Ajax programming using Digi's Advanced Web Server

1 Document History

Date	Version	Change Description
3/12/09	V1.0	Initial entry
3/13/09	V1.1	Additional content and edits

2 Table of Contents

Contents

3 Introduction

Asynchronous JavaScript and XML (Ajax) is an new and exciting use of existing technologies that make web-based application act more like workstation-based applications. This white paper discussed the use of Ajax with Digi's NET+OS-based Advanced Web Server (AWS). Example code can be found at this link: http://ftp1.digi.com/support/documentation/aws_and_ajax.zip

3.1 Problem Solved

In the normal web-based, form-based application today, a user fills in a form. The problem is that the user must completely fill in the form before the data is sent to the device. In many circles today, this is considered slow and backwards. To give a web-based application a more workstation-based feel, developers want their applications to update fields as the data is entered by the user. Additionally they would like this to happen without the user really being aware that it is happening. This paper discusses how to do this in a NET+OS AWS-based environment.

Additionally, up until now, there has been no documentation available to describe how to take advantage of this technology in an NET+OS AWS environment. This paper addresses this deficit.

3.2 Audience

This document is intended for users with experience developing applications using either of Digi International's NET+OS development environments. This includes the Green Hills IDE-based development environment, the gnu-based command line interface (CLI) development environment and the gnu-based ESP (GUI)-based development environments. In order to get the most out of this document, the user should have experience developing web-based NET+OS applications using the advanced web server (AWS) and its pbuilder utility. The user should also have some knowledge of HTML and JavaScript development. In addition, as part of JavaScript knowledge, the user should have some understanding of object oriented programming terms as they are used in the description of the JavaScript code examples.

3.3 Assumptions

This document assumes that the user has access to a Digi International NET+OS-based development environment and an embedded development board for trying out the recommendations described in this document. You will be able to get some knowledge from reading this document only, but actually developing a test application and deploying that application to an embedded device will afford the reader a deeper understanding of the material presented herein.

Part of this document describes programming in the JavaScript language. It is important to understand, if you do not already know so, that there is no relationship between JavaScript and Java. The only thing relating in these two languages is the use of Java in their names.

The techniques used in this document are applicable to all of Digi International's embedded devices that run with Digi International's NET+OS embedded operating system.

The techniques described in this document, require the use of additional pbuilder comment tag attributes associated with form fields. In normal mode, the pbuilder utility does not recognize these attributes. For this reason, a change to the file PbSetup.txt is required. Look for the string useJavaScript. It may be commented out. If it is commented out, uncomment it. If it is already uncommented, then leave it alone.

3.4 Scope

This document describes techniques for using existing technologies for implementing web-based applications using Ajax techniques. This document does not present any new technologies. Instead it presents new ways to utilize existing technologies.

The following items and subjects are considered outside the scope of this document:

- A tutorial on HTML programming. An HTML file is included in the example application. However no general HTML programming techniques are included
- A tutorial on JavaScript programming. The HTML file included in the example application does include some JavaScript code. However no general JavaScript programming techniques are included in this document
- A tutorial on C programming. The root.c and pbuilder-generated C files are included in the example application. However, no general C programming techniques are included in this document
- A tutorial on web design.
- Use of the Basic Web Server (BWS). This paper is applicable to the Advanced Web Server)AWS) only.
- A tutorial on tcp/ip. The web browser and the web server communicate using the tcp/ip suite of protocols. Discussion of anything related to this topic is outside the scope of this document
- At this time, the sample application supports the Microsoft Internet Explorer browser only. As time allows we hope to add support for other web browsers.
- Though Ajax can transfer xml or text, in this paper and the accompanying example application, we transfer text only. Use of xml is outside the scope of this document.

Should you feel you want to enhance your understanding of some of these topics I can recommend the following books as a start:

- For starting Ajax programming: Ajax for Dummies. Steve Holzner, PHD. Published by Wiley Publishing, Inc.

- For starting JavaScript programming: JavaScript, The Definitive Guide. David Flanagan. Published by O'Reilly & Associates, Inc.
- For starting HML web design: HTML & XHTML. Chuck Musciano & Bill Kennedy. Published by O'Reilly & Associates, Inc.
- For an understanding of the HTTP protocol: HTTP The Definitive Guide. David Gourley & Brian Totty. Published by O'Reilly & Associates, Inc.

3.5 Theory of Operation

In general and prior to Ajax techniques, web-based applications worked as follows: A user browses to the web server that houses the application the user wants to access. This might be a site related to a store from which the user wants to order some product. This might be a site related to a subject about which the user wants to learn more. This could be a site relating to an organization to which the user wants to join. The user is presented with an HTML-based form. The user fills in the form and hits the submit key. At that time, the user's data is sent over an HTTP/tcp/ip message (or set of messages) to the web server. The web server would digest the data, update database that is part of the web server's application. The web server might also kick off some process for doing some other work. The feeling became that following this process was slow and less responsive then users were used to when using a workstation or desktop-based application.

Use of Ajax, as part of a web-based application, fixes this weakness. In a nutshell, as users fill in fields in a form and tabs to the next field, the fields from the prior field can be transferred to the web server for processing. The JavaScript code packages up the data, that needs to go to the web server, and sends it in an HTML/tcp/ip packet, to the web server. In addition, when used in asynchronous mode the sending of the data (by the browser) and the processing of the data (by the web server) can proceed at the same time that the user is entering the next field. Thus, the user is unaware that all of this advanced technology is going on.

How is this done? Ajax introduces the XMLHttpRequest object. This JavaScript-accessible object gives your browser application full control over sending messages to, and receiving messages from your web server. So instead of waiting for the submit key to be hit and the data being sent only then, the data can be sent on a field-by-field basis. More specifically, additional attributes are added to pbuilder comment tags instructing the web browser of JavaScript functions to be called on certain actions. In this case that action is when a field changes (onChange). It is the JavaScript function that is responsible for packaging up the data and sending it to the web server along with accessing data from the web server.

4 Using Ajax Techniques in AWS

4.1 Introduction

In most cases, the easiest way to understand the use of Ajax techniques in the NET+OS and AWS environments, is to look at a working example. So for the next few sections of this white paper, we will be referring to the example application associated with this

white paper. If you have not downloaded the example application and made its files available to your favorite editor, you should do so now before proceeding.

The first section deconstructs the html file (ajax_example.html). We will look at both the AWS comment tags and the JavaScript functions. These are related as we will see the comment tags make reference to and thus make calls to the JavaScript functions.

The second section looks at the AWS "stub" and helper functions. These are contained in the file ajax_example_v.c. These functions act as the interfaces between the web browser/web server as a matched set and the embedded device. That is the web browser makes requests. The web server (AWS) fields these requests and processes these requests by interacting with the embedded device. For actions going back to the browser, the device uses stub functions to interact with the web server (AWS) which, in turn, communicates with the web browser.

4.2 The HTML file

4.2.1 HTML code

The html file used in the example application and to which we will make many references is entitled ajax_example.html. If you are developing in the Green Hills Multi IDE environment or the gnu CLI environment, this file is located in src\examples\<project name>\pbuilder\html. If you are developing in the gnu ESP IDE environment, this file is located in one of two places. If your project is built with services, it is in <project>\web\html\ajax_example.html. If you built a sample application without services, it is located in <project>\pbuilder\html\ajax_example.html.

Because I want to start with code familiar with Digi customers and then move to code that might be unfamiliar with Digi customers, we will begin near the end of the file and work backwards. Look for the line that begins as follows:

```
<-- RpFormInput TYPE=text Name=theName……..
onChange="postSomeData('theName');"  -- >
```

For anyone familiar with developing in a NET+OS AWS environment, most of this should look familiar. One thing should be pointed out immediately. In this comment tag, I have given the field a name. In addition, in the RpFormHeader comment tag, I have given the form a name. So an initial question is do I really need to give everything names? The answer when programming using Ajax techniques is that your life will be much, much simpler, if everything has a name. When we get to the JavaScript, portion, I will explain way this is so. For now, please take this on faith and make a note of it.

The attribute to the comment tag that may be unfamiliar, is onChange. The onChange attribute says "when the user changes this field in the form and as soon as the user then defocuses this field, do something. In this case, the "do something" is make a call to function postSomeData(). Please notice a couple of things. First, the function call is in quotes and ends with a semi-colon. These are required. Second, I am passing the name of

the field to the function. This allows the function to perform the correct setup to update only that field (in this case, theName).

4.2.2 JavaScript code

There are three JavasScript functions defined in the file, as follows:

- makeTheRequest()
- getUpdatedServerData()
- postSomeData()

4.2.2.1 makeTheRequest()

The core of Ajax programming is the XMLHttpRequest. This is a data structure used by Ajax for communicating through the web browser to the web server. There are three varieties of this structure, which is the reason for the multiple try/catch pairs. The hope is that one of them will be recognized and placed in the request variable. All Ajax tasks, begin with an attempt to create one of these structures.

4.2.2.2 getUpdatedServerData()

For the purposes of this paper, there are two actions you might want to perform in Ajax. One would be to get data (for example updated data) from the device. The other might be to post data to the device. We will look at the first one in this section and the second one in the next section.

My reason for this function is as follows. Your embedded device might, on its own, update data. Based on this assumption, you might like the ability to request an update of the data from the embedded device. Function getUpdatedServerData() performs this task.

The first subtask this function performs is to create a variable called whereIsTheCustomer. For simplicity purposes, I have hard-coded the ip address of my device in the variable. You will probably want to do something more creative for obtaining the ip address of the device. The remainder of the constant is the path to a file on the device's file system that is going to be pulled from the device. You'll want to notice the specifics of the path format. "FS" tells AWS that the thing we want is in the file system. "RAM0" says that the thing we want is in device RAM0. If the file was on the FLASH file system, you'd change this to FLASH0. Finally formsData.txt is the actual file name. Since the file is created in a "helper function in the file with the AWS stub functions, I will not discuss its creation in this section, only its use.

You will notice the call to the function makeTheRequest(). This creates the XMLHttpRequest data structure for use by the Ajax code. If the data structure is created, we call the open method of the request instance, passing it the action we want to perform (GET) and the access string that we created above. The calls to the method setRequestHeader are attempts to set HTTP headers for the outgoing HTTP request. I do not believe that actually help with much but act as a demonstration on how to set HTTP headers in Ajax.

The next line needs some explanation. We are setting up an unnamed function that will be called when (and if) we get a response for the request we are making. Member status == 200 is the OK return code. Member readyState ==4 is the complete ready state. You will need a book on Ajax to get the complete list of status codes and ready states. This is beyond the scope of this paper.

Assuming we got valid data and it is complete, we need to do something with it. The data returned in this example is in the form of name/value pairs. The name and the value are separated by an equal sign ("="). Further the pairs are separated from each other with a pound sign ("#"). Thus the purpose of the remainder of this function is to break the set name /value pairs into an array of them. Then each array member is, in turn, placed into a second array, where the equal sign is removed. Thus we end up with a two member array with the name in position 0 and the value in position 1. Finally the value is placed into the value member of the form field structure which is part of the form structure which is part of the document structure. For more in depth descriptions of these members, you'll need a JavaScript book. As a reminder, I recommend one in the Scope section of this paper. Further detail are outside the scope of this paper.

4.2.2.3 postSomeData()

The JavaScript function postSomeData is used to send field-by-field updates to the web server (AWS). The parameter passed in is a key to the field to be updated. If you'll remember, this is passed in the onChange attribute to the form field comment tag in the html code. If you forget, this might be a good time to jump back to the html code section and review that before continuing. The first thing we do is set up the variable whereIsTheCustomer. We did a similar task in function getUpdatedServerData(). The difference here is the path and the object of the request. We are NOT asking for the web page. We are , instead looking for the actual form in the Forms directory, which is built into your AWS application. To find this for your application, you will need to edit your .c file associated with your html file. In my case, this is ajax_example.c. further, in this example the path you are looking for is /Forms/ajax_example_1.

Here you see that the parameter is used to differentiate which form field we are dealing with. The postData variable holds the form field instance. The dataType holds a string constant that will be used in building the request. There will be more on this in a moment. The length holds the length of the string contained in the dataType variable. After all this is set up, you'll notice a call to the function makeTheRequest(). You remember that the structure created therein is the heart and soul of Ajax programming.

The request's open method is called to open a channel to the web server. The request's setRequestHeader() method is called. This header IS REQUIRED for the request to be handled by the web server.

You will notice that we have a similar unnamed function for handling the return from the web server. In our case, the return is not used. In your application you might want to check for failed return codes.

The code just after the unnamed function, sends the request to the web server. Notice that we are creating a string in the form of (for example) theName=”Mickey Mouse”. If you know something about HTTP requests you’ll notice that the ? is missing. This is because the Ajax code takes care of that for us. The dataType variable, mentioned above, holds the string placed to the left of the equal sign. In this case, datatype holds the string “theName”.

*4.3 The “stub” functions (*_v.c file)*

4.3.1 Introduction

This section describes the stub and helper functions that make up this AWS-based Ajax application. What you will probably notice is that besides the creation of the file in the file system, there is nothing new here. The “heavy lifting” in the Ajax-ness of this application is in the html and JavaScript code, not in the AWS stub functions.

4.3.2 Operations

The first thing you will notice is that there are a set of get and set functions, one for each field in your html form. Further I would b surprised if your reaction was not something like, “This looks just like any other set of get and set functions that I have written before”. If that is your reaction, you are correct. The only difference is the call to function updateFile() at the end of each set function. This function facilitates the getUpdatedServerData() Javascript function that we talked about earlier. That is the only function that we spend any time with, in this file.

First we create a read and a write buffer. The buffer size is arbitrary and highly application dependant. You’ll be the best judge of this. Also notice that I create them once and reuse them, each time around. The file is opened with an open call. I chose to use the C library function calls as opposed to using the NET+OS specific file system function calls. This was done for simplicity, your mileage may vary. The file name is fixed and needs to be the same as the file that is requested in the getUpdatedServerData() JavaScript function call, described above. The file is read into the read buffer. If this is the first time through, I fill the read buffer with some default data. This purely simplifies later code.

The file we are talking about is a method, my method for getting device data updates back to the browser’s form. The format of this file, as designed by me is as follows: it is a set of name/value pairs. In each name/value pair, the name is separated from the value by an equal sign (“=”). Further each pair is separated from the next one by a pound sign (“#”). Thus, if you look at the strtok call, you notice that I am making an array of name/value pairs. The lion’s share of the rest of the code builds the write buffer from a combination of the contents of the name/value pairs array and the latest field being set. Next the file is closed. This is done to reset the file pointer back to the beginning (I want

to overwrite as opposed to concatenate). Next the file is reopened and the data is written back to the file system. Lastly, the file is closed.

There are a couple of important issues that you need to keep in mind if you are going to implement something like this. First, in file bsp.h (pre-NET+OS V7.4) or file bsp_fs.h (NET+OS V7.4 and above) you'll want to set manifest constant BSP_INCLUDE_FILESYSTEM_FOR_CLIBRARY to TRUE and rebuild your `bsp.` `Also in your applications appconf.h, the manifest constant APP_FILE_SYSTEM must be defined.` Second, by default, NET+OS does not have a sense of time. For many applications, this is of no consequence. If you are going to pull files off a device's file system, using Ajax, it is a problem. It is a BIG problem. The problem is that without a clock, all files get the time/date January 1, 1970 at midnight. Thus when you try successive gets, you notice that the browser's caching system is getting in the way of your retrieving updated data. To fix this, I recommend turning on SNTP. This way every time you write data to the file system, the file's date will be updated and caching problems will be averted. To turn on SNTP, in your appconf.h define APP_USE_NETWORK_TIME_PROTOCOL and set the sntp server addresses to something other than 0.0.0.0. In bsp.h (pre NET+OS V7.4) or in bsp_sys.h (NET+OS V7.4 and above) define BSP_INCLUDE_SNTP to TRUE and rebuild the BSP. The BSP will automatically set up time.

5 Application specific information

The application is a simple AWS-enabled NET+OS example application. The web page (ajax_example.html) includes a form with 6 text fields that are updatable. Additionally, there is a submit button. There is also a second button allowing the user to update the content of the form with device information (assuming the information changed in the device. The application should be buildable in the Green Hills, GNU CLI or GNU ESP environments.

I have included an ftp server in the application, in addition to the AWS. This will allow a user to access the file system and see the server data file while it is being written to the file system. In file root.c look for calls to NAsetSysAccess(). These calls set up users and passwords for users. You will need a user and password pair to successfully log into the ftp server.

When the ftp server some up, you might see the error message "APP: FLASH0 – FSInstallFileSystem[-1]." In your hyperterm output. All this means is that a FLASH file system already existed. It is not a fatal error for this example application.

6 Conclusion

Though this is a simple example, we hope this gives a web developer a taste of what can be accomplished using Ajax techniques in Digi's NET+OS AWS environment.

7 Glossary

This glossary explains some of the terms and acronyms used in this document.

- Advanced Web Server (AWS) – A component of Digi's NET+OS operating system that provides an embedded web server.
- Ajax - (asynchronous JavaScript and xml). A method for using html, JavaScript and (sometimes) xml to enhance the usability of web-based applications
- CLI – command line interface. The style of interacting people did with computers before the advent of GUI. The DOS shell, Bourne shell, C shell and the Bash shell all implement forms of command line interface.
- Comment tag – The extra content added to HTML files, that the PBuilder utility uses to generate stub functions
- Green Hills Multi – An IDE (integrated development environment) produced by Green Hills
- Helper functions – additional functions, used by stub functions for accomplishing some task.
- HTML – hypertext markup language – a tag-based language used for implementing web pages
- JavaScript – A c-like language used in HTML files for performing additional processing not available through HTML alone
- Member – in object oriented parlance – a variable associated with an instance object. Might also be seen as a field in a data structure.
- Method – an operation associated with an instance object.
- MS Internet Explorer – Microsoft's web browser
- NET+OS - an embedded real time operating system sold by Digi International
- PBuilder utility – a component of the Digi's AWS that converts HTML files into c code for inclusion in a NET-OS-based web-enabled embedded application
- Stub function – a set of functions, generated by running HTML files through the PBuilder utility. The stub functions act as the go-betweens, between the device and the web server (AWS).
- tcp/ip – a suite of protocols used for the transporting of data across the internet. HTML, JavaScript, Ajax and many other objects are transported within tcp/ip packets.
- Web Browser – a program used to interact with web servers. MS Internet Explorer, Netscape Navigator and Mozilla Firefox are all examples of web browsers.
- Xml – extensible markup language. A tag-based language used to store information