



Handling multiple submit buttons in  
NET+OS development environment  
web-based applications

## 1 Document History

Date	Version	Change Description	Initials
1/31/2014	V1.0	Initial Entry	JZW
2/4/2014	V1.1	More initial entry	JZW
2/10/2014	V1.3	First round updates	JZW

## 2 Table of Contents

1	Document History .....	2
2	Table of Contents .....	3
3	Introduction .....	4
3.1	Problem Solved .....	4
3.2	Audience .....	4
3.3	Scope .....	4
3.4	Theory of Operation .....	5
4	Why more than one submit button? .....	5
5	Accessing the identity of the submit button clicked .....	5
5.1	The web page we are using .....	6
6	The post-processing function .....	9
7	Delaying the write (set) function .....	11
8	When are my “get” functions called? .....	13
9	Conclusion .....	13
10	Appendix .....	13
10.1	Glossary of terms .....	13

### **3 Introduction**

A simple web application page might contain one submit button. Maybe one submit button and a reset button. The question this paper answers is, can NET+OS's advanced web server (AWS) component handle multiple submit buttons? The answer is yes.

#### **3.1 Problem Solved**

A simple web page would allow a user to fill in some fields and hit a single submit button. The submit button would signal the browser to send the data to the web server. The web page might also contain a reset button that signals the browser to clear the fields in the web page without sending any messages to the web server.

Suppose, though, you have developed a more sophisticated web page. This web page might allow multiple actions by virtue of having multiple submit buttons, in addition to the reset button. This paper explains a method for allowing the AWS engine to sense which of the multiple submit buttons was clicked and have the AWS engine process the form differently based on which submit button was clicked.

#### **3.2 Audience**

This white paper is geared for developers with experience in developing web pages, web applications and applications within the NET+OS development environment. This white paper should not be considered a beginners guide.

#### **3.3 Scope**

This paper assumes the reader has knowledge of and experience in developing web pages using HTML, web applications and general applications within the Digi's NET+OS development environment. In addition it assumes knowledge of the pbuilder utility which is a software tool of the AWS component of the NET+OS development environment. There have been other papers published that give more of a beginners view of developing applications in this environment. Further, this paper discusses creating a web page containing more than one submit button. This is the focus of this paper.

This paper makes no attempt to explain any of the following:

- General development of applications using Digi's NET+Os development environment
- General development of web applications
- C code development
- HTML
- Javascript
- TCP/IP
- Socket coding
- Any other subject not described above

### **3.4 Theory of Operation**

The project included with this paper demonstrates a simple web page using HTML tags and pbuilder comment tags. Further, the web page associated with the application contains multiple submit buttons. The application uses the pbuilder API `RpGetSubmitButtonValue()` for requesting the value associated with a particular submit button. After obtaining this submit button value we will look at dispatching to different activities based on the submit button pressed.

## **4 Why more than one submit button?**

Up until recently my understanding was that a web page needed one and only one submit button. It might also employ a reset button, but the reset button interacts with the browser and thus does not interact with the web server. The web page allowed the user to fill in the fields of a web page and then the submit button sent, via a web browser, the data contained in the web page fields on to the web server. The web server would update some fields on the web server, might update fields held by the web server (device data) and return updated fields to the browser.

Recently a customer requested a web page with the ability to perform an update (the more conventional application as described above) but to also be able to read data from the device without performing any updates. Thus a read button and a write button would be required.

Based on the requirement above, we have two main issues to solve. First, since when a user clicks on any submit button, the action taken by the web server is to call all “set” stub functions associated with that web page. So in the case of a “read” submit button, how do we keep the “set” stub function from updating device data? Second, the way the web engine works, all of the “set” stub functions are executed before the post page function is called (the post page function is a function associated with the clicking of a submit button). This is where we can identify which submit button was clicked. The problem is that it is too late to keep the “set” functions from executing in the case where we want to perform a read function only.

The overriding question posed by the two dilemma described above is how do I successfully dispatch on the submit button clicked, if I will not have access to the identity of the submit button clicked until after all of my “set” functions have executed? In the sections below we will look at my proposed solution to this issue.

The example application can be found at this [link](#).

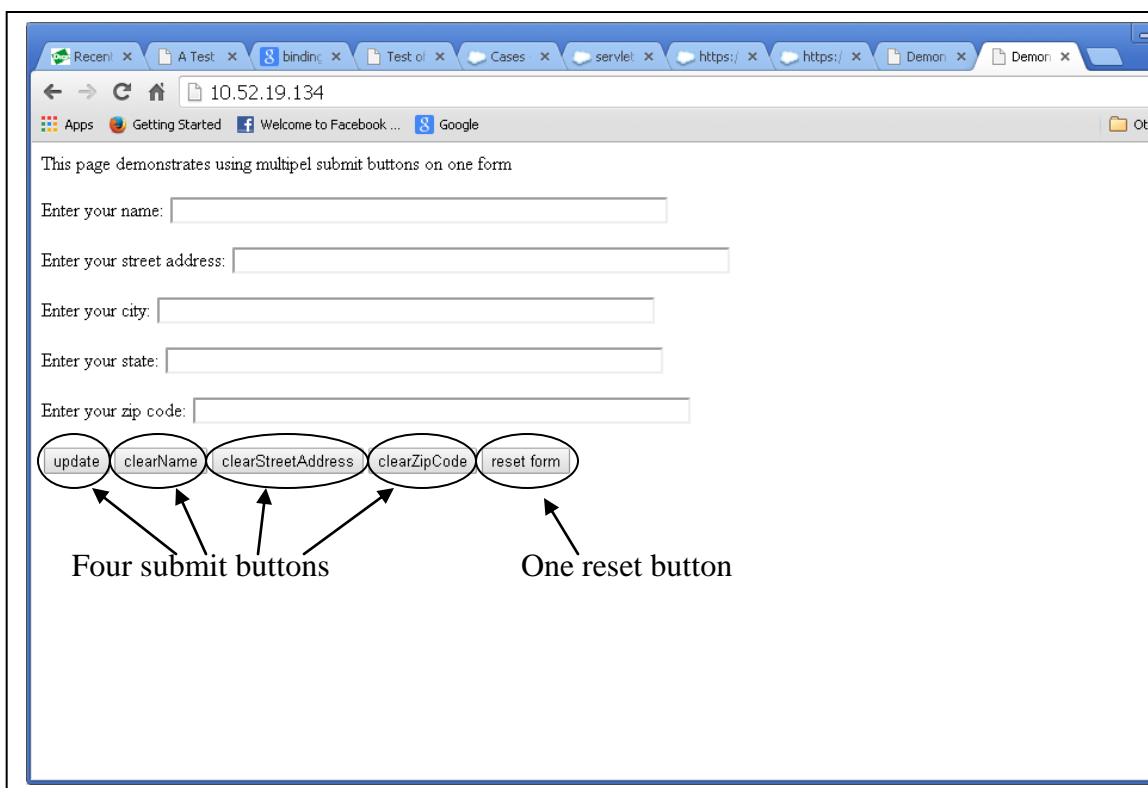
## **5 Accessing the identity of the submit button clicked**

If your application contains multiple submit buttons on a page, then your application needs a way to determine which submit button was pressed. The advanced web server’s toolkit includes an API entitled `RpGetSubmitButtonValue` for this purpose. This call returns a `char *` and is passed the `void *` representing the web server’s data structure. API

RpGetSubmitButtonValue is described in the Advanced WebServer Toolkit document. To access the web server's internal data structure, use API RpHSGetServerData. This requires no input and returns a pointer to void. If the returned value is NULL, then the web server is not running. Additionally from experimentation it appears that the only place that you will receive a valid return value from RpGetSubmitValue is within a post processing function. We will discuss this later.

### 5.1 The web page we are using

The following graphic displays the application's web page.



Notice that I have created a web page containing four submit buttons and a reset button. As described earlier, the reset button activates activity in the browser only and does not send data to the web server. The submit buttons are entitled update, clearName, clearStreetAddress and clearZipCode. The reset button is entitled reset form.

The next graphic contains the HTML and pbuilder comment tag code used to create this page.


## Handling multiple submit buttons in web applications

```
<!-- RpPageHeader RpFunctionPtr="prePageProcessing" -->
<html>
<head>
<title>Demonstrates using multiple submit buttons</title>
</head>
<body>
<!-- RpDZT -->
This page demonstrates using multiple submit buttons on one form
<p>
<!-- RpEnd -->
<!-- RpFormHeader method="post" RpFunctionPtr="postProcessTheForm" -->
<form method="post">
<!-- RpDZT -->
Enter your name:
<!-- RpEnd -->
<!-- RpFormInput type="text" name="theName" Size="64" MaxLength="128"
      RpGetType=Function RpGetPtr="getTheName" RpSetType=function
      RpSetPtr="setTheName" -->
<input type="text" size="64" maxlength="128" name="theName" >
<!-- RpEnd -->
<!-- RpDZT -->
<p>
Enter your street address:
<!-- RpEnd -->
<!-- RpFormInput type="text" name="theStreetAddress" Size="64"
      MaxLength="128"
      RpGetType=Function RpGetPtr="getTheStreetAddress"
      RpSetType=function RpSetPtr="setTheStreetAddress" -->
<input type="text" size="64" maxlength="128" name="streetAddress">
<!-- RpEnd -->
<!-- RpDZT -->
<p>
Enter your city:
<!-- RpEnd -->
<!-- RpFormInput type="text" name="theCity" Size="64" MaxLength="128"
      RpGetType=Function RpGetPtr="getTheCity" RpSetType=function
      RpSetPtr="setTheCity" -->
<input type="text" size="64" maxlength="128" name="theCity">
<!-- RpEnd -->
<!-- RpDZT -->
<p>
Enter your state:
<!-- RpEnd -->
<!-- RpFormInput type="text" name="theState" Size="64" MaxLength="128"
      RpGetType=Function RpGetPtr="getTheState" RpSetType=function
      RpSetPtr="setTheState" -->
<input type="text" size="64" maxlength="128" name="theState">
<!-- RpEnd -->
<!-- RpDZT -->
<p>
Enter your zip code:
<!-- RpEnd -->
<!-- RpFormInput type="text" name="theZipCode" Size="64"
      MaxLength="128"
      RpGetType=Function RpGetPtr="getTheZipCode" RpSetType=function
      RpSetPtr="setTheZipCode" -->
<input type="text" size="10" maxlength="10" name="theZipCode">
```

Post processing function

## Handling multiple submit buttons in web applications

```
<!-- RpEnd -->
<!-- RpDZT -->
<p>
<!-- RpEnd -->
<!-- RpFormInput type="submit" name="update" value="update" -->
<input type="submit" name="update" value="update">
<!-- RpEnd -->
<!-- RpFormInput type="submit" name="clearName" value="clearName" -->
<input type="submit" name="clearName" value="clearName">
<!-- RpEnd -->
<!-- RpFormInput type="submit" name="clearStreetAddress"
value="clearStreetAddress" -->
<input type="submit" name="clearStreetAddress"
value="clearStreetAddress">
<!-- RpEnd -->
<!-- RpFormInput type="submit" name="clearZipCode" value="clearZipCode"
-->
<input type="submit" name="clearZipCode" value="clearZipCode">
<!-- RpEnd -->
<!-- RpFormInput type="RESET" name="reset form" value="reset form" -->
<input type="reset" name="reset" value="reset form">
<!-- RpEnd -->
<!-- RpEndForm -->
</form>
<!-- RpEnd -->
</body>
</html>
```



Multiple submit buttons



I have encircled two key areas of the HTML and comment tag code. The first is entitled “Post processing function” The function is entitled “postProcessTheForm” and is associated with the form. This is the function that will be called as a result of clicking any of the submit buttons and after all form fields have been processed. So it is in this function that we will test for which submit button was pressed. The second is entitled “Multiple submit buttons” and shows both the HTML and the pbuilder comment tags for describing the submit buttons.

## 6 The post-processing function

As described in the last chapter, we want a post-processing function, associated with the form, in which we can discern which submit button was pressed. To get there I will digress a bit.

You use the pbuilder utility to convert your HTML (and jpeg, img, JavaScript....etc) files into C code. The Pbuilder utility creates four files, of which two are most important for this discussion. The html page associated with the project that is included with this paper is called web\_page\_multiple\_submits.htm. The pbuilder utility creates two C code files as a result of running against this htm file. One is called web\_page\_multiple\_submits.c. This contains function prototypes and page object definitions. I call this boilerplate. Generally speaking, you will never edit this file and it will be recreated every time you run the pbuilder utility. The other file is called web\_page\_multiple\_submits\_v.c. This file contains functional declarations and you will (generally) edit this file by filling in the stub functions with code to access your device data. So what we want to concentrate on is the code within web\_page\_multiple\_submits\_v.c that differentiates among the submit button clicks.

## Handling multiple submit buttons in web applications

The following code is from the \_v.c file that comes with this paper:

```
// This API is called on the clicking of one of the submit buttons
extern void postProcessTheForm(void *theTaskDataPtr, Signed16Ptr
theIndexValuesPtr);
void postProcessTheForm(void *theTaskDataPtr, Signed16Ptr theIndexValuesPtr) {

    printf("In postProcessTheForm\n");
    // which button was pressed?

    // get the value on the submit button presses
    if(theTaskDataPtr != NULL)
    {
        whichSubmitButton = RpGetSubmitButtonValue(theTaskDataPtr);
    }
    else
    {
        // it is unlikely this will ever be called. It is for defensive
programming only
        printf("danger, AWS not running!\n");
        return;
    }
    printf("The %s submit button was pressed\n", whichSubmitButton);
    // move temp values to perm values as needed
    if(strcmp("update", whichSubmitButton) == 0)
    {
        copyTempToPerm(theTempDataCache, thePermDataCache);
    }

    // clear out the name value in perm structure so the get returns blank
    if(strcmp("clearName", whichSubmitButton) == 0)
    {
        memset(thePermDataCache->theName, '\0', MAX_STRING_LEN);
        thePermDataCache->theNameLen = 0;
    }

    // clear out the address value so the get returns blank
    if(strcmp("clearStreetAddress", whichSubmitButton) == 0)
    {
        memset(thePermDataCache->theStreetAddress, '\0', MAX_STRING_LEN);
        thePermDataCache->theAddressLen = 0;
    }

    // clear out the zip code value so the get returns blank
    if(strcmp("clearZipCode", whichSubmitButton) == 0)
    {
        memset(thePermDataCache->theZipCode, '\0', MAX_STRING_LEN);
        thePermDataCache->theZipCodeLen = 0;
    }

    return;
}
```

To start, variable `whichSubmitButton` is defined as `char * whichSubmitButton;` and is defined globally. The function we are looking at is called `postProcessTheForm`. The formal parameter entitled `theTaskDataPtr` is the `void *` pointer to the AWS internal data structure. Because of this we do not have to call `NAHSGetServerData` as the server data is supplied to us. Notice the call

```
whichSubmitButton = RpGetSubmitValue(theTaskDataPtr);
```

This requests the value from the submit button that was clicked. What is returned is a pointer to `char`. You do not have to `malloc` space for this. Also do not free this as the pointer is owned by the AWS engine. The `strcmp` calls then dispatches the activity based on the string value that was returned by `RpGetSubmitButtonValue`. The last three are straight forward as they clear out a value so that the value returned to the browser is blank (empty string). The first one is more interesting and we will look into it in the next chapter.

The remaining three clear out a field in our device data structure. Thus when the get functions run, the browser page will show an empty field, which is what we asked for with that particular submit button.

## 7 Delaying the write (set) function

As I stated earlier in this document, we have a problem that we must solve. That is the set functions run before the post form processing function runs. So the problem is that I will be updating my device data before I know whether or not I want to update my device data. In our case, three of the submit buttons want to clear a particular field, one per submit button. My problem is that by clicking a submit button I will be sending down data for all fields on the form and thus all fields in my device data.

The solution I have chosen is to use a shadow or temporary device data structure. I have defined one data type as follows:

```
typedef struct _theDataCache
{
    char * theName;
    int theNameLen;
    char * theStreetAddress;
    int theAddressLen;
    char * theCity;
    int theCityLen;
    char * theState;
    int theStateLen;
    char * theZipCode;
    int theZipCodeLen;
} theDataCache;
```

Further I have declared two variables of this type as follows:

## Handling multiple submit buttons in web applications

```
// used before post form processing API is called
theDataCache * theTempDataCache;
// if warranted move data from temp to here
theDataCache * thePermDataCache;
```

The variable called theTempDataCache is updated whenever any submit button is clicked. So when any submit button is clicked, the fields in the variable theTempDataCache are updated. The fields in the variable called thePermDataCache are only updated fully, when the “update” submit button is clicked. Further, for the three other submit buttons that clear a singular field, their code sets that particular field in the variable thePermDataCache to an empty string. Conversely, whenever the “get” functions are executed, they read exclusively from the variable thePermDataCache.

So in the post form processing function entitled postProcessTheForm we find the following code:

```
// move temp values to perm values as needed
if(strcmp("update", whichSubmitButton) == 0)
{
    copyTempToPerm(theTempDataCache, thePermDataCache);
}
```

And the function copyTempToPerm looks like this:

```
// call this API to copy all fields from temp storage to perm storage
// on click of an update submit button
int copyTempToPerm(theDataCache * localTempDataCache, theDataCache *
localPermDataCache)
{
    if((localTempDataCache == NULL) || localPermDataCache == NULL)
    {
        printf("NULL pointer violation - copyTempToPerm\n");
        return -1;
    }
    strcpy(localPermDataCache->theName, localTempDataCache->theName);
    localPermDataCache->theNameLen = localTempDataCache->theNameLen;
    strcpy(localPermDataCache->theStreetAddress, localTempDataCache->
>theStreetAddress);
    localPermDataCache->theAddressLen = localTempDataCache->
>theAddressLen;
    strcpy(localPermDataCache->theCity, localTempDataCache->theCity);
    localPermDataCache->theCityLen = localTempDataCache->theCityLen;
    strcpy(localPermDataCache->theState, localTempDataCache->
>theState);
    localPermDataCache->theStateLen = localTempDataCache->
>theStateLen;
    strcpy(localPermDataCache->theZipCode, localTempDataCache->
>theZipCode);
    localPermDataCache->theZipCodeLen = localTempDataCache->
>theZipCodeLen;
```

```
        return 0;  
    }
```

So, I delay the writing (setting / set functions) to permanent storage until the submit button is processed and I have identified it as the update submit button. Otherwise I discard the contents of the temporary storage. But the set functions have to set something. So they set to temporary storage. To emphasize the point; the temporary storage gets updated to permanent storage when the “update” submit button is clicked otherwise the temporary storage is discarded.

## 8 When are my “get” functions called?

When processing completes in the post form processing function, execution is returned to the web server engine. The web server engine next calls all of my “get” functions. Since my “get” functions read from my permanent storage only, they pick up the updated fields and return the updated data to the browser.

## 9 Conclusion

The average web-based application needs one submit button only. Occasionally, though, a web-based application may need one or more web pages that contain more than one submit button. In this paper we have demonstrated how to build a web page with multiple submit buttons. Further we have demonstrated the comment tags required to support multiple submit buttons. Lastly, we have demonstrated how to deal with the fact that set stub functions are called before the form post processing function is called. By initially setting “temporary” storage and only writing to “permanent” storage when I click the “update” submit button, I can facilitate the processing of multiple submit buttons.

We hope you find this white paper helpful in your designs.

## 10 Appendix

### 10.1 Glossary of terms

- AWS (Advanced Web Server) – a component of the NET+OS development environment that implements an embedded web server
- Browser – A software tool used to communicate with web servers. Examples are MicroSoft Internet Explorer, Google Chrome, Mozilla Firefox, Apple Safari
- Comment Tags – HTML comment-like tags added to HTML files. Used to instruct the pbuilder utility of variables or function calls to generate
- Get function – a function generated by the pbuilder utility. Called by the web server engine for accessing device data to be returned to a web browser
- HTML – hypertext markup language. According to Wikipedia, “the main markup language for creating web pages and other information that can be displayed in a web browser.”

## Handling multiple submit buttons in web applications

- JavaScript – according to Wikipedia, “a dynamic computer language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed”.
- NET+OS – an embedded operating system and embedded development environment developed and sold by Digi International
- Pbuilder – A utility shipped as part of the AWS component of the NET+OS development environment. It is used to convert HTML files and other web-based files into C code for use in C-based applications
- Set Function – a function generated by the pbuilder utility. Called by the web server engine. It takes fields sent from a browser and uses that data to update device data on the device.
- Stub function – a generic term relating to “get” and “set” functions generated by the pbuilder utility. Called “stub” functions because when they are first generated by the pbuilder utility they are stubs. That is, they do not do anything. It is up to the developer to fill in the functionality of the function.
- TCP/IP – A suite of internet protocols used in computers for transferring data from one machine to another. IP, TCP, UDP, FTP, TFTP, SNMP are all examples of internet protocols. According to Wikipedia the set of TCP and IP “make up a reliable, ordered, error-checked delivery of a stream of octets between programs running on computers connected to a local area network”.