++ code snippet (line 261 - 271), by using a `HttpParameterIterator`.

The next part of the form emits the A to Z buttons. The HTML framework designed by the HTML designer had 26 static input tag fields for the A to Z buttons. The C programmer realized that there is no need to have 26 static input fields when one can dynamically create the 26 fields from a simple loop. This makes the CSP page smaller and thus, one uses less memory.

How it works

A CSP page is converted into C or C++ code by using our CSP compiler. Our CSP compiler is known as CspCompile. The C or C++ code generated by the CspCompiler is compiled into object code using a C/C++ compiler and linked into your application. All of the C/C++ code enclosed within the CSP tags will go into the page service function, which is automatically generated by the CSP compiler.

The C/C++ code produced by the CSP compiler inherits from the `HttpPage` type. An instance of an `HttpPage` can be inserted into the virtual file system and will be called by the web-server when a user requests the page. A `HttpPage` is very similar in functionality to a [Java Servlet](#).

It is also possible to write your own version of an `HttpPage` and insert the page into the virtual file system. It is sometimes better to write an `HttpPage` directly instead of using the CSP compiler if the page contains very little presentation but a lot of logic.

If you look at the [C++ code generated by the CSP compiler](#) for the Hangman game, you will see that all the C++ code we added to the CSP page is now part of the generated code. You will see the beginning of the service function if you go to line 184. You will also see a "C #line pre-processor directive" a few lines down. The CSP compiler emits line directives into the generated code such that a C compiler can show the correct line number in the HTML file. This means that the C compiler will show you the line of the error in the HTML file if you should make a typo in the embedded C code. Most C compiler supports the #line directive and will show you the CSP page and not the auto-generated C++ code. This is also the case when debugging the code with a debugger.

You will see the following code if you go to line 243:

```
if(httpWriteSection(this->data,response,this->blocks[0].offset,this->blocks[0].size))
    goto L_epilogue;
```

The code emits one of the static HTML sections we had in the CSP page. The actual data is not embedded into the code, but kept separately in another file. Barracuda is very flexible in the way it handles HTML sections in the CSP pages.

One can have the web-server read the sections from:

- A file system.
- Directly from Flash memory.
- One can convert the data file to a large C array, which can then be compiled and linked with the application.

If the "httpWriteSection" function returns a negative number, the processor will jump to the epilogue code. We do not use C++ exceptions since most embedded systems do not use them. Instead we use the "goto L_epilogue;", which is to where the `<%e >`, the code enclosed within the CSP epilogue tag is inserted.

Conclusion

We have now shown you how easy it is to use the CSP tag language for making advanced dynamic HTML on the server side. We have barely scratched the surface of the endless possibilities of the powerful CSP language. Barracuda also provides a massive C and C++ API for working with server side scripting.

Exercise

We discussed the danger of maintaining state information inside the HTML page and that it is better to use a persistent session object for maintaining the state information.

If you have the Barracuda development environment, a good exercise would be to rewrite the game to use a persistent session object for storing the state information.

The CSP page contains many CSP tags and can be hard to read. A better design is to break the CSP page into several pages such that it is easier to read. The pages can be combined at runtime using `HttpResponse::include`.

Introduction to web-security and the Barracuda Virtual File System

A device that is constantly connected to the internet needs to be protected from being used by unauthorized users. Internet security can be divided into the following categories.

- Authenticating users.
- Authorizing an authenticated user.
- Preventing eavesdropping and modification of data.

The web-infrastructure has well defined API's for all of the above security mechanisms. The Barracuda Web-Server implements and makes it very easy setup a protection mechanism for your embedded device. All browsers today implement the above security types. Thus, by piggybacking on the well defined web-security mechanisms, your device can be configured with advanced security control in no time.

It is needless to say that the above security mechanism would be very hard and time consuming to implement if you use a proprietary protocol for controlling your device.

It is suggested to read the [device control](#) and the [CSP tag language](#) white papers before you continue reading this white paper.

We have a [small demo program](#), which implements a secure photo album. We suggest that you download and try this demo program. The demo program is a Windows NT/XP executable.

Preventing eavesdropping and modification of data

Eavesdropping and modification of data is prevented by using a Secure Socket Connection (SSL). SSL is the recommended method for protecting sensitive information.

Creating a `secure socket listener` object is similar to how we created an `insecure socket listener` object in the device control white paper.

A detailed explanation of how to use and initialize the SSL engine is outside the scope of this article. Please see our partner's SSL white paper for more information.

Authenticating users

The HTTP protocol has support for two authentication mechanisms, Basic Authentication and Digest Authentication. The Barracuda Web-Server has support for Basic and Digest Authentication. The Web-Server also supports form based login.

- Basic Authentication - a method, supported by HTTP 1.1, for requiring a username and password to be supplied for protected resources. If the authentication fails, the server returns HTTP status code 401 (Unauthorized). Basic authentication transmits username and password as plain text making it insecure, unless protected by SSL.
- Digest Authentication - a method, supported by HTTP 1.1, which is similar to Basic Authentication, but where password data is encrypted for transmission using a hashing algorithm. This method is far from as secure as SSL and is for this reason considered "poor man's encryption".
- Form Authentication - the login process is performed by a custom web-page, allowing us to customize its appearance and the nature of error messages. More importantly, it allows us to apply more secure authentication methods. In order to be secure, form authentication should be used in conjunction with SSL.

Basic and Digest Authentication sends a HTTP challenge command to the browser. The browser will then open a pop-up window, where the user can enter a user name and a password.



The image shows a Windows-style dialog box titled "Prompt". It has a blue title bar with a question mark icon on the left and a close button (X) on the right. The main content area is light beige and contains the following text and controls:

- A question mark icon followed by the text: "Enter username and password for "DemoRealm" at 127.0.0.1:8080"
- The label "User Name:" followed by a text input field.
- The label "Password:" followed by a text input field.
- A checkbox labeled "Use Password Manager to remember this password." which is currently unchecked.
- At the bottom, there are two buttons: "OK" and "Cancel".

A weakness in the Basic and Digest Authentication mechanism is that the server cannot customize the login pop-up window, thus both Basic and Digest Authentication is considered to be less user-friendly. It is today more common to use FORM based login in conjunction with SSL and this is also our recommended login method.

Basic, Digest and Form based logins in the Barracuda Web-Server are implemented by using a persistent session object. Many other embedded web-servers do not have persistent session handling, thus they do not give you FORM based login as an option since FORM based login requires persistent session handling.

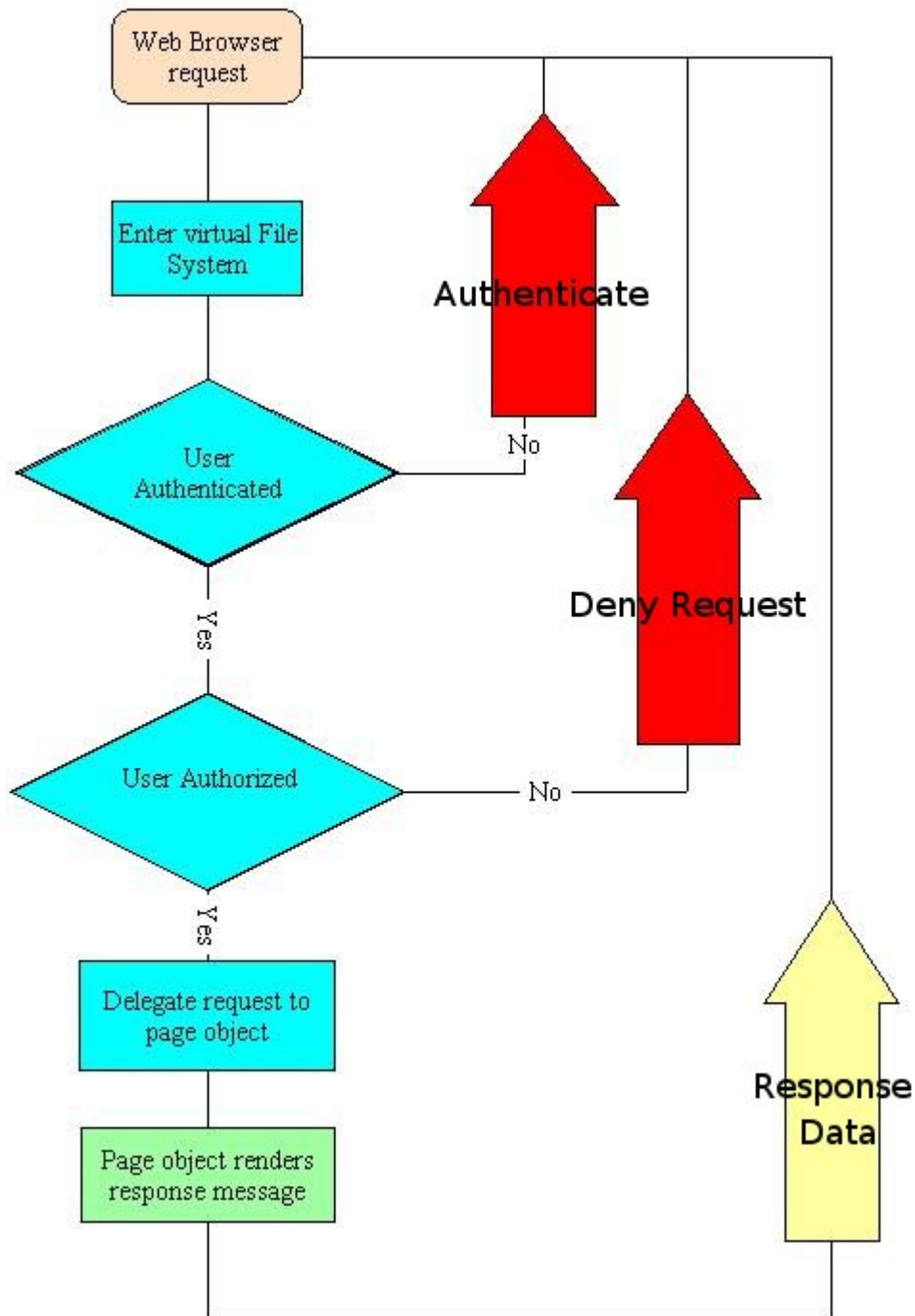
Authorizing users

One must have an understanding of principals and roles for understanding how the authorization mechanism works. A principal basically identifies an entity that can interact or perform work in a system. It can be a person, company - just about anything in fact.

A role groups certain actions together, and we can then specify certain principals having particular roles, thus giving those principals clearance to perform the action.

This is similar to a concept of users and groups in a UNIX system, where users are generally people that may access the system, and groups represent the position that users can hold. For example, a company system may have a user called Allen Smith, who belongs to the group, human resources.

Thus an authenticated user might not have clearance to perform certain actions. The authentication and/or authorization process may be seen as:



Authentication and authorization is performed on a directory level in the web-server. A user might first be asked to login when descending into a directory branch. When the user has successfully been authenticated, the user's principal is checked with the directory node's role. If the user is part of the role, the user is allowed to access the directory. A directory hierarchy can be setup to be more restrictive as the user is descending into the sub-directories thus the user might be denied access further down the directory hierarchy.

We must first explain how the virtual file system works before we can further discuss how to use the security mechanism in the web-server.

The virtual file system

We briefly mentioned the virtual file system in the device control white paper, where we explained that HttpDir is the directory node and HttpPage is the page node. An instance of the HttpDir class can contain sub-directories and/or pages and this is how the directory hierarchy in Barracuda works.

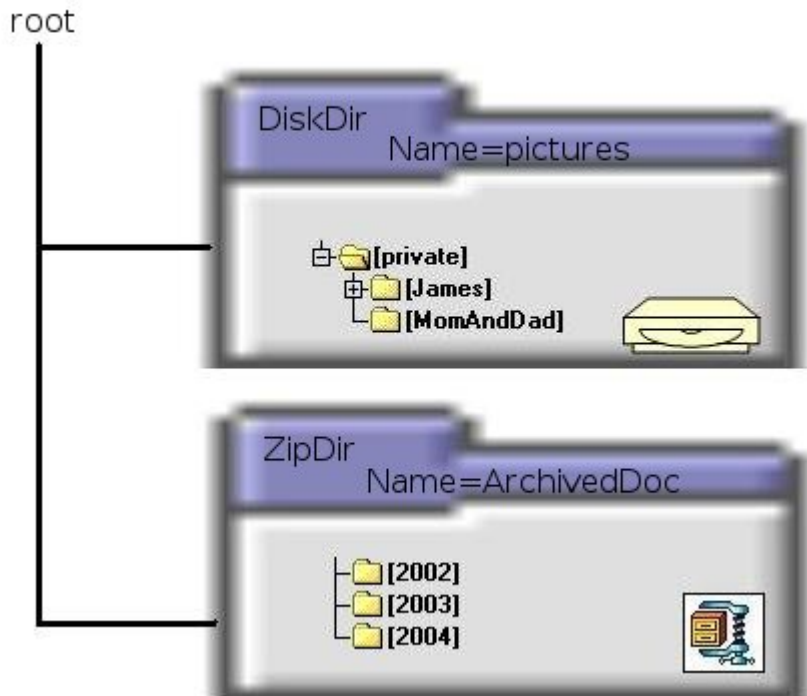
One of the unique features of the HttpDir class is that its functionality can be overloaded, thus by inheriting from the HttpDir class, one can implement unique features for each directory node. Barracuda comes with a number of pre-defined classes that inherit from the HttpDir class.

Two very useful classes, which inherit from HttpDir are:

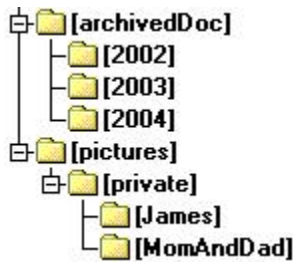
- DiskDir is the interface to a regular file system. A DiskDir node extends the URL search path into searching a file system for the requested file.
- ZipDir turns a ZIP archive into a compressed read-only file system.

Now let us assume that the Smith family wants to setup a web-server with a family picture album and with a number of archived documents. The archived documents will not be changed, so the Smith family decides to zip the archived documents into a zip file. New pictures might be added to the album when available so a zip file is not a good solution. The Smith family decides to use a DiskDir for reading the photo album.

The Barracuda virtual file system is set up with a root directory node. The root directory is setup with one instance of the DiskDir class and one instance of the ZipDir class.



A browser requesting documents from the server will see the various HttpDir nodes in the virtual file system as a coherent directory tree. From the clients perspective, the virtual file system can be seen as the following:



For example, an URL such as <http://Smith.com/pictures/p1.jpg> makes the DiskDir object search its root path for p1.jpg. The root path is setup when the DiskDir object is created. It could be, for example, "c:\Documents and Settings\users\My Documents\My Pictures" if run on a Windows machine.

For the above directory structure, a security manager is constructed such that only family members can access the "ArchivedDoc" and the "private" directory. The "picture" directory is for public access. The security manager is also setup such that only James is allowed to access the "James" directory and James mother and father are the only ones that can access the "MomAndDad" directory. We will discuss how to install a security manager later.

A virtual directory structure can consist of many HttpDir instances and of specialized types that sub-class and overload the functionality in the HttpDir class.

A virtual directory structure can be created manually as we did in the device control white-paper or be automatically created by the powerful host tools that come with the Barracuda Web-Server.

We used the CSP compiler in the CSP tag language white-paper for compiling the hangman game. What we did not explain is that the data file produced by the CSP compiler contains information about the relative path to the page. You can think of this data file as an object file. The CSP linker links all these data files together, calculates the internal offset positions and produces two files.

- A data file - which contains all of the page object's presentation (HTML) data.
- A C file - you compile and link with your system. This code automatically creates the virtual files system when the web-server starts.

The possibilities with the virtual file system are almost endless. In this article, we have limited ourselves to covering only the basics.

The Barracuda security manager

The Barracuda security manager is a collection of a number of classes. It should be no surprise that the security functionality is installed in the virtual file system and that a specialized version of the `HttpDir` type overloads the base functionality by adding authentication and authorization to the directory node. The sub-classed `HttpDir` class is called `AuthenticateDir`. The `AuthenticateDir` class is just a few lines of code. All of the security related logic is kept in a number of classes that is referenced by `AuthenticateDir`. [A detailed explanation of the security classes can be found here.](#)

The `AuthenticateDir` service function:

```
1 static int AuthenticateDir_service(AuthenticateDir* o,
2                                   const char* relPath,
3                                   HttpResponse* response)
4 {
5
6     if(AuthenticatorInterface_authenticate(
7         o->authenticator,o->roles,response))
8     {
9         /*Authenticated. Call the directory service function.*/
10        return (*o->orgService)((HttpDir*)o, relPath, response);
11    }
12    else
13    {
14        /* User is not authenticated.
15         * Return "file found" such that the virtual file system
16         * does not look for duplicate directories.
17         */
18        return 0; /* 0 = found i.e. no error.*/
19    }
20 }
```

- The directory service function is activated by the web-server or the top directory when the browser sends a request. The `relPath` argument (line 2) is the relative path for the current directory node. The relative path comes from the requested URL.
- Line 6 is where we delegate the authentication and/or authorization to the authentication/authorization logic. The function returns true if the user is authenticated and authorized.
- Line 10 is where we delegate the request to the original service function (the overloaded service function). Recall that Barracuda is a C library and that we cannot use virtual functions. We must instead use function pointers. What we do is to call the original `HttpDir` service function, which delegates the request to the correct `HttpPage` object.

The `AuthenticateDir` class can easily be extended to perform customized security checks. For example, one could bypass the security manager if the request is from the local network or prevent certain domains from even accessing the directory.

One could also create a simple filter that denies the request if the request is not using a secure channel (SSL). The service function could either send back a response message that the client must use a secure channel, or the service function could automatically redirect the client to a secure channel.

Conclusion

Web-security is a complex and important topic. This article has given you an introduction to the Barracuda security mechanism and the virtual file system. The Barracuda security mechanism models the [security mechanism in J2EE](#), though in a simplified form. Many good books can be found that cover web-security and especially the J2EE security mechanisms. We also discussed the benefit of using a custom FORM login instead of using Basic or Digest HTTP authentication. Our Barracuda standalone Windows demo has an example of a customized FORM login, which clearly demonstrates the user friendliness of a customized login. You see the custom FORM login example when you press the "Barracuda Documentation" link in the standalone Windows demo.

A Trace Tool

Using the EventHandler

Most embedded development tools come with some sort of debugging aid. For example, an integrated environment such as the Metrowerks IDE can be used for setting breakpoints and stepping through the source code in an embedded device. Many of these debuggers run in so called freeze mode, thus effectively stopping all tasks when a breakpoint is hit. It is for this reason common to instrument the code with printf's such that one can debug or trace the code without halting the target.

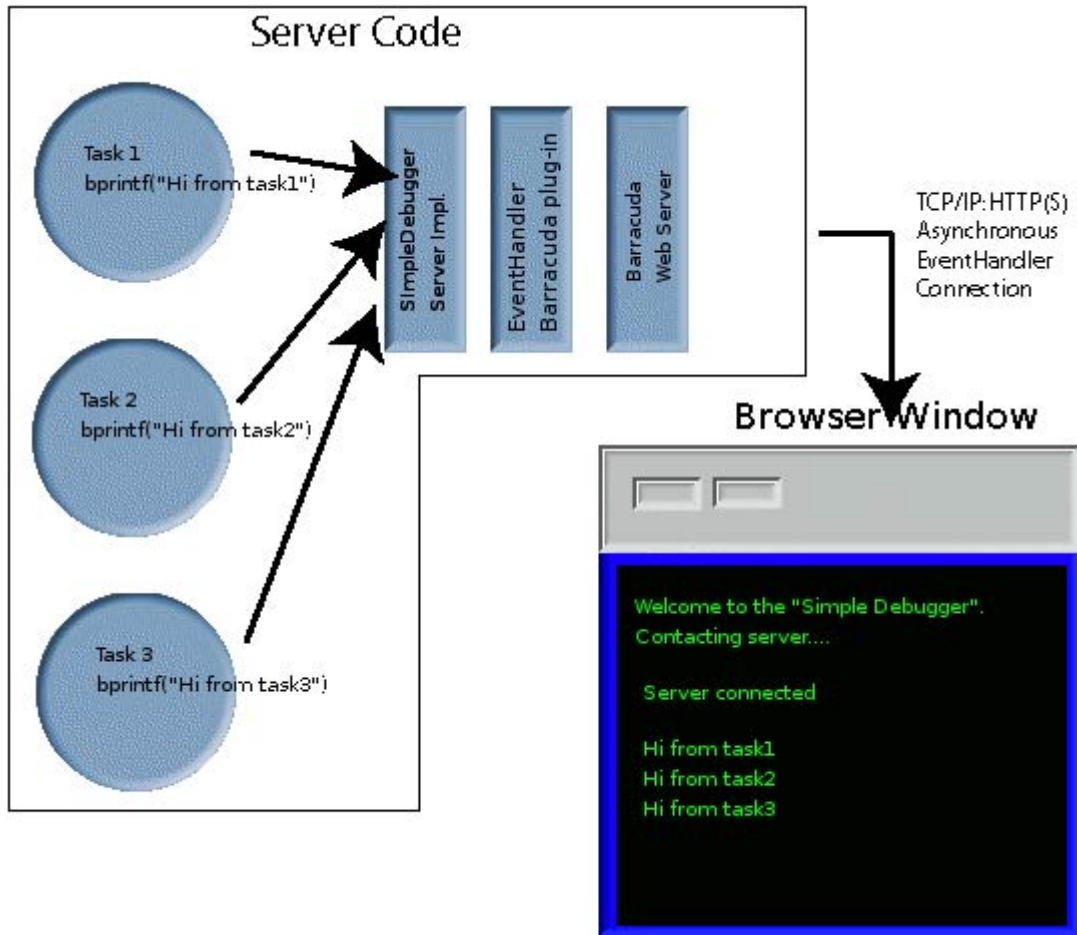
Using printf in an embedded device is sometimes a challenge as not all devices have a console.

It is very easy to create a trace library using the Barracuda Web-Server and the Barracuda EventHandler plug-in. A browser window will be used as the console. By connecting a browser to the server and using the persistent connection feature of the EventHandler, the trace data can be sent in real-time from the server to the browser window.

The following print functions will be created.

```
void bprintf(int prio, const char* fmt, ...);
void bvprintf(int prio, const char* fmt, va_list argList);
void bflush(void);
```

The functions bprintf and bvprintf works just like the ANSI C printf and vprintf functions, except for the first argument, which is the message priority. The trailing 'b' in the function names is short for browser window -- i.e., the message is sent to a browser window and not the standard console. Data is not immediately sent to the client, but is buffered in the server code. Function bflush flushes whatever data that is in the buffer to the client.



Designing Rich Client User Interfaces

The client side of the simple debugger cannot be constructed using static HTML. A client side application using the EventHandler must be implemented in [JavaScript \(ECMAScript\)](#). A client application that can dynamically change the GUI can be referred to as Dynamic HTML or just [DHTML](#).

A simple template for such an application is shown below.

```
<html>
  <head>
    <script>
      onload=function() {
        };
    </script>
  </head>
  <body>
  </body>
</html>
```


As you can see, the HTML body tag contains no HTML elements. Elements will be added and removed to the browser's DOM by using JavaScript code. The script tag contains one event function declaration in the above example. The onload event is automatically triggered by the browser when all components of a HTML page is loaded. We will later use this function for starting up and connecting the EventHandler to the server.

A DHTML client using the EventHandler is loaded and initialized as follows:

1. The user requests the page from the server by typing in the URL to the DHTML client interface.
2. The browser sends a GET request to the server.
3. The server responds by sending the DHTML page to the client.
4. The browser parses the DHTML page.
5. The browser loads images and external referenced JavaScript code by sending GET requests to the server.
6. The browser calls the onload function when all external referenced files are loaded.
7. The onload function initializes the EventHandler and sets up a persistent connection to the server.

Writing and testing the initial code without using Barracuda.

As you saw in the above DHTML template, the HTML body contained no HTML elements. Most graphical user interfaces contain parts that never change and our simple debugger is no different. We will add some HTML code to the body of the document and add most of the JavaScript code for the client.

The client can at this point be developed without using the Barracuda Web-Server since the client interface does not require server side scripting. A simple text editor and a browser is all that is needed for doing nearly all of the client side development.

Development of the user interface is ideally left to a DHTML programmer. A DHTML programmer can develop and test the GUI code offsite. At this point in the development, the server interface can be simulated. When the GUI development is completed, the stub code can be replaced by the interface functions defined in the EventHandler Interface Definition File.

We have added some stub functions (test code) to the [Simple Debugger](#) such that we can test some of the functionality without the server. Please open the [Simple Debugger](#) Test Code.

The [Simple Debugger](#) Test Code prints out "Contacting server...", but the test code will not contact the server since we have not added the EventHandler code to the DHTML test code at this stage.

The EraseTrace button is the only code completed so far. The button simply erases all data in the client.

The Enable/Disable Trace button will eventually send an "enable/disable trace command" to the server. The Enable/Disable Trace button currently prints out some text in the console. This is used to test how text sent from the server is formatted in the console window.

The Trace Level combo box lets the user select the trace priority. A total of 10 priority levels can be set. Level 0 is the highest priority and if selected, only trace messages with priority level 0 are sent to the console.

You should now study the source code for the [Simple Debugger](#). **Right click in the [Simple Debugger](#) browser window and select "view source".**

Do you have a difficulty understanding how the client side code works? Now is a good time to read up on the following technologies:

- HTML.
- Cascading Style Sheets.
- JavaScript.
- Using JavaScript and the Document Object Model.

The online and free w3schools.com is an excellent resource for all of the above technologies.

Creating the interface definition file (IDL)

The next step is designing the interface definition file. Barracuda comes with an IDL compiler that can translate the IDL definition into JavaScript code and C/C++ code.

SimpleDbg.ehi:

```
client SimpleDbgServer2ClientIntf
{
  asynchAddText { string text; }
  asynchOnInitSetStatus { int traceButtonState; int tracePriorityLevel; }

server SimpleDbgClient2ServerIntf
{
  setTraceButtonState { int state; }
  setTraceLevel { int priority; }
}
```

As you saw from the [Simple Debugger](#) Test Code, we have already constructed the code for the server to client interface. The EventHandler stub compiler otherwise generates the skeleton for these files.

The EventHandler stub compiler also generates some C header files and C source files. You can either generate C code or C++ code. We have used C code for this example.

Adding the client side EventHandler code

The client side EventHandler code, which is in a separate JavaScript file, must be loaded into the DHTML client. The following two declarations are added to the header section.

```
<script src="/eh/jseh.js"></script>
<script src="/autogen/SimpleDbgClient2ServerIntf.js"></script>
```

The first line loads the EventHandler stack and the second line loads the JavaScript stubs that were generated by the EventHandler stub compiler when we compiled the SimpleDbg.ehi interface definition file.

The EventHandler and the server stub interface are created in the onload function:

```
var eh = new EventHandler("/SimpleDebugger.interface");  
var serverIntf = new SimpleDbgClient2ServerIntf(eh);
```

The first line creates an instance of the EventHandler stack. The argument is the URL to the server side of the SimpleDbgServer2ClientIntf, which we declared in the SimpleDbg.ehi interface definition file.

The next step is to add code for sending commands to the server from the local event functions. For example, the clientToggleTrace event function in the [Simple Debugger](#) Test Code can disable the trace by calling the following function:

```
serverIntf.setTraceButtonState(0);
```

Re-enable the trace is:

```
serverIntf.setTraceButtonState(1);
```

ANSI C and object-oriented programming

Barracuda, which is an object oriented C library, provides a C code API and a C++ code API. In this white paper, we have used the C interface.

We suggest that you read the [introduction to Object Oriented programming in C code](#) before you continue reading this white paper.

Downloading the example code

It is now time to download the example code. The example code contains all source code for the [Simple Debugger](#) and a pre-compiled executable server for MS Windows. The server starts automatically when you run the [example code's self extracting ZIP file](#).

The source code you should study is:

html\SimpleDebugger.html The DHTML client.

src\SimpleDebugger.c The server [Simple Debugger](#) implementation.

The server side implementation of the Simple Debugger

The [Simple Debugger](#) implementation is designed such that only one client can be connected to the server. You can test this by opening a separate browser window and type in the URL to the SimpleDebugger. The server is a shared resource, and one must have this in mind when designing a client server application. It would be easy to change the [Simple Debugger](#) to be multi user enabled, but the purpose with this demo is to show you how a user can exclusively lock the server application.

Typically, a multi user enabled application should automatically synchronize all connected DHTML clients. For example, if one user changes the trace level in the combo box, then all connected clients should be updated. The way this works is that the client changing the trace value sends a change command to the server. The server responds by sending an asynchronous change user interface command to all connected clients.

You can download the [Barracuda demo](#) if you would like to see examples of multi user enabled applications. The Barracuda demo contains a multi user enabled slide show and a multi user enabled MP3 player.

The server side implementation of the Simple Debugger is in src\ SimpleDebugger.c.

The SimpleDebugger source code is heavily commented, but in short:

The core server functionality is encapsulated in class SimpleDebugger. The SimpleDebugger contains the following member functions:

newClientCon	called when a new client initiates a persistent connection.
clientConTerminated	called when a client terminates a persistent connection.
setTraceButtonState	called when the user presses the Enable/Disable Trace Button.
setTraceLevel	called when the user selects a new priority value in the Trace Level combo box.
send2Browser	Sends data to client when trace buffer is full.
vprintf	Works like the ANSI C vprintf function, except for that the data is sent to a browser window via function send2Browser.
flush	Send data in trace buffer to client.
constructor	Initializes all data for this class.

Conclusion

JavaScript code can manipulate the DOM in the browser window, thus creating rich user interfaces. One can either use the DOM directly as we did in the Simple Debugger or use a widget library such as [Bindows](#).

Another possibility is to use [SVG](#). Scalable Vector Graphics (SVG) is an exciting new XML-based language for Web graphics from the [World Wide Web Consortium](#) (W3C). One can create advanced interactive graphical user interfaces using SVG. The [Barracuda demo](#) contains an example of a rich client interface implemented in SVG.

One can easily design browser based rich client interfaces that can offload the processing required in the embedded systems. Embedded systems are normally resource constrained and offloading the processing to a client can ease the burden on the embedded device. Most host computers today are very powerful and can easily execute large JavaScript applications.

JavaScript is a very interesting language and although it looks similar to C code at first glance, you will soon discover the power of this amazing language. A JavaScript application can be compressed and stored in the server. The client loads the compressed image, deflates the code and starts executing the application.

JavaScript is normally limited to the environment exposed by the browser's JavaScript engine, but our EventHandler extends the functionality of this language. One can easily design complex distributed applications using the EventHandler.