Developing CGI-based AWS Applications Using Digi's NET+OS Development Environment

1 Document History

Date	Version	Change Description	
6/4/2010	V1.0	Initial Entry	
6/8/2010	V4.0	Continued entry	
6/16/2010	V5.0	Textual cleanup	

2 Table of Contents

3 Introduction

Common Gateway Interface (CGI) is a method of developing web pages where you the developer take control of what and how your project's web pages are displayed, away from the web server. Taking over this control has both benefits and costs to you as a developer of web applications.

This white paper discusses the mechanics of developing a CGI-based web application using Digi's advanced web server (AWS) as well as some of the strengths and weaknesses of developing web applications using the CGI method. Digi's AWS is a component of Digi's NET+OS embedded development environment.

Click here to download the example code referenced in this document.

3.1 Problem Solved

The lion's share of developers using Digi's NET+OS development environment, that are developing applications that contains a web component, will develop their web pages in, what we call, the conventional way. We define the conventional way the use of the following steps:

- Develop some html pages
- Run the html pages through the pbuilder utility
- Fill in the "stub functions" which were generated by the pbuilder utility with code to access and/or update device data.
- Compile, link, build and download your application.

Using this method, the AWS hands web page data to the stub functions, on a function-by-function basis. For many customers, this method is adequate. However, for a subset of our customers, this is inadequate. These customers require more control of the data and the generation of page content. For these customers CGI is the solution. One of the main differences between the "conventional" method and the CGI method of web project development is that for pages processed through CGI, the pbuilder utility is not used for generating C code and the associated stub functions. This might leave you asking, then how does my NET+OS application communicate with my browser and exchange html/http data? This paper explores the answer to this question. Hopefully, by the end of this white paper you'll understand the answer to this question.

3.2 Audience

This white paper is written for a particularly technical audience. Users of this document should be seasoned NET+OS developers with experience in AWS development and possess, at least, an understanding if not working knowledge of developing CGI-based applications on other platforms (for example on an Apache web server on a PC or a Linux system).

3.3 Assumptions

You are probably reading this white paper because, either you are finding the pbuilder-based, stub function-centric method of AWS development too restrictive or you have performed CGI development on other systems and want to know how to do so in the NET+OS development environment. Either way, I am assuming that that you know what CGI is and you have some knowledge of how CGI interfaces with web servers and web browsers.

If you have no knowledge of NET+OS-based AWS development, then you probably want read up on developing projects under the NET+OS development environment and using the AWS component, before continuing.

3.4 Scope

This white paper delves directly into NET+OS's AWS CGI interface. This paper discusses the following:

- NET+OS's CGI interface
- Accessing incoming data
- Including device data into web pages
- Interfacing into pages compiled by the pbuilder utility
- Interfacing into pages located on the NET+OS file system

This white paper does not discuss the following topics and thus are outside the scope of this document:

- Developing AWS applications in general (again this paper focuses on CGI-based applications)
- Developing using Digi's NET+OS embedded development environment
- Developing using Digi's NET+OS AWS environment
- Writing and debugging C code
- Writing html code (we assume you know how to develop a web page)
- Developing AJAX (asynchronous JavaScript and XML)
- Debugging tcp/ip issues
- Any security issues that might arise including cross site scripting
- Any information about Digi's Embedded Linux development environment

Again this paper assumes you have knowledge of Digi's NET+OS embedded development environment, web page development and debugging same. This paper solely focuses on CGI-development in a NET+OS AWS environment.

3.5 Theory of Operation

When developing AWS-based projects, the assumption made is that when a user surfs into your device, the AWS engine looks at the URL of the page requested and finds the object in the AWS' table of page objects that represents that URL. For each field, on the web page, the associated stub function (a stub function is a callback function generated by the pbuilder utility and is generally associated with a web page's form object) is found and called. Any data associated with the field, is removed from the key-value pair format and passed as a parameter to the associated stub function.

On the one hand, this method is all nice and clean. All you, the developer need to worry about is getting or putting your device data, to or from formal parameters of the object's stub function. The AWS engine worries about formatting the data into a web page and getting it back to the browser. On the other hand, you the developer have lost all control of the lower level processing of the web pages and the data. If you wanted to drastically change the format or contents of web pages, based on some input from the user, it may be difficult or impossible to do if the AWS engine is controlling access to pages and data.

In the CGI model (my term) you fully control the html data (html code and associated data) that is returned to the browser. In fact, you can generate this in real time, if you choose. So you could return a "boiler-plate" web page every time a browser sends in a request. Alternatively, you could look at incoming data and send back different html based on any combination of incoming data. That is up to you. So in the CGI model, you are purely using the AWS engine as a conduit between you and the browser(s).

So how does all this bolt together? In a nutshell, the key is the function RpExternalCgi, which is contained in the cgi.c file that needs to be part of your application. When the AWS engine receives a URL that it cannot find in its table of page objects, the AWS engine calls the RpExternalCgi function. Among other things, the AWS engine passes the URL of the incoming page to function RpExternalCgi. Within the RpExternalCgi function, you need to trap for each URL that your application recognizes and intends to process. The portion of RpExternalCgi that recognizes a URL is then free to return any valid, fully formed html page to the browser, through the AWS engine. The AWS engine adds http header information to your page, prior to sending it out.

4 Basics

4.1 RpExternalCgi()

As stated earlier, the primary interface between the advanced web server and your CGI code is the API RpExternalCgi(). The AWS calls the API RpExternalCgi, when the AWS gets an URL from a browser and the URL is not recognized (not contained in the AWS's table to known URLs) by AWS. API RpExternalCgi is called with two formal parameters, namely theTaskDataPtr and theCgiPtr. theTaskDataPtr is an internal data structure used by the AWS engine. Since you are working outside of the scope of the pbuilder utility, the information contained therein is unlikely to be useful to you. The parameter theCgiPtr is another matter. theCgiPtr is the method the AWS uses to pass input, obtained from the browser, to you and through which you pass data back to the browser. It might be a good idea for you to look inside the API reference guide. Search for RpExternalCgi() and then click on the live link to rpCgiPtr, the data type of theCgiPtr to allow you to get familiar with the fields of rpCgiPtr.

4.2 Most used fields of the rpCgiPtr structure

This section describes the fields contained within the rpCgiPtr with which you will be dealing most often. They are presented in no particular order. In the section of this paper that discusses the attached project, the purposes of these fields will be discussed. The rpCgiPtr structure is defined in the API reference guide. To include this structure include "`#include "http_awsapi.h"`" `in your application.`

4.2.1 fHttpRequest

This field describes the type of operation that this transaction represents. The field passed in is an enum representing a "get" a "post" or a "head".

4.2.2 fPathPtr

This field contains the URL of the request received from the browser. It is stripped of the http:// and stripped of the ip address. To fully understand the format of the information passed in this field, we recommend you print out this field in early testing phase.

4.2.3 fResponseState

When sending data back through the AWS engine, use this field (an enum) for informing the AWS engine of the state of affairs of your data as a whole. You could be all done, this could be one of more buffers coming or, there could be more data coming but you are not prepared, at this time to provide a buffer.

4.2.4 fHttpResponse

If you were using the pbuilder utility and the AWS to process web requests, the AWS engine would inform the browser of things such as "all is ok", "not authorized" or "not modified". This field allows you to return this type of information, through the AWS engine to the browser. This field is an enum.

4.2.5 fResponseBufferPtr

This field carries a pointer to a buffer containing all of the data to be returned to the browser, through the AWS engine. This includes the html code and any device data included with the html code. This pointer needs to be around after RpCgiReturns control to the AWS engine.

4.2.6 fResponseBufferLength

This field must contain the length of the contents of fResponseBufferPtr.

4.2.7 fArgumentBufferPtr

This is a buffer pointer containing the data (forms fields) provided by the browser. This data is in the "raw state" as provided by the browser. That is a set of key/value pairs. You'll need to be prepared to parse this list in order to have access to the fields. A good starting point might be to "practice" with a simple web page and a simple CGI function and print out the contents of this field.

4.2.8 fArgumentBufferLength

This field contains the length of buffer fArgumentBufferPtr.

5 Example Application Explanation

This section explains the example application that accompanies this paper. This example application demonstrates a CGI application. The application is designed and implemented as follows: An initial html page, index.htm, is accessed. This page has an anchor (link) to the first of a number of CGI-based pages. You can jump between the CGI and pbuilder-based pages. Digi_CGI_Request displays a web page containing a form with three fields to fill in and a submit button for submitting the form. Submitting the form brings up page Digi_CGI_Post. This part of the application also prints out the arguments that the user entered into the form (on submission you'll probably want to update your device with these arguments. file_system_based_page.htm demonstrates pulling of an html file from the file system and passing it back to the user's browser. file_that_does_not_exist.htm demonstrates handling an error (requested page no available). my_gigunda_page.htm demonstrates passing a large file back to the browser.

5.1 root.c

First we'll examine root.c. If you edit file root.c you'll see that it performs two main functions. First it starts up the ftp server. Second it starts up the advanced web server. The ftp server is started up to allow you to write two files to the file system. This is done as two of the CGI pages served are pulled from the file system. The ftp server is not required except as a method for placing files onto the file system.

5.2 cgi.c

The lion's share of the application is handled in this file.

Open this file with an editor and search for API RpExternalCgi(). The function is made up of five "if" statements. Each of the following sections analyzes one if statement.

5.2.1 Get Digi_CGI_Request

This first section looks for an fHttpRequest containing eRpCgiHttpGet (get request) and an fPathPtr (the URL) containing "/Digi_CGI_Request". This request was generated by clicking the link on page index.htm. If you are familiar with html, you will have noticed that the code being placed in the buffer, when accepted by the browser, displays a form containing 3 fields for a user to fill in. Additionally the form contains a submit button that, when clicked, will submit that form, to the web server on your device.

Since this is the last (and first) buffer for this web page (the browser and AWS should not expect any additional data), fResponseState is set to eRpCgiLastBuffer. Since we are aware of no errors associated with processing this page, fHttpResponse is set to eRpCgiHttpOk. A pointer to the buffer into which you placed the data for this page is placed into fResponseBufferPtr. Lastly fResponseBufferLength is set to the length of the buffer.

Since we will be true to no other if statements, we'll fall to the return statement. The AWS engine takes the appropriate fields from thsCgiPtr, assembles the data and sends the html code back to the browser that made the initial request.

5.2.2 Post Digi_CGI_Post

The next section looks at a post transaction. This would be an attempt to update information on the device. In this case we look for fHttpRequest being set to eRpCgiHttpPost and fPathPtr being set to "/Digi_CGI_Post". To see how we got here, we need to go back to the last html page that we generated. If you look at the form command, the action attribute for the form command was "Digi_CGI_Post". Thus when you hit the submit key and the page was submitted to the device, the browser then requested /Digi_CGI_Post from the device. The page congratulates you for successfully submitting a post. Additionally the page tells you that the arguments (what you entered into the form) are displayed in the dialog screen. As explained above, this is the contents of fArgumentBufferPtr. If you were going to process this posting, you'd need to parse the contents of fArgumentBufferPtr in order to extract the values for processing.

Next notice that this page contains three anchors. One points to file_system_based_page.htm, one points to this_file_does_not_exist.htm and the third points to my_gigunda_page.htm. We'll look into these three pages in the next three sections.

5.2.3 Get file_system_based_page

This is the first of two pages that do not represent html code contained in the c code of API RpExternalCgi() but instead represent the reading of data from the file system and using CGI for shipping that data back to the browser, via the AWS engine. As in the last two sections, we trap for this page by looking for an fHttpRequest containing eRpCgiHttpGet and an fPathPtr containing "/file_system_based_page.htm". If found the code attempts to open and then read file_system_based_page.htm from the file system. In this case, the file is assumed to be on the FLASH file system. The file could have just as well been in the RAM file system. If the file was on the RAM file system, then the file opened would be "RAM0/file_system_based_page.htm". Since we "know" that this file will fit into one buffer, the status fields in the rpCgiPtr are set up to reflect this. In a follow-on section, we'll look at a scenario, where this is not the case.

5.2.4 Get file_that_does_not_exist.htm

We wanted to ensure that we included a page that demonstrated an access failure. This section shows how to handle a file read or file open error.

We trap for a get and for file_that_does_not_exist.htm. This is a file that I know does not exist on the FLASH of my device. In this case, please notice that fHttpResponse is set to eRpCgiHttpNotFound. If you ran the application and surfed to this page (an anchor on the post page) you will get the error page "The web page cannot be found" and "HTTP 404". This page was caused to be returned due to the eRpCgiHttpNotfound error assigned to fHttpResponse. You'll want to look up the other available responses in case others are appropriate for your application. They are described in the API reference guide.

5.2.5 Get my_gigunda_page.htm

In this section we look at another page, which reads a file from the file system, but in this case, the file will not fit into one buffer. Thus multiple reads/send pairs must be performed, in order to get the entire file back to the browser.

As in the last section, we first trap for a request for this page, by looking for an fHttpRequest set to eRpCgiHttpGet and an fPathPtr containing "/my_gigunda_page.htm". If we successfully open the file, we start reading the file. Now the read could fail, we could hit end of file or we could read a portion of the file successfully. If the read fails, we set last buffer, set the buffer to NULL, set buffer length to 0. The AWS engine will discontinue asking for file data. If we get to end of file (bytes read is 0). We set last buffer, set buffer pointer to NULL, set response to not found and set buffer length to 0. If the last read is successful and this is not end of file, fResponseState is set to buffer complete (this buffer is full but expect more), response is set to ok, a pointer to the filled buffer is assigned to fResponseBufferPtr and the buffer length is set to the length of the buffer.

5.3 index.htm

For all sample projects developed under NET+OS, we use index.htm as the default page for accessing the web server. I chose to continue with that model. Thus when running this sample project and accessing it from a browser, if you surf to the device, using URL http://<ip address of the device>, you'll get to page index.htm. This page has a live link to take you to the first of the pages in the sample that are served via CGI. So this page is the only one in the sample that was processed via the pbuilder utility. There is no requirement that you include a pbuilder-utility-based page in your CGI-based application. Instead, including this, demonstrates that you can combine CGI and pbuilder-derived pages in one application.

6 Advantages and Disadvantages of Developing with CGI

If you choose to develop your web-based application using CGI, clearly, you have much more control over the minute-by-minute processing of web pages as they enter your device. You could parse and look at forms objects, entered by the user and sent from the browser and decide to generate vastly different web pages based on those objects. You can pull files out of the device's file system and send them back to the browser. Clearly you are in full control and you have ultimate flexibility.

The downside is that you do all of the "heavy lifting". You must parse the data (forms objects) presented by the browser. You must merge the device data and the html code in response to requests. Approximately 85% of the work that the AWS does for you, your application must perform. In addition, as you add more web pages to your application, you must add then to the RpExternalCgi function.

Whether you use the pbuilder utility vs CGI is up to you. You'll need to decide how much control your application requires vs how much of the html work you and your application are willing to do.

7 Conclusion

Most NET+OS customers developing web-based applications will find the pbuilder utility-based method of web page compilation adequate for their needs. They'd prefer to give up ultimate control in return for the AWS engine doing the lion's share of the heavy lifting". On the other hand, other customer will want more control over their web pages or from prior development, are more comfortable developing web applications through CGI. For those customers who require this level of control, NET+OS's AWS provides a CGI interface. Hopefully through this white paper and the associated project, we have provided you with a starting point and a greater understanding of CGI development under NET+OS's AWS component.

8 Appendix

8.1 Glossary of terms

- AJAX – Asynchronous Javascript and XML
- AWS – Advanced Web Server – Digi's NET+OS development environment component for adding web pages to an embedded project
- CGI – Common Gateway Interface – a method for creating a web component to a project where the html code is generated by C code at run time.
- NET+LX – Digi's Linux-based development environment
- NET+OS – Digi's embedded development environment based on the Threadx microkernel
- Pbuilder utility – A component of AWS that converts html into C code for inclusion into a NET+OS embedded project

9 Citations

Hamilton, Jacqueline. "CGI010.com Learn CGI". Jacqueline Hamilton 2004. <http://www.cgi101.com/book/intro.html>