Generating HTML content on the fly using Digi International's Advanced Web Server

1 Document History

Date	Initials	Change Description
1/17/08	JZW	Initial entry
1/18/08	JZW	First round of grammar/spelling corrections
1/21/08	JZW	Add in edits

2 Table of Contents

3 Introduction

3.1 Problem Solved

You are creating a web-based application using Digi International's Advanced Web Server (AWS). You realize that due to the complex matrix of users of your system, you might need hundreds of different web pages, to meet the needs of your potential users. Further you realize that most of these pages are variations on a common theme with some differences. Is there a way to dynamically generate the HTML content (on-the-fly)? The answer is yes. This white paper describes a method for dynamically generating HTML content and at execution time allowing your application to access these pages.

3.2 Audience

This paper is intended for sophisticated technical users who understand and have experience creating applications using Digi International's NET+OS (gnu or Green Hills) and the AWS component of NET+OS. The user either wants to modify an existing AWS application allowing it to dynamically generate HTML content or the user is creating a new AWS application and again, wants the application to be capable of dynamically generating HTML content.

3.3 Assumptions

This white paper assumes that the reader has extensive experience using Digi International's NET+OS, Digi International's AWS, NET+OS's file system and standard C file system function calls. If you are weak in any of these areas, you might want to do a little studying in your area(s) of weakness before proceeding with reading this white paper.

3.4 Scope

This document describes the HTML, AWS comment tags, PBuilder utility use and stub function generation in the building of an AWS application capable of generating HTML pages dynamically. That is the content of the HTML pages is decided at run time not compile time. The following are non-goals of this white paper:

- Teach the user HTML page authoring
- Teach the user how to add AWS comment tags to an HTML page
- Teach the user how to build applications using Digi International's NET+OS (gnu or Green Hills)
- Provide any information relating to Digi International's .net product
- Provide any information relating to Digi International's LxNETES product

3.5 Theory of Operation

One or more web pages are served from the user's device and displayed in the user's browser. One or more of these browser-displayed pages allows the user to enter data and/or select options. Based on these entries and selections, variables are updated. Based on the state and or content of these variables, HTML code is generated, and written to one or more files on NET+OS's file system (in the device). Further, based on the customer's selections, the currently displayed page is changed from the current page to one of the page(s) previously written to the device's file system. Ultimately, one (or more) of the dynamically-generated pages gives the user the option of returning to his/her starting point (web page).

3.6 Conventions

Advanced Web Server (AWS) comment tags have the following format:
<!—Tag Name →
Unfortunately, MS Word makes some invalid changes to the comment tags. Thus I'll explain what they should be. A comment tag is made up of a left angle bracket '<' followed by an exclamation mark '!' and two consecutive minus signs. There should be no spaces between the angle bracket, the exclamation mark and the minus signs. Next

there should be a space. Following the space you place the tag name and options required for the tag. After the tag name and options, leave one space, followed by two minus signs and a right angle bracket '>'. Again there should be no space between the two minus signs and the right angle bracket.

4 Basics

The following section describes the basics of creating a NET+OS project as described above.

4.1 HTML code

My modus operandi in creating AWS applications and their associated web pages is to create the web pages in raw HTML code, without regard to the requirements of the AWS. In the next section, I discuss what you need to add for supporting AWS. Thus your basic web page might look something like this:

```
<html>
<head>
<title>My test page</title>
</head>
<body>
Content line one
Content line two
Content line three
</body>
</html>
```

You should then test your pages and their flow using your favorite browser. Now clearly you can not test the dynamic part, at this step. This will only test your static content. You should get your pages in a good state before continuing.

4.2 AWS Comment Tags

AWS comment tags are the method for getting the browser, the AWS and your application all talking together. Comment tags in conjunction with the stub functions (described later in this document) give the AWS access to device data. If you have not already done so, this would be a good time to review the document entitled Digi Advanced Web Server Toolkit. This document describes the basic operation of the AWS, the PBuilder utility and describes the array of comment tags available to your application.

An AWS comment tag has the following form:

```
<!— comment tag →
<!—RpEnd →
```

The simplest comment tag is one used in conjunction with static HTML data, as follows

```
<html>
<head>
<title>My Little Test Page</title>
```

```
</head>
<body>
<!—RpDZT →
This text will be displayed on your browser screen
<!—RpEnd →
</body>
</html>
```

The more complicated comment tags allow you to define stub functions that will either get device data and pass it to the AWS for displaying the data on your browser, or will take data from the browser and hand it to a stub function for storing device data or for directing the device to engage in some activity. Description of these functions is outside the scope of this document. Again, please review the Digi Advanced Web Server Toolkit for that information.

4.3 PBuilder Utility

Ultimately, your HTML code or some component(s) of it needs to be converted into C code, allowing it to run on your device. The purpose of the PBuilder utility (<NET+OS directory top\bin\PBuilder.exe) is to perform this transformation. The file list.bat, (or pbuilder.pbb under Digi ESP) described in the next section, facilitates this operation. If you are unfamiliar with the PBuilder utility, please review the Digi Advanced Web Server Toolkit document.

Among other files, running the PBuilder utility generates a .c file for each web page you created and one _v.c file containing your stub functions (AWS callbacks). If you populate your stub functions and then run the PBuilder utility again, you will lose your work. For that reason, we recommend using the following directory structure for your project:

```
Main directory
  |\
  | \
  |  \
  |   \
  |    \
  |     \
          32b
pbuilder
   |
   |
  html
```

Place your htm(l) files in the html directory. Run the PBuilder utility from the pbuilder directory. When PBuilder runs successfully it will create <html file name>.c files and <the first html file name mentioned in list.bat>_v.c file in the html directory. Copy the _c.v file into the pbuilder directory (one level up). Start filling in the stub functions. When/if you must run the PBuilder utility again, only the _v.c file with empty functions will be overwritten. Copy any new functions created in _v.c (in the pbuilder/html directory) and paste them into the _v.c file located in the pbuilder directory. The <html page>.c files can safely be overwritten as you do not normally edit these files.

4.3.1 list.bat

The list.bat file (AKA pbuilder.pbb if you are using Digi ESP) is normally placed in the pbuilder directory, mentioned above. The file lists the files on which the PBuilder utility is to operate. Since this file is located in the pbuilder directory, the paths of files described in the list.bat file, must be relative to the pbuilder directory. So the contents of the list.bat file might look something like this:

```
html/file1.htm
html/file2.htm
html/file3.htm
```

The contents of this file are text so the file is editable with your favorite text editor. Please note that the last line must have a carriage return after it. Also note that the PBuilder and the AWS always assume that the first file described in the list.bat file is to be the “main” page of your web application. So ensure that whatever file you want as the main page is listed first in list.bat.

4.4 Stub Functions

What I refer to as stub functions are the AWS callback functions created by running the PBuilder utility against your AWS comment tag-laden HTML files. After their initial creation, the functions do nothing. Your job is to fill in the stub functions with whatever action(s) you need the functions to perform. Please consult the NET+OS project, associated with this document for ideas on what actions your functions might perform. But this is highly customer dependant.

4.4.1 Variables

It is critical to remember that your stub functions are called from the AWS. In addition, there is no guarantee in which order the functions will be called, or whether you’ll be swapped out in between the return function of your stub function (when your function stack becomes invalid) and where in the AWS the variable you are returning is actually used. Therefore, you can not return stack variables from your functions. You must either define them as static or declare them globally. In addition, remember that in most cases, more than one user will access the AWS application. Therefore you will need to architect a method for keeping the use of one user’s variable from colliding with that of another user.

5 Generating the HTML

This section describes the C code required in a stub function for generating the on-the-fly web page.

5.1 BSP and the file system

Before starting you must ensure that the file system exists so your application can access it. The easiest way to do this is to allow the BSP to create the file system for you. The way you do this is by finding the bsp-owned manifest constant entitled BSP_INCLUDE_FILESYSTEM_FOR_CLIBRARY. In NET+OS V7.2 and below, this was contained in src\bsp\platforms\<your platform>bsp.h. If you are running in the Digi ESP environment, this file is contained in your workspace. In NET+OS versions above V7.2, this constant is declared in file *myplatform\bsp_fs.h* . Next change the declaration of this constant to TRUE. Next rebuild your bsp and then your application.

5.2 File creation

Ultimately there must be a user-filled-in function that creates the file(s) required for the application. The following is an example of such a function:

```
extern void generateAndJumpPages(void *theTaskDataPtr, Signed16Ptr
theIndexValuesPtr);
void generateAndJumpPages(void *theTaskDataPtr, Signed16Ptr theIndexValuesPtr) {

   FILE *fp = NULL;
         void * theServerDataStructure = NULL;

   fp = fopen("RAM0/theNewPage.htm","w+");
         if(fp == NULL)
         {
            printf("Unable to open file\n");
         }
         printf("file opened\n");

   fprintf(fp,"%s\n", "<html>");
   fprintf(fp,"%s\n", "<head>");
   fprintf(fp,"%s\n", "<title> New test web page </title>");
   fprintf(fp,"%s\n", "</head>");
   fprintf(fp,"%s\n", "<body>");
         fprintf(fp,"%s\n", "You are accessing a web page created on the fly");
         fprintf(fp,"%s\n", "<br>");
   switch(thePageToGenerate)
         {
            case 3:
            fprintf(fp,"%s\n", "You selected page 3");
            fprintf(fp,"%s\n", "<br>");
                  break;
                  case 4:
            fprintf(fp,"%s\n", "You selected page 4");
            fprintf(fp,"%s\n", "<br>");
                  break;
                  case 5:
```

```
            fprintf(fp,"%s\n", "You selected page 5");
            fprintf(fp,"%s\n", "<br>");
                  break;
                  default:
            fprintf(fp,"%s\n", "You selected an unknown page");
            fprintf(fp,"%s\n", "<br>");
                  break;
         }
   fprintf(fp,"%s\n", "There should be a link on this page to get you back to the first
page");
         fprintf(fp,"%s\n", "<br>");
         fprintf(fp, "%s %s %s\n", "You selected animal ", theAnimalName, " on the
selection page");
         fprintf(fp,"%s\n", "<br>");
   fprintf(fp,"%s\n", "</body>");
   fprintf(fp,"%s\n", "<a href=/jzwPageOne.htm> Return to the first page</a>");
   fprintf(fp,"%s\n", "</html>");
         fclose(fp);

   // get the AWS data structure
         theServerDataStructure = RpHSGetServerData();
         if(theServerDataStructure == NULL)
         {
            printf("AWS data structure not available\n");
                  return;
         }
         // allow file close to finish
         tx_thread_sleep(10);
         RpSetNextFilePage(theServerDataStructure, "/FS/RAM0/theNewPage.htm");

         return;
}
```

The call to fopen() opens a file in the device's file system. If you were opening multiple files, you'd need multiple calls to fopen().

As you can see, the calls to fprintf() are writing the HTML code into the file. Please notice the switch statement. This is where I generate different page content based on input from the user's entry via their browser. Also notice in the following line:

```
         fprintf(fp, "%s %s %s\n", "You selected animal ", theAnimalName, " on the
selection page");
```

The animal name changes as the user enters different animal names again though his/her browser.

Clearly this is a simple example but it shows that based on user input, I can generate different page content that will be displayed to the user in his/her browser.

When you are done writing all your content into the file please remember to close the file(s). fclose(fp);

Last, please notice the following call as it is subtle but critical:

```
        RpSetNextFilePage(theServerDataStructure, "/FS/RAM0/theNewPage.htm");
```

This call instructs the AWS to change the next page to be displayed to the user. Additionally it tells the AWS to find the web content for this page in the file system, as opposed to the AWS's page table. This is the jumping off point from the pages that were built into your AWS application to the dynamic pages that your application generated (and placed in the file system). In addition please notice the following fprintf() statement:

```
   fprintf(fp,"%s\n", "<a href=/jzwPageOne.htm> Return to the first page</a>");
```

By including a line like this, somewhere on your page, you leave your dynamic page the ability to link back to a page contained in the AWS's internal table.

5.3 File request format

The format of the file and page references is quite critical to ensuring that your AWS application runs successfully. Please notice the following call to RpSetNextFilePage():

```
        RpSetNextFilePage(theServerDataStructure, "/FS/RAM0/theNewPage.htm");
```

The format of the file reference is /FS/RAM0/theNewPage.htm. /FS tells AWS that the page is on the file system. /RAM0 tells the AWS that the file is in the RAM file system, device RAM0. If the file you wanted was in the FLASH file system, you'd replace RAM0 with FLASH0. Last /theNewPage.htm is the file name in the file RAM system.

Next notice the following:

```
   fprintf(fp,"%s\n", "<a href=/jzwPageOne.htm> Return to the first page</a>");
```

First notice that I have left out the /FS and the /RAM0. this says that the page is stored in the AWS's internal page table. Also notice that I start the file name with a forward slash "/". If you reference a page name from within a file on the file system, AWS assumes that the page to which you are referring to is also in the file system. So you'd get a file not found error. Thus I'd recommend that you include the full path reference when referencing web pages.

6 AWS to stub function hand-off

A crucial question whose answer might lead to a successful AWS application is how do I structure my AWS comment tags so that when my user hits the submit button, I get to a function that will create my files and send me to the correct web page? Please review the following web code, including AWS comment tags:

```
<html>
<head>
<title>Digi test application, page one</title>
</head>
<body>
<!-- RpDZT -->
This is the Digi test application for generating web pages on the fly
This is page two
<!-- RpEnd -->
<!-- RpFormHeader RpFunctionPtr="generateAndJumpPages"
   RpNextPage=PgjzwSubmittedPage -->
<form method=POST action="jzwSubmittedPage.htm">
<!-- RpEnd -->
Please select a page and it will be generated:
<br>
<!-- RpDZT -->
Page 3
<!-- RpEnd -->
<!-- RpFormInput TYPE=radio NAME=onTheFlyRadioButtons VALUE="three"
RpItemNumber=3
    RpSetType=Function RpSetPtr=setThePageToGenerate RpGetType=Function
          RpGetPtr=getThePageToGenerate -->
<input type=radio name=jzwePageThree value="3"> Page3
<!-- RpEnd -->
<br>
<!-- RpDZT -->
Page 4
<!-- RpEnd -->
<!-- RpFormInput TYPE=radio NAME=onTheFlyRadioButtons VALUE="four"
RpItemNumber=4
    RpSetType=Function RpSetPtr=setThePageToGenerate RpGetType=Function
          RpGetPtr=getThePageToGenerate -->
<input type=radio name=jzwePageFour value="4"> Page4
<!-- RpEnd -->
<br>
<!-- RpDZT -->
Page 5
<!-- RpEnd -->
<!-- RpFormInput TYPE=radio NAME=onTheFlyRadioButtons VALUE="five"
RpItemNumber=5
    RpSetType=Function RpSetPtr=setThePageToGenerate RpGetType=Function
          RpGetPtr=getThePageToGenerate -->
<input type=radio name=jzwePageFive value="5"> Page5
```

```
<!-- RpEnd -->
<br>
<p>
<!-- RpDZT -->
Please select your favorite animal:
<!-- RpEnd -->
<br>
<!-- RpFormInput TYPE="text" NAME="animalName" SIZE="32"
MAXLENGTH="32"
  RpGetType=Function RpGetPtr=getTheAnimalName
  RpSetType=Function RpSetPtr=setTheAnimalName -->
<input type=text name=animalSelected size=32 maxelength=32>
<!-- RpEnd -->
<p>
<!-- RpFormInput TYPE=SUBMIT VALUE="Submit the puppy" -->
<input type=submit>Go
<!-- RpEnd -->
<!-- RpFormEnd -->
</form>
<!-- RpEnd -->
</body>
</html>
```

In the following line:

```
<!-- RpFormHeader RpFunctionPtr="generateAndJumpPages"
   RpNextPage=PgjzwSubmittedPage -->
```

The comment tag RpFunctionPtr refers to function generateAndJumpPages(). If you remember from the section entitled File Creation, generateAndJumpPages() is the function that generates the file and changes the "next page". So by setting RpFunctionPtr to a function that creates files and sets the next page, the PBuilder utility creates a stub function that you fill in with the appropriate code. The AWS will call into this function after your user hits the submit key on the appropriate web page. Further the RpForInput tags, show the code required to implement the radio buttons, the text input and the submit key. Again, please consult the Digi Advanced Web Server Toolkit document for more information on AWS comment tags.

7 Caveats

The following section briefly describes areas to be careful of when creating AWS applications that access dynamic pages.

7.1 Multiple users

7.1.1 Files

In most cases your AWS application will service multiple users. You want to ensure that the input of one user does not corrupt the inputs of another. Therefore you must architect your application (stub functions) to ensure that users get a unique file. The Digi Advanced Web Server Toolkit document describes calls to access your connection number. You could generate files names that contain the user's connection number as a way of keeping file names from colliding. You'll need to think about this when creating your application.

7.1.2 Variables

The variables that the stub functions use to store and retrieve device data are another place where the sharing of variables between users could pose a problem. You will also want to architect something in your application to ensure that one user's data does not corrupt another user's data. Again you might use the connection number as an index into a table of global variables as a way of eliminating corruption. This is clearly something that you must think about when architecting your application.

8 Conclusion

Your application might not be as complicated as needing the ability to create dynamic web pages. But if you do need this capability, this document demonstrates the methods required to implement them. If you are unfamiliar with AWS comment tags, I urge you to review the document entitled Digi Advanced Web Server Toolkit in your Documentation directory.

Example Sources