Developing AWS-based Applications Under NET+OS

A white paper and project

Table of Contents

1 Introduction

Many developers now use web access as the primary method for interrogating their embedded devices. The Advanced Web Server is the preferred web server for embedded devices based on NET+OS. There are some customers, though, that find the AWS somewhat intimidating. It should not be. The AWS is a powerful web server with many features to automate web-based applications. Like anything else, first getting started can be the most daunting step. This paper and the application it describes represent an attempt to help customers get over that step.

NET+OS (currently) ships two web server offerings. The first is called the Basic Web Server (BWS), while the second is called the Advanced Web Server (AWS). BWS is geared more for simple web applications. That said, any application (that we know of) that can be developed using the BWS can also be developed using AWS. Digi generally recommends that customers use AWS as at some point we hope to end of life (EOL) BWS. But for the foreseeable future, Digi International ships NET+OS with both web servers.

This paper and its associated application are geared exclusively for the AWS. Said another way, if you are looking for information about BWS in this document, please close this document immediately as you will be disappointed.

1.1 Purpose

This white paper describes a large web application to be made freely available to customers and field personnel. They (the paper and the application) were developed as a teaching tool to help customers learn and develop the skills and knowledge necessary to successfully develop embedded device applications that contain the AWS. The application contains many of the AWS comment tags, including repeat groups, forms, tables and java applets. The intent is that a customer could use this as a jumping off point for developing their own AWS-based application. Additionally, I have included an example of including a java applet and a javascript program within an AWS application, as we have received a number of questions from customers on how to include these in such applications.

1.2 Intended Audience

This paper is intended for anyone needing some help understanding either existing AWS-based embedded applications or in developing an AWS-based application from scratch. The presumption is that the user has some knowledge of developing NET+OS-based embedded applications and maybe has a little experience with simple embedded web applications but now wants to “take the deep dive”. Additionally, the presumption is that the user has knowledge of C programming, network programming and html programming.

1.3 Scope

This paper and the application it describes should be applicable to all versions of NET+OS that incorporate the TRECK tcp/ip stack and its associated Internet Address Manager (IAM) (NET+OS V7.0 and later). This application could be ported back to prior versions of NET+OS but Digi International has no plans for doing so at this time. A customer could do so using their own time and expense.

This paper and its associated application do not teach any of the following:

- C programming

- Tcp/ip networking
- Socket programming
- html programming
- Basic Web Server (BWS)
- NET+OS development
- Java programming
- Creating java .jar files
- Javascript programming

Should you need help with any of these subjects, there are plenty of good books available on these subjects.

1.4 Document Conventions

1.5 Edit History

Date	Initials	Change Description	
1/1108	JZW	Initial entry	
1/23/08	JZW	Start filling in sections	
1/25/08	JZW	First round of my edits	
1/29/08	JZW	Include Bubba's comments	
1/29/08	JZW	Add appendix section on migrating to ESP	

2 AWS Comment Tag Demonstrations

In AWS comment tags are the method for giving a web page access to device variables. They are added to your html code, generally after you are happy with your general layout. We generally recommend the following set of steps in developing your web application:

- Develop your html pages without the comment tags
- Test your html pages using your browser of choice
- Add the comment tags
- Run the html pages through the PBuilder utility
- Flesh out the stub functions

The format of a comment tag is as follows:
<!-- Tag options -->

The symbols are left angle bracket, an exclamation mark, two minus signs (dashes) a space, the actual tag, a space, the tag's options (space separated) a final space, two minus signs (dashes) and a right angle bracket. Every comment tag must be terminated with an ending comment tag containing the RpEnd tag (see the example below).The following represent a few examples of correct comment tags:

- <!-- RpDataZeroTerminated -->
- <!-- RpEnd -->

- <!-- RpDisplayText RpGetType=Function RpGetPt=getTheData RpTextType=ASCII Size=10 -->

The first tag is used to surround plain static text. The second tag terminates all tags. The third tag would be used to display device data on a web page.

If you are unfamiliar with AWS and comment tags you might want to read the document entitled Digi Advanced Web Server Toolkit. It is contained in the Documents directory of your kit tree.

2.1 General Device Data Access

This section describes comment tags used to display data to which you are not giving the user the ability to modify. Most of the pages included in the attached AWS application are of this type. The following simplified html file might be used to display some device data onto a web page through a web browser:

```
<html>
<head>
<title>Test title page</title>
</head>
<body>
<!-- RpDataZeroTerminated -->
The device's temperature is currently:
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Function RpGetPt=getTheDevicesTemp RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
</body>
</html>
```

The AWS calls the function getTheDevicesTemp(), which returns to the AWS the current temp value. AWS passes this value to the browser for display.

2.2 Using Repeat Groups

Suppose you have some table entries, table rows, characters within a row or other web output that repeats. That is the device data might be different but the surrounding html code is the same. AWS allows you to place the html code and the comment tags within some special tags called repeat groups. These allow some amount of html and tags to be repeatedly called until some terminating event happens. Repeat groups come in a couple of varieties. In the attached example application I make use of two types namely, RpRepeatGroup allows you to emulate a for loop in C. That is you know the number of times you will go through the loop at compile time. RpRepeatGroupWhile repeatedly executes code (the code being a function that you define) until your code tells it to stop. In a later section, I'll explain this in greater detail.

Let's assume you have a group of four interfaces, on your device and you'd like a web page that displays some information about the four interfaces. Now one way to implement this would be to write four separate sections of html and comment tag code that implements the four sections. An example might be as follows:

```
<html>
```

```
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<!-- RpDataZeroterminated -->
Device interface information
<!-- RpEnd -->
<!-- group 1 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices1Name RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices1Speed RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices1Protocol
RpTextType=ASCII Size=10 -->
<!-- RpEnd -->
<!-- group 2 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices2Name RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices2Speed RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
```

```
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices2Protocol
RpTextType=ASCII Size=10 -->
<!-- RpEnd -->
<!-- group 3 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices3Name RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices3Speed RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices3Protocol
RpTextType=ASCII Size=10 -->
<!-- RpEnd -->
<!-- group 4 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices4Name RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices4Speed RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevices4Protocol
RpTextType=ASCII Size=10 -->
<!-- RpEnd -->
</body>
</html>
```

Now using repeat groups, the code would look like the following:

```
<html>
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<!-- RpDataZeroterminated -->
Device interface information
<!-- RpEnd -->
<!-- RpRepeatGroup RpStart=1 RpLimit=4 RpIncrement=1 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesName RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesSpeed RpTextType=ASCII
Size=10 -->
<!-- RpEnd -->
<!-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesProtocol
RpTextType=ASCII Size=10 -->
<!-- RpEnd -->
 <!-- RpLastItemInGroup -->
</body>
</html>
```

This code creates a loop starting at 1 and going to 4 and incrementing by 1. Each time a row of titles is displayed. Next a row of device data is displayed. How this gets converted to C code and how the index is passed, will be discussed later in the document. One thing to notice, if your data acquisition comment tag is not in a Repeat group, leave the RpGetType equal to Function. If it is in a repeat group, then set the RpGetType to Complex. Complex will assure that an index will be passed into the stub function.

Now suppose you did not know how many interfaces your device had because your code runs on different types of devices, so you do not know how many interfaces there are until you are executing. The above code could be modified to handle this as follows:

```
 <html>
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<!—RpDataZeroterminated -->
Device interface information
<!-- RpEnd -->
<!-- RpRepeatGroupWhile RpfunctionPtr=areWeDoneYet RpMaxItems=20 -->
<!-- RpDataZeroTerminated -->
Interface Name
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface speed
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
Interface protocol
<!-- RpEnd -->
<!-- RpDataZeroTerminated -->
<br>
<!-- RpEnd -->
<-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesName RpTextType=ASCII
Size=10 -->
<-- RpEnd -->
<-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesSpeed RpTextType=ASCII
Size=10 -->
<-- RpEnd -->
<-- RpDisplayText RpGetType=Complex RpGetPt=getTheDevicesProtocol RpTextType=ASCII
Size=10 -->
<-- RpEnd -->
 <-- RpLastItemInGroup -->
</body>
</html>
```

2.3 Including Images

Many companies would like to have a corporate logo on all web pages served by a device, so that when users access that device, they are reminded of the sponsoring company. The logo could be included in the application or it could be accessed from the devices file system (assuming the application includes a file system). To include a corporate logo, you might use the following code:

```
<html>
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<img src=”Images/myCompanyLogo.jpg” align=right >
```

```
</body>
</html>
```

If the image file were located in your device's file system, you could modify the code as follows:

```
<html>
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<img src="/FS/RAM0/Images/myCompanyLogo.jpg" align=right >
</body>
</html>
```

For the RAM-based file system. If the image were located in FLASH, you could do the following:

```
<html>
<head>
<title>Test page for repeat groups</title>
</head>
<body>
<img src="/FS/FLASH0/Images/myCompanyLogo.jpg" align=right >
</body>
</html>
```

2.4 Hidden fields

If your web page were made up of many device data fields, when that page is displayed, each field is accessed and then displayed. There is a good chance that those fields are accessed through some helper functions and might be contained in some sort of data structure. So for each field, your application could call the helper function, pull the data et all. The problem is that this would be quite resource inefficient. It might be easier to access the data once, store it in some global structure and then have each field display access the structure. The question is how do you set up the web page such that a function is called to access the device data, before the fields get displayed? You do this with a hidden field within a form. A hidden field is an html device for performing some task but not showing anything to the user, at the time that that field is accessed by the browser.

So I want a page that accesses some complicated device data and prepares it for display on the remainder of the page. You might do the following:

```
<html>
<head>
<title>Test page for hidden fields</title>
</head>
```

```
<body>
<!-- RpFormHeader method=get  -->
<form>
<!-- RpFormInput  type=hidden name=theDeviceStructureStuff
     RpGetType=Function RpGetPtr=getTheDeviceStructureStuff -->
<input type=hidden name= theDeviceStructureStuff value=1 >
<-- RpEnd -->
<!--  RpformEnd  -->
</form>
<-- RpEnd -->
.
.
.
Code that displays the fields
.
.
.
</body>
</html>
```

The code from the section entitled general device access might replace the "code that displays fields" comment in the above html code. The main thing to take away from this section is that when ever this page is accessed, the function called getTheDeviceStructureStuff will be executed. Your application dependant code you execute and access the structure containing the device data for display.

3 Building AWS Projects

The previous section discussed some of the more interesting AWS comment tags that you will encounter when looking through the included application. This section briefly discusses the mechanics of creating an AWS project.

3.1 Design the Web Screens

The first step is to design your web screens. If you are going to include device data, then you'll need to understand what data is available to you and the methods (functions) available for accessing that data. It is possible that there are no functions and instead, the data is directory accessible. You'll want to note that also. But the point is as in any other coding effort, design your screens.

3.2 Write the HTML

What I like to do next is to write your screens in html. That is pure html with no AWS comment tags. I do this so that I can get a sense of what the screens will look like. Also I can check the links between the screens so that I know that the links make sense. If I have a screen from which I can't get back because I forgot a link, now is the time to test that. Test your pages using different web browsers (MS IE, Monzilla firefox, Netscape Navigator). You might even try different version of the browsers. Make sure that the look you want is consistent. If not fiddle with the html code until it is consistent. This is a good time to test it before you need to download the code into your device. It is fairly easy to place all your pages in a directory on your PC and access them with your browser.

3.3 Add the AWS Comment Tags to the HTML Code

Now that you have your web pages looking as you like, add your AWS comment tags to your html pages. Remember that almost all AWS comment tags require a terminating RpEnd comment tag. The only exception is RpLastItemInGroup. Generally speaking you surround your html code with the AWS comment tags. Place the functional comment tag on top of the html code and the RpEnd tag under the html code. By using this method, you can continue to test your plain html code on your PC (in case you change things) and then compile, download and test on your device.

You should stick to good coding conventions, remembering that the code will be converted into C code. So give all functions meaningful names so that you can cross reference function calls to web pages, in case debugging is needed.

3.4 Edit the PBuilder Project Description File

You now have working html code and what you believe is reasonable AWS comment tags surrounding your html code in your html files. Now it is time to convert your html code into C code. To do this a description file is needed. In older versions of NET+OS, this file was called list.bat. In newer versions of NET+OS this file is called pbuilder.pbb. Either way, it is a text file describing the files that require conversion.

One important caveat is that the first file listed in the description file is assumed to be the "home" page in your device. So when including the files in the description file, keep track of which is to be the device's home page.

We have developed a convention that the directory tree of the application, for an AWS application, contains a pbuilder directory and that the pbuilder directory contains an html directory. We place the html files (containing the comment tags) in the pbuilder\html directory. The description file is located in the pbuilder directory. Digi recommends that customers adopt this convention. Additionally, java applets and image files are also placed in the pbuilder\html directory.

The description file needs to contain listings of ALL files to be converted into C, this includes html files, images and java applets. If a page, image or java applet is to be contained in the devices file system instead of being "built in" then that file should not be listed in the description file.

3.5 Run Pbuilder

Your html files are all written. Your comment tags have been added to your html files. It is now time to convert your html and comment tags into C source code. You do this using the Pbuilder utility. The Pbuilder utility is located in the \bin directory of your NET+OS kit tree.

At this point it is important to point out Pbuilder's actions. When you execute the Pbuilder utility, it will create a .c file and a _v.c file. Now, Pbuilder can create one of these files for each html file or it can create one .c and one _v.c file containing the C source code for ALL html files. This is purely a personal decision. In the application attached to this paper, I chose to create one .c and one _v.c file that would include the C code for ALL of the html files. Again this is a personal decision.

Let’s say you have an html file entitled “manage_tempurature.html”. The .c file created by Pbuilder would be “manage_tempurature.c”. The _v.c file would be entitled “manage_tempurature_v.c”.

Let’s say I have 3 html files entitled “front_page.html”, manage_tempurature.html and “manage_speed.html”. Additionally, let’s assume that front_page.html is listed first in the list.bat or pbuilder.pbb file. If I have Pbuilder create a single file, the files created would be entitled “front_page.c” and “front_page_v.c”. front_page.c contains all the data structures needed by AWS to display the web pages while front_page_v.c contains the stub functions that you’ll fill in, giving AWS access to device variables. On the other hand if you decide to have the Pbuilder utility create multiple files, then after running Pbuilder, you should see the following files:

- front_page.c
- front_page_v.c
- manage_tempurature.c
- manage_tempurature_v.c
- manage_speed.c
- manage_speed_v.c

Generally, you do not edit the .c file(s). You do, though edit the _v.c files as these contain the stub functions. What you need to understand is that every time you run the Pbuilder utility, new _v.c file(s) will be created. Thus you could destroy the work you did filling in the stub functions, by rerunning the Pbuilder utility. To alleviate this, we recommend the following. If you look at the example application entitled nahttp_pd, you will notice that under the main directory, there is a pbuilder directory. Under the pbuilder directory, there is an html directory. Place your .html files in the \pbuilder\html directory. Place the list.bat or pbuilder.pbb file in the \pbuilder directory. When you list the .html files in the list.bat or pbuilder.pbb file, list them relative to the pbuilder directory. Thus using the example above, your list.bat or pbuilder.pbb file would like the following:

```
html\front_page.html
html\manage_tempurature.html
html\manage_speed.html
```

The Pbuilder utility runs from the DOS prompt. Thus you must set your current directory when in the DOS prompt. Set your current directory to the pbuilder directory. Run the Pbuilder utility from this directory. Since the list.bat or pbuilder.pbb files lists the .html files as being in the \html directory, it will place the .c and _v.c files also in the \html directory. After running the Pbuilder utility for the first time, copy the _v.c file(s) only into your pbuilder directory for the application. When you rerun the Pbuilder utility, just copy the stub functions that are added to the copy of _v.c in the \pbuilder\html directory into the copy of _v.c located in \pbuilder. This ensures that you do not lose the work that you previously did.

3.6 Identify the Device Data and Their Locations

You know your device and what device data is directly available to your application. Additionally review the API reference guide for calls that make other device data available to your application. This will get you ready for your next steps.

3.7 Fill in the Stub Functions

I recommend filling in the stub functions in two steps. But first I will digress. Please remember that we are now talking about editing the functions in the _v.c file(s) located in the \pbuilder directory. Do not modify the copy of the _v.c file(s) in the \pbuilder\html directory as that will be modified every time you run the Pbuilder utility.

As a first pass, we recommend that instead of actually writing the stub functions, that you just place return function calls in the stub functions. Thus suppose you have a stub function called getThePhysicalAddress. Change the existing return call to something like return "00:40:9D:BA:DB:AD". Do this for all of your stub function. The "get" stub functions are created with char * definitions. Be sure to comment them out (for now) to allow the file to compile. Now compile, link and build your project with these"dummy" functions and test out your device for look and feel. When you are happy with what you see, go to the second phase.

In the second phase you will actually fill in the stub functions with the code to either get or set the device data. Remember that set functions will store the data from the browser, into your device and that get functions get the device data from the device and give it to AWS for display by the browser.

One major caveat; "get" functions return data, to the AWS. The function then exits. If the data you return is a stack variable, you will see system crashes and failures. Ensure that returned variables are global or have the static directive.

3.8 Build the Project

Depending on which environment you are building in, will dictate the steps required to successfully build your project. If you are building under Green Hills, you will need to update the .gpj file to reflect the .c and _v.c files that are now part of your application. Remember that the .c files are in \pbuilder\html and the _v.c files are in \pbuilder. If you are building under gnu, you will need to update the make file in a similar way that you updated the .gpj file. If you are using ESP, for the foreseeable future you have an interesting characteristic to overcome. ESP works by scanning the directory structure of your application and finding all .c and _v.c files. It then compiles all it finds. The issue is that you have _v.c files in both \pbuilder and \pbuilder\html. ESP will compile both copies and at link time your stub functions will cause duplicate linker definitions. Your link phase will fail. To forestall this, rename all of the _v.c files in the \pbuilder\html directory to hide the fact that the extension is .c. Thus a file called manage_tempurature_v.c is renamed to manage_tempurature_v.c_hideme. By doing this ESP will not see these files in its scanning phase and you are more likely to successfully compile and link.

3.9 Download and Test

Finally, after you have successfully compiled, linked and built your application, down load it into your device using what ever methods you are used to using. You should now be able to test your device for real.

4 Project Description

4.1 Introduction

This AWS application was specifically developed to help customers with using two areas of NET+OS. First it was developed to help customers see real world uses of AWS and its comment tags. The hope is that customers will see that use of the AWS is not that intimidating. Second, it was developed to demonstrate a number of APIs described in the API reference guide. Our hope is that you will find these helpful in your development of NET+OS applications that include AWS.

The application displays a large number of NET+OS internal data. This includes data associated with the application, the bsp, NVRAM, ADDP and the tcp/ip stack as relating to both wired and wireless access. The data is displayed in a number of tables to give some formatting to the output data. As discussed above, where appropriate, repeat groups were used to take advantage of this feature of AWS.

Additionally, the application contains a java applet to demonstrate the inclusion of a java applet in an AWS application. Also the application includes a javascript “moving” icon on the DNS page to demonstrate including javascript code.

4.2 Project Directories

4.2.1 Top Level

Contains the root.c file, the apconf.h file and any other .c and .h files required that are not included in system locations (such as file.c and cgi.c).

4.2.2 Pbuilder

Contains list.bat or pbuilder.pbb. Also contains other files generated by pbuilder including RpPages.c, RpUsrDct.c, RpUsrDct.h, RpUsrDct.txt and rpUsrDctEnglish.c. It also contains PbSetupUp.txt. this is a file that directs the Pbuilder utility’s actions. Please consult the Advanced Web Server toolkit manual for more details on this. In addition the working copy of the _v.c file is placed here.

4.2.3 Pbuilder\html

All .htm, html files along with image and jar files and the .c (as opposed to _v.c) generated by the Pbuilder utility are placed here.

4.2.4 32b

The make file and image.gpj files are placed here. Also the output of the compile and build processes are placed here.

Please note if you are developing, all of your files are placed under the ESP development area.

4.3 Project Capabilities

The project is made up of a number of pages accessible from the main page. You will notice that each page includes a Digi logo, except for the dns page which demonstrates a javascript animation icon in place of the digi icon. The purpose for these is to demonstrate logo inclusion techniques within your web pages.

4.3.1 main page

The main page (default page you access when browsing to the device) lists a number of live links to other pages in the application. Most of the sub pages have links back to this page. The html for this page is included in file na_web_app_front_page.htm.

4.3.2 addp page

This page lists data related to the addp component of NET+OS. Most of fields were accessed via function calls as described in the API reference manual. The html page that references this page is entitled na_addp_web_page.htm. This page contains basic device data access only.

4.3.3 application configuration page

Most of the data contained on this page are related to data stored in the appconf.h file and stored in RAM, once the application is downloaded. The data could also be stored in NVRAM if that attribute is set in appconf.h. The html page that references this page is entitled na_app_config.htm. This page contains basic device data access only.

4.3.4 bsp attributes page

This page contains fields related to the bsp. The most interesting fields on this page are related to the state of the phy. The html page that references this page is entitled na_bsp_web_page.htm. . This page contains basic device data access only.

4.3.5 dev board attributes page

Dev board attributes are fields stored in NVRAM and used as defaults for the dev board when NET+OS first boots up. The html page that references this page is entitled na_dev_board_web_page.htm. This is the first page that has some interesting stuff in it. If you edit the file, you'll see near the beginning a form containing a hidden page reference. The hidden page references function getTheDevBoardStuff(). If you edit na_web_app_front_page_v.c (in the \pbuilder directory) and search for getTheDevBoardStuff, you'll see how the dev board attributes are accessed and stored in a global area for access by the other stub functions related to the dev board page.

4.3.6 dns attributes page

This page contains fields related to dns attributes. The html page that references this page is entitled na_dns_web_page.htm. This page has two interesting attributes. First if you edit the .htm file, you'll notice that this file also contains a form that contain a hidden field. The hidden field calls a function entitled getTheDNSServers(). Secondly, near the bottom of na_dns_web_page.htm, you'll notice a repeat group. This is an example a "for loop" type repeat group. This repeat group repeats twice, for two DNS servers. It causes the function getDNSserver() to be called twice. Since the type is Complex, function getDNSserver() will be passed an index. Since function getTheDNSServers() stored the DNS server addresses in an array, the index is passed into function getDNSserver to access one array entry and then another.

In addition, I have replaced the Digi logo on this page with some javascript that simulates the earth spinning. This is a demonstration of including javascript on an AWS web page.

4.3.7 global IP attributes page

This page displays attributes relating to global IP information. The html page that references this page is entitled na_ip_global_web_page.htm. This page simply displays device data.

4.3.8 IAM attributes (NVRAM) page

This page displays IAM information as it is stored in NVRAM. The html page that references this page is entitled na_web_app_iam_devboard.htm. You'll notice that this file contains a large "for loop type" repeat group that repeats 4 times. You notice that both titles and device data access are repeated. In addition near the end of the repeat group you'll notice a second repeat group. This is a "while loop type" repeat group. This was used to display, in a row, all of the DHCP options known. Since I do not know how many there are, I chose a repeat while over a repeat. Also notice that there are two functions involved. The RpRepeatGroupWhile contains a call to function iamAreWeDoneYet. This function works hand-in-hand with the AWS to tell the AWS when to quit the loop. getIamStaticDHCPOpts actually gets the options. Since the type is Complex, a loop index is passed into the function. This can be used to access the array of DHCP options.

4.3.9 IAM attributes (RAM) page

This page contains attributes relating to the current state of the tcp/ip stack and IAM. You should be able to relate the data displayed on the dialog at bootup with the data displayed on this page. The html page that references this page is entitled na_iam_RAM_web_page.htm. You'll notice that at the beginning of the file there is a form containing a hidden field. The hidden field references function getTheIAMStuff. This function (if you edit na_web_app_front_page_v.c) fills a global structure with data that the individual field stub functions will access and pass into the AWS for display in the browser. Also notice the "for loop style" repeat group for accessing the different network interfaces. Within that notice a "while loop" type repeat group for access DHCP options and displaying them in a horizontal list. Notice that the functions of both loops are type Complex, ensuring that an index is passed into the function.

4.3.10 live wireless attributes page

This page displays the attributes of the wireless part of the system. The html code for this page is included in file na_web_app_wireless_page.htm. There is no way, in the html or AWS comment tag code to know whether or not you are running on a system that supports wireless. Thus what I have done in the stub function is surround the code with an #if BSP_WANT_WIRELESS == TRUE. If this is set to false (the platform does not support wireless, I return a not supported string, otherwise I return the device data. The html file that supports this page is entitled na_web_app_wireless_page.htm. At the top of the file is a form containing a hidden field. This is used to access the wireless information and place the information into a global structure. Please notice about ½ way through the file there is a "while loop type" repeat group for displaying the WEP keys. Also if you edit na_web_app_front_page_v.c and look for function getWirelessPassword, you'll notice that all it does is return "***********". There are two reasons for this. First , I did not want my password displayed. Second it demonstrates something I talked about earlier. That is as a first pass, instead of fill in the stub function, return somethign meaningful to get the look and feel of the AWS output. This is an example of doing this.

4.3.11 java applet demonstration page

This page demonstrates the ability to include a java applet in an AWS application. In this case the applet is contained in a .jar file (java archive). The jar file is requested by the browser and the classes within the .jar file are executed. The java .jar file used is entitled javascrollpane.jar. The html file that supports this page is entitled na_web_applet_page.htm.

4.3.12 update dev board fields page

This page is a quick demonstration on how to implement a password protected page that allows fields to be updated. The html for this page is included in file na_update_fields.htm. The username is either “Netsilicon” or “digi” depending if you are using my server.h or your own. The password is “sysadm. The username and password are defined in file server.h and the manifest constants in which they are declared are APP_USERNAME and APP_PASSWORD. They are used in the call to NAsetSysAccess in file root.c in the top level directory of the project. Update the available fields and then go back and check that the fields were actually updated. The fields should be updated on the development board attributes page.

If you edit na_update_fields.htm, as part of the RpPageHeader comment tag you’ll see the option RpAccess=Realm1. This is what sets the page up to be password protected and accessible only by users with the realm1 username and password. The RpformHeader comment tag has the option of RpNextPage. This directs AWS to send the user to page Pgupdate_all_fields after the page is completely processed. Also please notice that the RpformInput comment tag has options for RpGetPtr and RpSetPtr. The rpGetPtr defines the function to be called for filling in the browser form when the page is displayed. The RpSetPtr defines the function to be called (by AWS) to take the data entered by the user and updating the device data.

5 Appendix

5.1 Glossary of terms

Advanced Web Server (AWS)
A component of Digi International’s Net+OS that allows for the development of web-based applications
Basic Web Server (BWS)
A less sophisticated component of Digi International’s NET+OS that also allows for the development of web-based application. Where possible, Digi International recommends the use of the AWS over the use of the BWS.
Comment tag
An object or set of objects added to html files to facilitate the access and/or display of device data in AWS applications.
Hidden field
An html and AWS technique allowing a web page to access device data without displaying information to the user, on a browser page. In AWS, it is generally used to access device data structures or set up some context for later use.
html
Hypertext markup language. The source code that is sent to browsers for displaying information to users.
Internet Address Manager (IAM)
NET+OS
An embedded operating system and development environment produced and shipped by Digi International

Pbuilder utility
A component of AWS that actively converts html and comment tags into C code for compiling into an AWS application.
Repeat Groups
A type of AWS comment tag allow for areas of a web page that are extremely similar in nature to be coded with less html code.
Stub functions
C code functions, generated by the Pbuilder utility and filled in by the customer for access and storing device data.
TRECK tcp/ip
The embedded tcp/ip stack that ships with Net+OS V7.0 and later.

5.2 Getting the project into ESP format

To begin with, set current directory (or through Windows explorer) in your NET+OS tree, create a directory under \src\examples. Give that directory some name that you'll remember. Next extract the project associated with this white paper into that directory. Now follow the instructions below.

The instructions for migrating a Makefile project into Digi ESP are described in the following area in Digi ESP:

Click on help
Click on Tips and Tricks
Click on Eclipse platform
Click on OK
Click on Digi ESP for NET+OS User Guide
Click on Tasks
Click on Migrating a Makefile Based NET+OS Application to a Digi ESP Project
Then follow the directions.

Example Source