Taking Advantage of Digi's Advanced Web Server's Repeat Group Feature

1 Document History

Date	Version	Change Description	
3/9/10	V1.0	Initial Entry	
3/22/10	V2.0	Continued entry	
3/29/10	V3.0	Add in corrections	
3/31/10	V4.0	Fill in glossary	

2 Table of Contents

3 Introduction

Repeat groups are an Advanced Web Server (AWS) technique, whereby the AWS generates html tags and accesses device data on a repeating basis, for applications where the same controls repeat. This saves memory space as the html is not stored, but is generated as required on request from a browser. This document describes the development of a Digi Net+OS project which includes Digi AWS repeat groups. Sample code here - http://ftp1.digi.com/support/documentation/repeat_groups_project.zip

3.1 Problem Solved

There is a class of projects that involve web server access that involve multiple instances of objects. That is, objects repeat in the applications. One method for implementing such an application would be to create multiple web pages where each page is tied to an instance of the object. This would have two deficiencies. First it would be quite tedious to implement. Secondly, as conditions changed, additional work (tedious work) would be required to update the application to meet the changes in the real world that the web application is monitoring.

Another way to implement such an application would be for the data being monitored to control the generation of web server data so that as conditions changed, the web pages would automatically change accordingly. This type of dynamic web page generation can be achieved using AWS repeat groups. Now, one caveat from the start, what is changing is the number of objects within a structure (or multiple structures). But, there is some basic structure in the application that will be assumed to be constrained. As we look at the application that this paper describes, I hope you will get a feel for what I am talking about.

What this paper describes is the second method. That is, the ability for a Digi AWS application to change the number of objects displayed on a web page, based on the current condition of device data.

3.2 Audience

The audience for this paper is software engineers with experience developing firmware projects under Digi's NET+OS development system. Additionally the audience will have had experience developing Digi NET+OS projects that include web pages served by the AWS component of Digi's NET+OS development system.

3.3 Assumptions

This paper assumes that you are going to open up the files in the attached application and look at the files that are discussed later in this paper. You can probably gain insight by reading this paper only. On the other hand, I would expect you to gain further insight by reading this paper and following along with the files in the application.

3.4 Scope

The scope of this white paper is to describe the use of repeat groups in a Digi NET+OS development project using the AWS component of Digi's NET+OS development environment.

The scope of this white paper does not include any of the following:

- Training about developing web (html) pages
- Training about developing projects under Digi's NET+OS development system
- Training about AJAX
- Training about developing C code projects
- Training about bringing up a new platform using Digi's NET+OS development system

3.5 Theory of Operation

The advanced web server (AWS) has gone to some lengths to help the developer save memory. One example of this is the use of dictionaries. The technique described in this paper is repeat groups. This technique allows for an object on a web page (in an html file) that might occur multiple times, to be described once, in the html code, and have AWS generate the required code for accessing and displaying multiple instances of that object to the browser. In addition, depending on your implementation, the number of instances of that object can vary, from browse to browse. The theory is that the description of that object and its attributes is consistent across all instances of that object. If you are describing printers, they all have a consistent set of items attached to that printer (unless those items are also repeat groups). This picture of that object, that printer in this case can be described once. Then at execution time, you can tell AWS how many of things objects, in this case those printers, actually exist. The browser has no notion of the changes, except more or less of that object are displayed in the browser.

4 Basics

For Developing a Digi AWS-based web page that includes repeat groups, I would recommend the following steps. This is a conservative approach, but one that I find brings you to success quickly.

1. Create a flat html file (no repeat groups or AWS comment tags).
2. Continually surf to this page using your favorite browser until it has the look you desire.
3. To the html file add AWS comment tags that deal with data access only (no repeat groups yet)
4. Run this file(s) through the pbuilder utility
5. Fill in the "stub functions" (contained in the _v.c file) passing static data back through the callbacks (we are not ready for "real" device data yet)

6. Make sure you have a backup copy of your _v.c file as we will be running the pbuilder utility again, and doing so overwrites the _v.c file.
7. Build your application, and browse to the device that is running your application and evaluate the output.
8. Continually perform steps 3 – 7 until your page looks acceptable
9. Take a hard look at your web page, as it stands now, and get a feel for what parts should repeat and what parts should be nested in what other parts.
10. At the beginning of an area that you'd like to repeat add the appropriate AWS comment tag. In the attached project, I used the *RpRepeatGroupDynamic*. There are, in fact, three different repeat group "header tags", RpRepeatGroupDynamic, RpRepeatGroup and RpRepeatGroupWhile. These are described in the Advanced Web Server Toolkit guide.
11. At the area where the repeat group should end do not forget to add an *RpLastItemInGroup* comment tag. Also keep in mind that unlike most other AWS comment tags, these comment tags do NOT pair with an RpEnd comment tag.
12. Place your repeat group header and trailer comment tags in the appropriate places in your html file. This is a step where attention to detail is very important.
13. Rerun your html file(s) through the pbuilder utility. Again remember to back up your _v.c file (each time) before rerunning the pbuilder utility.
14. Rebuild your NET+OS application and download it into your device.
15. Browse to your device and check the look of your web page.
16. Repeat steps 13 – 15 as needed
17. Now replace the static data, in the stub functions with your device data.
18. Rebuild and retest as needed.

4.1 Indices in Your Stub Functions

When you get your repeat groups running cleanly through the pbuilder and compiling and linking under the NET+OS development environment, you'll want to start debugging your repeat groups. There is a good chance that your repeat groups will be nested and thus the Digi AWS will pass your stub function(s) indices. The way these indices are passed is not well documented, so I will spend a little time describing it here.

When an AWS comment tag includes the get or set type of Complex (required for repeat groups) you will notice that the last parameter passed to many of your stub functions is a field entitled theIndexValuesPtr. The word Values is most important here. There may be between one and eight indices contained in this parameter (you can nest repeat groups up to eight deep). What this actually contains is an array of indices. The 0^{th} element contains the least frequently changing index. The 1th element contains the next most frequently changing element and so on up to eight (0 – 7). There is nothing inherent in what is passed to your stub function, to tell you how many of the elements in the index array are valid. The assumption is that you know the layout of web page and your data and so you know how many indices to extract for any given field.

The presumption is that you have arranged your data such that the indices are useful for accessing the correct data for this get/set.

To wrap this section up, I will provide an example; let's say your application is managing a number of farms. Each farm has one or more barns and each barn has some number of horses. The 0^{th} index would represent the farm number. The 1st index would represent the barn on that farm. The 2^{nd} index would represent the horse within that barn on that farm.

5 Example Application Explanation

The application associated with this white paper implements a printer management system. In this mythical world, there are a number of buildings, in our case ten. Each building has two site managers. Each building has a varying number of printers. Each printer has a varying number of paper trays that can be in varying states. Also each printer can have a varying number of print jobs. Those jobs have owners that are associated with those jobs (presumably the owners requested that the jobs be printed).

The available data is implemented in a number of arrays. Which element of the array is accessed at any point in time, is controlled by some pseudo-random data (based on the index and the number of timer ticks the operating system has counted). In addition, the number of printers, print jobs and printer trays are all controlled by similar pseudo-random numbers. In your application, the presumption is that you'll be tracking some real device data, and things will change or not change based on that data. In this case, a random number is used to show that the number of repeats from access to access can change (but does not have to do so).

Based on paragraph one, you can see that the data is nested to a maximum depth of three (building/site manager, building/printer, building/printer/tray, building/printer/job). The advanced web server allows for a maximum depth of eight.

5.1 Html code and comment tags

For this section of this paper, you'll want to open file rg_main_page.html. It should be contained in the project directory\pbuilder\html.

The web page is fairly simple. It is the repeat group structure that makes it a little daunting. The first repeat group starts just after the <table> tag. This sets up the repeat group for the buildings. This repeat group ends just before the </table> tag. Please notice that this has a *function* pointer of `getBuildingLimits.` This causes a stub function entitled getBuildingLimits to be generated when the pbuilder utility processes this part of the html file. I'll explain what this function does when we explore the _v.c file.

The next repeat group starts just after the text "Site Manager Information". This sets up the repeat group for the number of site managers watching this building. You'll notice (and it is easy to see in this isolated case) that the html code and AWS comment tags for only one manager is stored here in the file. But when you run the application, you'll see two site manager records. This is the power of repeat groups. The site manager repeat group ends (RpLastItemInGroup comment tag) right before the next repeat group begins.

The next repeat group is for the printers in this building (so this repeat group is nested in the building repeat group). It has a function entitled getPrinterLimits. Again, we'll discuss this when we discuss the _v.c file.

As you follow the code in this html file, you'll see repeat groups for printer trays and printer jobs. I have named the "get limts" functions strategically, thus making them easier to find. I would recommend using similar naming conventions when you write your repeat groups.

The outer repeat group (building) and inner ones printer and jobs, all end near the end of the file. The repeat group for trays ends further up.

Also notice that within each repeat group, are the standard AWS comment tags for getting device data. These do not change when using repeat groups.

5.2 Rg_main_page_v.c

The next file we'll look at is called rp_main_page_v.c. It is located in the pbuilder directory of the project that came with this paper. This contains the stub functions generated by running the pbuilder utility against your html file(s).

The tables at the top of the file are used to represent device data. In a real application, you'd be accessing some information stored or generated by your device. For the purposes of this white paper, though, I am using these tables.

First, let's skip down to getSiteManagerLimits. In this case, I am assuming that we'll never have more than 2 site managers at any one building. The RpRepeatGroupDynamic AWS comment tag, causes code to run that is analogous to a for loop. So you see in getSiteManagerLimits we are setting up the parameters for a for loop. In a standard C language for loop you have a construct looking something like the following:
For (index = 0; index< some_value, index++). So, the first part gives the index an initial value (a start). The second part sets an upper limit for the loop and the third part increments the index. So, in getSiteManagerLimts you are setting up similar values. theStart is the initial value. theLimit is the limit on the number of iterations. theIncrement is the amount by which the index should increment each time.

If you look at getPrinterTrayLimts you'll see something a little different. In this case the limit is being set to a different value each time the page is surfed to. This shows that the limit does not have to be a fixed value. Thus, if the number of objects your device is managing changes over time, the number of repeats can also change allowing your application to keep up with the reality of your device. In my case, I am generating a random number based on the number of ticks since the system was rebooted. The mod 5 ensures that the number I generate equates to an entry in one of the tables above.

The remainder of the functions in this file, are either similar to the get limits functions we have already discussed or are standard get functions similar to what you have done on other AWS projects.

5.3 Indices

The only other things to explore in this file are the indices. We'll concentrate on function getThePrinterJobOwner. I concentrate on this function as it is nested three levels deep and thus has three (3) indices. Notice that I am treating the pointer as an array and am pulling the indices out one at a time. Remember also that index array[0] has the LEAST frequently changing index, that index array [1] has the next more frequently changing index and index array [2] has the next frequently changing index. I am using the indices to generate a pseudo-random number to use as an index, so that the information filling the fields of the web pages changes and thus is visually interesting. In your case, your information will probably not change that frequently.

5.4 The rest of the project

The remainder of the project is a standard run-of-the-mill AWS project and should build in a similar manner to any AWS-related projects you have built in the past.

6 Conclusion

Repeat groups give the developer the flexibility of having an AWS project expand and contract based on the number of objects being managed by the device. In the html file, only one instance of such an object is defined and then using the "get limts" callback (stub function) AWS generates code required to display the number of objects requisite for the situation. Additionally, I believe I have shown that over time, if the number of objects, under management by the device changes, the "get limits" callback can return different values, thus allowing the web pages returned to the browser to adjust with the changes but not requiring the developer to change any code (assuming the "get limits" callback has the ability to sense the change and communicate the change to AWS through the "get limits" callback).

7 Appendix

7.1 Glossary of terms

For every acronym and non-standard term, present the term/acronym and explain its meaning.

- Advanced Web Server (AWS) – a component of Digi's NET+OS embedded operating system that provides embedded web server capabilities to NET+OS's offerings.

- AJAX – Asynchronous JavaScript and XML. A method of writing html pages that provides a more PC-like response to browser-related activities.
- Browse – using a web browser (such as MS IE, Monzilla Firefox or Google Chrome to access a web server. Can also be accomplished more programmatically through APIs in languages such as PERL.
- Device data – Data that is stored within the device for which you are writing your application. Generally the accessing of this device data is the reason you are browsing into your device.
- Digi NET+OS – Digi International's embedded operating system and development environment.
- Html – Hypertext Markup Language. The language in which most web pages are written.
- pbuilder utility – a component of Digi's NET+OS embedded operating system. It is used for transforming html code and AWS comment tags into C code for inclusion in your AWS-enabled application.
- static data – Parts of a web page that do not change
- stub functions – As a result of running the pbuilder utility against your web page(s), functions are created in a c file whose name is <your html page>_v.c. Since in their initial instance, they are empty, they are referred to as stub functions. The idea is that you fill in the stub functions thus give the AWS access to your device's data (device data).
- Surf – A synonym for browsing.