

# Dynamic Web Pages With The Embedded Web Server

---

## The Rabbit-Geek's AJAX Workbook (RabbitWeb, XML, & JavaScript)

---

*Version 1.1*

*By*

*Puck Curtis*

*07/29/2009*

## Table of Contents

How to Use this Guide .....	5
Prerequisites – If You Can Ping, You Can Use This Thing!.....	5
Getting Help with TCP/IP and Wi-Fi Setup.....	5
The Study Guide or the Short Cut? .....	5
Dynamic C Code .....	6
HTML Code.....	6
XML File.....	6
Provide us with Your Feedback.....	6
Chapter 1 - The Server-Client Relationship.....	7
Example – An Analogy for a Normal HTML page .....	8
TIP: Auto-refreshing the Page is an Option.....	8
Chapter 2 - Embedded Devices have a limited CPU budget .....	9
Question – How Much Can this Little Guy Handle? .....	9
Answer – Quite a Bit! .....	9
Chapter 3 - Configure your Rabbit’s TCP/IP Settings .....	10
Chapter 4 - Serving a Basic Web Page .....	11
Prep Work – Libraries and Files .....	12
The MIME Table – Everyone Loves a Mime .....	12
The Resource Table – “Mr. File, Meet My Associate, Mr. Web Server.” .....	13
The Main Code .....	13
TIP – Don’t Choke Your Embedded Web Server .....	14
TIP - Using a Costate as a Throttle .....	15
More Web Server Examples.....	15
More Costate Examples .....	15
Chapter 5 – Introducing RabbitWeb .....	16
Chapter 6 - RabbitWeb and the Amazing <?z> Tags.....	17
The Echo.....	17
The Conditional.....	17
The Loop.....	17
Chapter 7 - Using RabbitWeb in a Basic Web Page .....	18
Dynamic C Code .....	18

TIP – You Can Limit the Possible Values.....	19
The HTML Code.....	19
Still Not Dynamic! .....	20
Chapter 8 - Our First Dynamic Web Page .....	21
Updating the Dynamic C code.....	22
Chapter 9 - Using a Conditional – The Mighty IF .....	23
Dynamic C Code .....	23
HTML Code - Two Web Pages in One File .....	24
Tip – Want More? .....	25
Chapter 10 - Interactive Web Pages Made Easy .....	26
Adding a Submit Button.....	26
Updating the Dynamic C code.....	27
Want More? .....	27
Chapter 11 - Detecting User Input from the Web Page.....	28
Adding a Callback Function to the Dynamic C Code .....	28
Other Possibilities? .....	30
Chapter 12 - Creating Loops in RabbitWeb to Display Arrays .....	31
The HTML Code.....	32
TIP - A Sneaky RabbitWeb Trick for Arrays .....	33
Want to Try More?.....	33
Chapter 13 - Adding JavaScript with RabbitWeb to your HTML Pages.....	34
Initializing JavaScript Variables with RabbitWeb .....	34
How to Create the Most Annoying Pop-up Box in History .....	35
The Annoying Pop-up Box.....	35
Saving the Annoying Pop-Up Box.....	36
Chapter 14 - Basic XML with the Rabbit Embedded Web Server .....	39
Dynamic C Code .....	39
Putting together a JavaScript to Read your XML data .....	41
Scary Looking JavaScript Code .....	41
Continued - The HTML body that works with the JavaScript.....	42
Breaking down the Scary Looking Javascript – HTML DOM.....	42
Dynamic C Code .....	45

Let's See the Dog Tags Work! .....	46
Where's the Magic? - AJAX .....	46
Chapter 15 - RabbitWeb and XML – The Holy Grail of Embedded Web Developers.....	47
The Teeny-Tiny and yet Magical my_data.xml file .....	47
Dynamic C code.....	48
The Magic HTML file with Special Kung Fu JavaScript .....	49
The Magic Page in Action! .....	50
Is this Really Magic? .....	51
Chapter 16 - Loopy JavaScript.....	52
The Joy of the setTimeout() Function .....	52
Easy setTimeout() Function Example.....	52
Using the setTimeout() Function to create an Infinite Loop.....	53
Exiting the Loop .....	54
Looking at the Page.....	54
Chapter 17 - Using a JavaScript Loop to Pass Dynamic XML Data in Real-Time .....	55
Dynamic C Code .....	55
The Amazing JavaScript.....	57
The HTML Body .....	58
Watching it All Work .....	58
Want more? .....	60
Chapter 18 - Dynamic Web Interface to Hardware in Real-Time .....	61
A Simple XML file for Our Little Switch .....	62
Dynamic C Code .....	62
The JavaScript .....	63
The HTML Body .....	64
How Does it Look? .....	64
Chapter 19 - Monitoring Every Parallel Port Pin on the MiniCore.....	66

## How to Use this Guide

This Guide is not a user's manual in the traditional sense. Instead, it is an informal workbook for a variety of different embedded web server projects.

### Prerequisites – If You Can Ping, You Can Use This Thing!

This guide is not intended to teach a user basic Internet and TCP/IP networking skills. In order to use the information in this guide, you should be able to configure your Rabbit device for your network so that you can ping it with a PC connected to the same network.

The first test program you should run before starting this guide is the **PINGME.C** sample program located in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\ICMP directory.

### Getting Help with TCP/IP and Wi-Fi Setup

You have plenty of options if you need to get help with basic TCP/IP and Wi-Fi.

1. Rabbit has a guide named **An Introduction to TCP/IP** available here:  
<http://www.rabbit.com/documentation/docs/manuals/TCPIP/Introduction/index.htm>
2. Rabbit has a guide named **An Introduction to Wi-Fi** available here:  
<http://www.rabbit.com/documentation/docs/manuals/WiFi/Introduction/WiFiIntro.pdf>
3. In addition, your Rabbit device's User Manual should have a chapter dedicated to TCP/IP.
4. **All of our most current documentation** is on our web site:  
<http://www.rabbit.com/docs/>

### The Study Guide or the Short Cut?

The chapters build on each other and one good approach would be to work from the beginning of the workbook to the end as a learning experience. Another equally valid approach would be to scan the chapter headings to find something similar to your project and use the example there as a starting point.

### Dynamic C Code

A text box containing code written in Dynamic C will have a gray background.

```
void main()
{
    printf("I am Dynamic C code");
}
```

### HTML Code

A text box containing HTML code will have a light green background.

```
<html>
<head></head>
<body>I am HTML Code</body>
</html>
```

### XML File

A text box containing XML will have a blue background.

```
<!-- My XML file -->
<Stuff>
    <My_File>I am an XML File</ My_File >
</Stuff>
```

### Provide us with Your Feedback

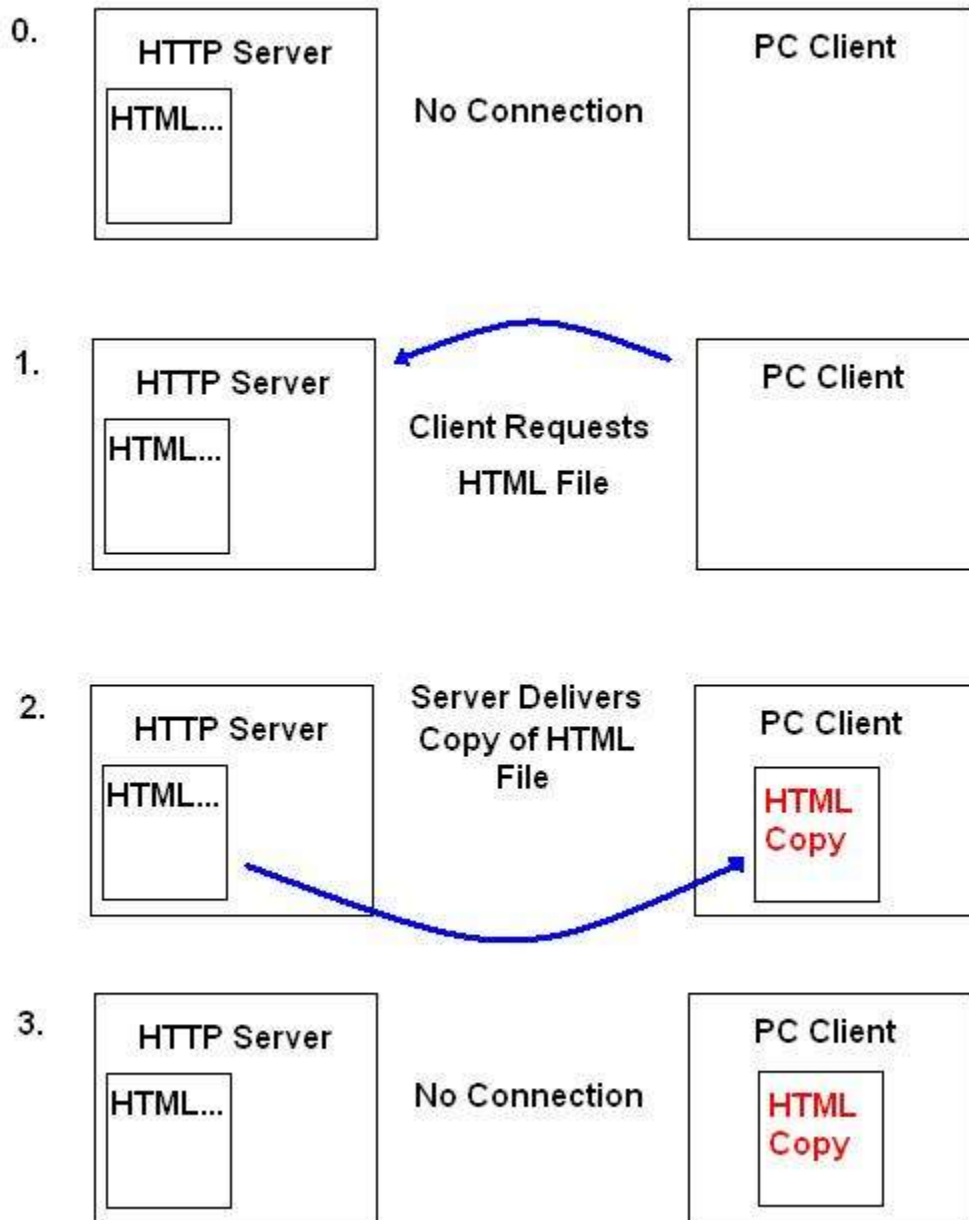
If there is something that:

- Does not work
- Could be explained better
- You think is missing
- You like

Please, let us know by e-mailing [support@rabbit.com](mailto:support@rabbit.com)

## Chapter 1 - The Server-Client Relationship

Web pages are not dynamic in nature. With some clever coding we can produce dynamic behavior, but first we should understand the inherent limitations of the relationship between the HTTP server and the client running a browser like Internet Explorer or Firefox.



The important thing to notice is that there is no continuous connection between the client and the web server. (After the HTML page has been served, we could shoot the server with a shotgun, and the client would not notice because it already has the data.)

### Example – An Analogy for a Normal HTML page

If we pretend that I am the server and you are the client we can recreate this relationship.

0. You are in your car and I am home watching my TV.  
(No connection between client and server.)
1. If you want directions to the pizza parlor, you call me.  
(Client requests data.)
2. I tell you how to get there.  
(Server supplies data.)
3. We hang up.  
(No connection between client and server.)
4. You get there but now you can't find the parking so you call me back.  
(Client makes new request for data.)
5. I tell you where to park.  
(Server supplies data.)
6. We hang up.  
(No connection between client and server.)

The key concept here is that there will be **no fresh data without a new request to the server**.

This is how a web server typically works. An easy way to demonstrate this is to load a web page and then unplug your PC's Ethernet cable. The web page will not notice that it does not have access to the web server anymore because it has already been loaded.

### TIP: Auto-refreshing the Page is an Option

With a bit of HTML code in the head element, you can have the page automatically refresh itself in a given time interval.

```
<meta http-equiv="refresh" content="300" >
```

The value in quotes after **content** is the number of seconds until the page automatically refreshes itself. If you want a different time interval, we could change the value of **content** in our refreshing meta tag. This example would refresh the page every 5 minutes (300 seconds).

In an auto-refreshing example, you would call my phone every 5 minutes for new instructions to the pizza parlor.



## Chapter 2 - Embedded Devices have a limited CPU budget

The PC viewing the web page typically has more resources than the Rabbit Web Server. You can use that to your advantage by pushing the heavy lifting out to the browser. By calling on the PC's memory and CPU, we can have very complicated script, code, or animation and still be very responsive as a web server without overly taxing our embedded device.

You can create very complicated code in the browser using:

- HTML
- Java Applets
- JavaScript
- Flash
- Ajax
- Yahoo User Interface
- Canvas
- Plenty more to boot...

Server-side scripts are not an option. While these might work just fine on a Linux web server, the Rabbit won't be able to run them. Even if it could, the processing power required to handle them might make it a bad choice. A good rule of thumb to remember is that anything that runs on the server side won't work because it would require a special command interpreter from the Rabbit.

You cannot use the following server-side scripting languages or programs:

- Java Servlets
- PHP
- SQL
- Apache
- MediaWiki
- WordPress
- Etc...

### Question – How Much Can this Little Guy Handle?

Can the new Rabbit Wi-Fi Minicore serve interactive web pages with the Rabbit 5000 processor?

### Answer – Quite a Bit!

Rabbit has been serving scripts and animations since the Rabbit 2000 and you can see an example in the C:\DCRABBIT\_9.62\Samples\OP7200\TCPIP\FLASH\_XML.C program.

That flash sample is running on the Rabbit 2000 at 22.1 MHz with 256K flash and 128K SRAM.

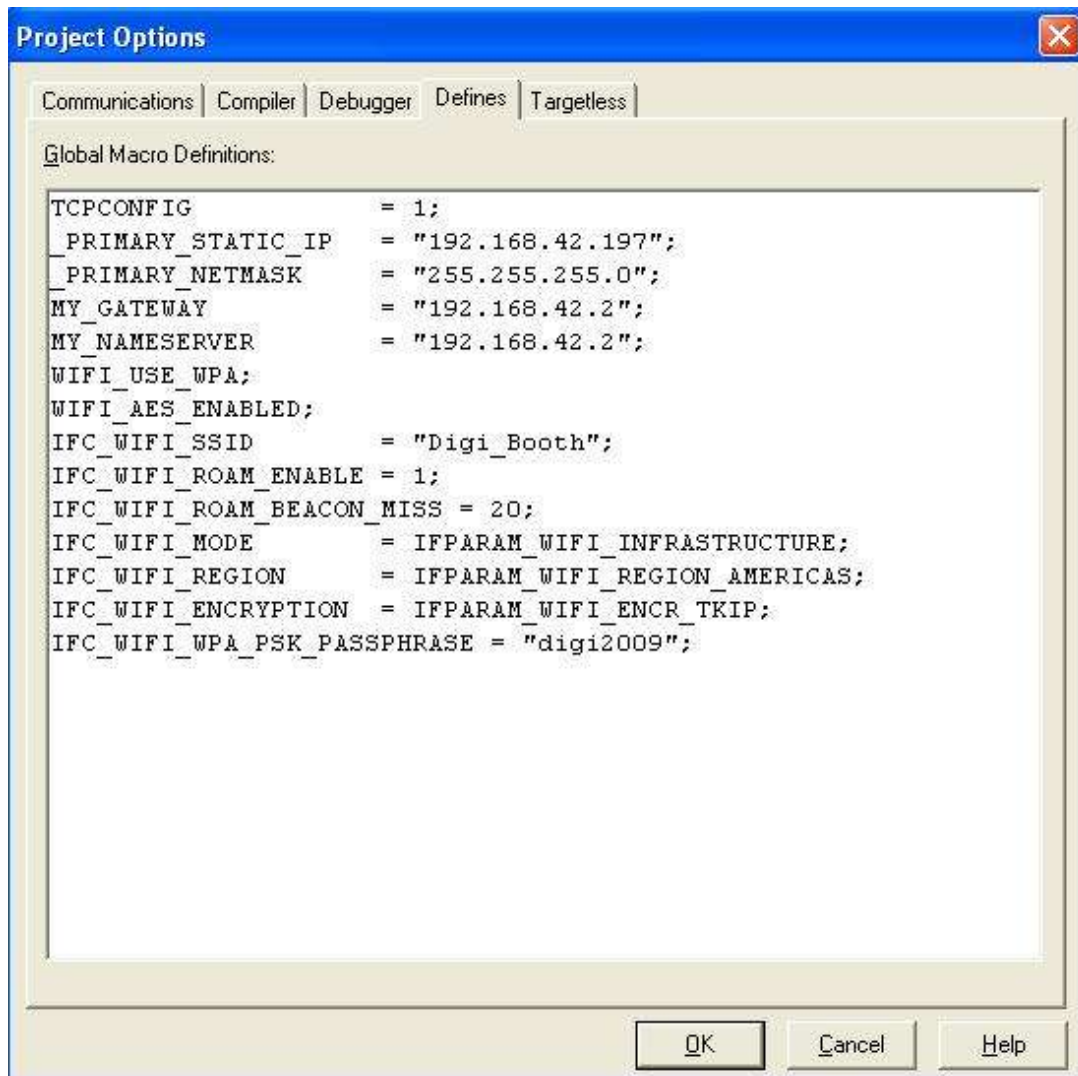
The Wi-Fi Minicore runs at 75 MHz with 1 MB of flash and 1 MB of SRAM and it will **knock the sockets** off the Rabbit 2000.

## Chapter 3 - Configure your Rabbit's TCP/IP Settings

In Dynamic C you can configure all the TCP/IP settings in the main menu under

**Options → Project Options → Defines**

Here is one example used for an RCM5600W Wi-Fi MiniCore:



The settings shown above were used at a tradeshow with an RCM5600W. Your own settings will be different depending on which Rabbit you choose (*Wi-Fi or Ethernet*) and the settings of your local network. Your network administrator should be able to help you configure your device for your network to avoid hidden pitfalls like proxies, blocked ports, and firewalls.

Remember that you should test your settings with the **PINGME.C** sample program in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\ICMP\ directory.

## Chapter 4 - Serving a Basic Web Page

Serving a web page with the Rabbit is very easy.

Here is an example of a simple web page:

```
<html>
<head></head>

<body>
Hello World!
</body>

</html>
```

The page generates the following in the browser:

```
Hello World!
```

If you have configured your TCP/IP settings you can serve the basic web page with a Rabbit device and the following Dynamic C code:

```
#use "dcrtcp.lib"
#use "http.lib"

#ximport "pages/hello_world.html" index_html

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html")
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html)
SSPEC_RESOURCETABLE_END

void main()
{
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
    }
}
```

## Prep Work – Libraries and Files

The two **#use** statements link the software libraries necessary for TCP/IP and HTTP.

```
#use "dcrtcp.lib"  
#use "http.lib"
```

We use the **#ximport** keyword to bring the files we need onto the Rabbit for the web server. During compile time, these files will be copied from the PC running the compiler into the XMEM on the Rabbit.

```
#ximport "pages/hello_world.html" index_html
```

The **#ximport** keyword expects a filename with a path on the PC running the compiler ("**pages/hello\_world.html**") and a local filename which will provide the physical address in XMEM to the program (**index\_html**). Notice that the local filename (or *symbol*) and the filename on the PC don't have to match.

## The MIME Table – Everyone Loves a Mime

Multipurpose Internet Mail Extensions (MIME) is an Internet standard that extends the format of HTTP to support:

- Text in character sets other than ASCII
- Non-text attachments
- Message bodies with multiple parts
- Header information in non-ASCII character sets

The MIME table is where we describe the types of resources our web page will use. This page uses an HTML file, but we could also add other resource types like GIFs, JPEGs, or scripting files. You only need to define the resource type one time and then you may use as many of those items as you like.

For example, with the MIME table below, I could serve as many different HTML pages as I like. If I added support for GIF images, I could have as many of those as I like as well.

```
SSPEC_MIMETABLE_START  
    SSPEC_MIME(".html", "text/html")  
SSPEC_MIMETABLE_END
```

In the example above we pass in two arguments. The first is the file extension like ".html", ".htm", "shmtl", or "zhtml". Then I use an Internet standard MIME type for my html file to show that it is a text file and should be treated as html.

You can find MIME types for all types of Internet files in the MIME specifications. Here are just a few of the file types you might work with.

File Extension	MIME Type	File Extension	MIME Type
.html, htm	text/html	.xml	text/xml
.css	text/css	.js	application/x-javascript
.gif	image/gif	.pdf	application/pdf
.jpeg	image/jpeg	.doc	application/msword

## The Resource Table – “Mr. File, Meet My Associate, Mr. Web Server.”

The Resource table is where we associate names of web server resources with references to items in actual memory. Here we show a file from extended memory identified as “XMEMFILE”. You could pull the file from root memory or a FAT file as well. The Resource Table expects a filename (“/index.html”) and an address in XMEM (**index\_html**).

```
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html)
SSPEC_RESOURCETABLE_END
```

Unlike the MIME table, I need to place every resource I plan on using in my web pages into this table. That means if I use 500 different .GIF files, I need 500 different entries here.

## The Main Code

Looking at the Dynamic C **main code** we only need four lines to get our server going. Most of these are simple initializations to get things up and running.

```
void main()
{
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
    }
}
```

The **sock\_init\_or\_exit()** function initializes the TCP/IP device and attempts to bring it up. This function will print out the IP address of your server at startup if you pass it a non-zero parameter. (In our case, we are passing a “1” so that it displays the network settings when it connects successfully. If the web server doesn’t come up, it will try to exit the program. While this function is good for debugging, it is not a great choice for a final application that may want to handle a failure in some way besides exiting the program.

```
sock_init_or_exit(1);
```

The **http\_init()** function initializes the web server. Reserving port 80 for our embedded web server will yield a performance increase. (The **tcp\_reserveport()** function is completely optional.)

```
http_init();
tcp_reserveport(80);
```

The HTTP handler drives the web server and you should repeatedly call the **http\_handler()** function in a loop. It will also drive the TCP/IP stack which makes the more common `tcp_tick()` function unnecessary. The `tcp_tick()` function drives the TCP stack and just as with the `tcp_tick()` function, the faster you call the `http_handler`, the more responsive your TCP/IP performance will be. (For more information on `tcp_tick()` and programs that use it, read through the Dynamic C TCP/IP User's Guides Volume 1 and Volume 2.)

**http\_handler();**

### TIP – Don't Choke Your Embedded Web Server

If you run into an issue with the web server being slow, you can add a timer to see how long it is between calls to the server. The code shown below is a little debugging trick you can use that will tell you how long your code is taking you between calls to the **http\_handler()** function.

```
void main()
{
    unsigned long time_0; //used to store a time from MS_TIMER
    time_0 = 0;           //initialize to zero

    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while(1)
    {
        printf("time elapsed = %lu\r", (MS_TIMER - time_0));
        http_handler();
        time_0 = MS_TIMER;
    }
}
```

If your web server is not responsive, you may have a function that is blocking for too long and it could be strangling the web server. One way to solve this issue is to add occasional **http\_handler()** function calls into your blocking functions to ensure that the web server is able to keep up.

### TIP - Using a Costate as a Throttle

Another possibility is to use Dynamic C's costates to slow down other tasks in your primary while loop. The main code below shows one possibility.

```
void main()
{
    int j;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler(); //Run as fast as possible!

        costate //Throttle back the C code so we can drive the web server
        {
            waitfor (DelayMs(100));
            printf("j = %d\r", j++);
        }

        costate //Throttle back your function so we can drive the web server
        {
            waitfor (DelayMs(400));
            //Your special function here
        }
    }
}
```

The code shown above uses **costate** cooperative multitasking with the **DelayMS()** function to restrict the **printf()** function so that it only executes once every 100 milliseconds and **your special function** executes every 400 milliseconds. That allows us to control the amount of time we dedicate to our web server.

### More Web Server Examples

See the STATIC.C program in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\HTTP directory for another simple example. This directory is filled with different examples using the Rabbit embedded web server.

### More Costate Examples

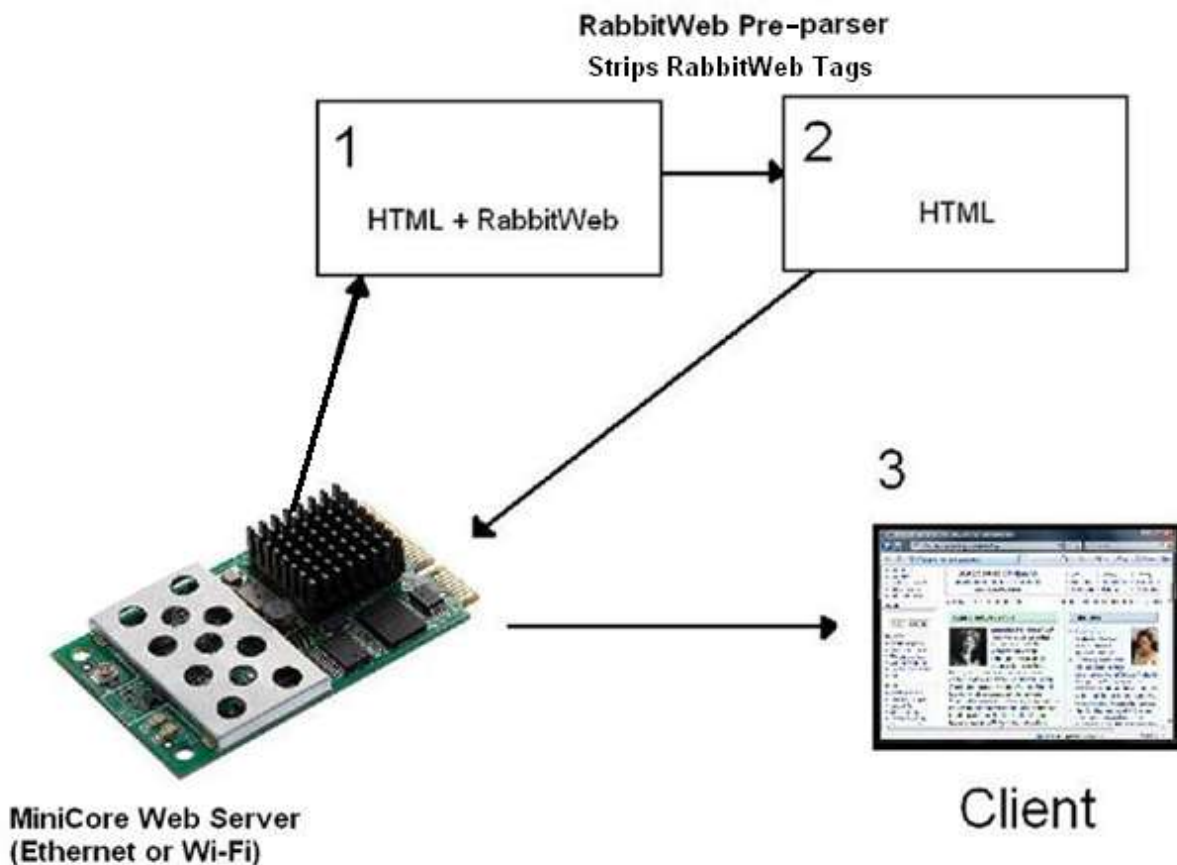
See the COSTATE1.C, COSTATE2.C, and COSTATE3.C example programs in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\HTTP directory for more cooperative multitasking examples.

## Chapter 5 – Introducing RabbitWeb

RabbitWeb takes HTML pages containing RabbitWeb tags and converts them into normal text, html, or data. That text could be HTML code or any other text file. We can use RabbitWeb to make a web browser into an interactive interface for our embedded controller.

RabbitWeb makes displaying data from the Rabbit very easy.

1. When the client requests the HTML file, the Rabbit still has an HTML file with RabbitWeb tags included. When the server receives the request for the HTML file, it passes the file through the RabbitWeb pre-parser.
2. When the RabbitWeb pre-parser is finished, the RabbitWeb tags have been replaced with plain text.
3. The modified version of the file will be served to the client.





## Chapter 6 - RabbitWeb and the Amazing <?z> Tags

The RabbitWeb scripting language has some basic functionality that allows easy integration with the embedded device. Rabbit uses a series of <?z> tags that will be interpreted and replaced before the page is served.

Here are three basic functions of RabbitWeb:

### The Echo

In order to display the value of a variable you will use the **echo** tag.

```
<?z echo($My_Data) ?>
```

This will replace the tag with the value of the My\_Data variable on the Rabbit embedded device.

### The Conditional

In order to test against the value in a variable you will use an **if** statement

```
<?z if ($My_Data == 0){ ?>  
    ALERT! ALERT! ALERT! (Could be any HTML code or text)  
<?z } ?>
```

This can be used to add or remove text from a file depending on the value of a variable on the Rabbit embedded device. In the snippet above, when the My\_Data variable is equal to zero the “**ALERT!**” message will be present in the file. When the My\_Data variable is not equal to zero, the “**ALERT!**” message will be stripped out.

### The Loop

This for loop can be used to repeat common text and is handy for displaying the value of an array.

```
<?z for ($A = 0; $A < 5; $A++) { ?>  
    <?z echo($My_Data[$A]) ?><br>  
<?z } ?>
```

This can be used to display all the elements of an array on the Rabbit device or just to repeat large blocks of text. In the snippet above, the first five elements of the My\_Data array will be displayed in the file. (The <br> tag is normal HTML that indicates a new line on the web page.)

## Chapter 7 - Using RabbitWeb in a Basic Web Page

Dynamic C code doesn't require many changes to use RabbitWeb.

### Dynamic C Code

The code below demonstrates some minor changes for handling RabbitWeb.

```
#define USE_RABBITWEB 1
#include "dcrtcp.lib"
#include "http.lib"

#include "samples/tcpip/http/pages/mypage.zhtml" mypage_zhtml

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler)
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", mypage_zhtml),
    SSPEC_RESOURCE_XMEMFILE("/index.zhtml", mypage_zhtml)
SSPEC_RESOURCETABLE_END

int My_Data; //Create a global variable
#include My_Data //Register it as a Web Variable

void main()
{
    My_Data = 2009;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
    }
}
```

Include the RabbitWeb pre-parser by putting this RabbitWeb macro into your code.

```
#define USE_RABBITWEB 1
```

The change to the MIME table runs the .ZHTML file through the ZHTML handler which are Rabbit functions to replace all the z-tags. Notice that we are using the **SSPEC\_MIME\_FUNC** option instead of **SSPEC\_MIME**.

```
SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler)
SSPEC_MIMETABLE_END
```

You could also run any other text file through the .ZHTML handler using the **SSPEC\_MIME\_FUNC** macro. This includes style sheets, JavaScripts, XML or other text files as well.

We declare the `My_Data` variable globally and then register it using the `#web` statement.

```
int My_Data;    //Create a global variable
#web My_Data    //Register it as a Web Variable
```

### TIP – You Can Limit the Possible Values

You can also place limits on the variable to ensure that it falls within boundaries that you set. If you want your integer between 1 and 1000, you can enforce that limit in the `#web` statement by changing it to:

```
//Restrict My_Data value to between 1 - 1000
#web My_Data (($My_Data > 1) && ($My_Data < 1000))
```

This could prevent your user from entering garbage data into your web interface. You might use this restriction to easily prevent a divide-by-zero error.

### The HTML Code

Here is a simple web page with the RabbitWeb tag included that points back to the `My_Data` variable. (I have placed the RabbitWeb echo tag in **bold green** text.)

```
<html>
<head>
</head>

<body>
My_Data = <z echo($My_Data) ?>
</body>

</html>
```

If the My\_Data integer variable is equal to '2009' on the Rabbit, the RabbitWeb pre-parser will automatically edit the HTML file before it is served to a browser. (I have shown the new data in **bold** text.)

```
<html>
<head></head>

<body>
My_Data = 2009
</body>

</html>
```

The page generates the following in the browser:

```
My_Data = 2009
```

### Still Not Dynamic!

Shouldn't RabbitWeb just make the page dynamic? RabbitWeb does not change the server and client relationship we demonstrated at the beginning of this workbook. RabbitWeb is just a tool for pre-parsing data in the HTML file and it doesn't do anything magical to the HTML file that makes it dynamic.

After the modified HTML page is served to the client, it is stuck with that version of the page until the user refreshes the page. A request for a new page will send the HTML file back through the RabbitWeb pre-parser and that new copy will contain the most current value instead of the RabbitWeb tags.

## Chapter 8 - Our First Dynamic Web Page

In order to show new data, we can include a meta tag to request that the page automatically reload. This has the advantage of being very easy to do with a single line of HTML code in the head element of the HTML file.

Here is the same page with the meta tag in the header asking for an automatic refresh. A meta tag is an HTML tag with “metadata” or “*data about other data*”. The meta tags are placed in the head section of the HTML file.)

When the page is loaded in the browser, the user’s browser will automatically request a refresh from the server every 5 seconds. (You will see the automatic refresh request in **bold** text and the RabbitWeb tag in **bold green** text.)

```
<html>
<head>
  <meta http-equiv="refresh" content="5" >
</head>

<body>
My_Data = <?z echo($My_Data) ?>
</body>

</html>
```

Assuming the My\_Data integer variable starts at ‘2009’, after it passes through the RabbitWeb pre-parser it will look like this:

```
<html>
<head>
  <meta http-equiv="refresh" content="5" >
</head>

<body>
My_Data = 2009
</body>

</html>
```

And the browser will display this:

```
My_Data = 2009
```

The page will still refresh every 5 seconds, and the Rabbit will still send a copy of fresh data even if it has not changed. If we assume that after 5 seconds the value of My\_Data has changed to '1967' the file will change showing the new value as it passes through the RabbitWeb pre-parser. (I have placed the updated data in **bold** text.)

```
<html>
<head>
  <meta http-equiv="refresh" content="5" >
</head>

<body>
My_Data = 1967
</body>

</html>
```

If the value on the Rabbit has changed, when the page refreshes it will display new data like this:

```
My_Data = 1967
```

### Updating the Dynamic C code

If we change the main code in the previous example, we can watch the value of the My\_Data variable change. (I have placed the new line of code in **bold** text.)

```
void main()
{
    My_Data = 2009;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
        My_Data++; //Update the value of My_Data continuously
    }
}
```

This example can work as a very simple dynamic web page for your embedded controller but it is wasteful when we consider the client and server relationship. If the web page is large with JavaScripts, cascading style sheets, and other resources, it is not an efficient way to pass our data.

If your web page is small and simple this is a good solution, but if you only want to pass data there are better ways like XML which we will consider later.

## Chapter 9 - Using a Conditional – The Mighty IF

A lab technician looking at a simple refreshing web page with a changing value might not notice the numbers toggling back and forth. If we could radically change the text in the HTML page we could make an alert more noticeable.

By creating a simple web page we can demonstrate how to display an interesting alert by completely changing the HTML code with RabbitWeb.

### Dynamic C Code

If we tweak the main code again, we can set up our test alert to see how the HTML code will change. The code shown below leaves My\_Data in the “good” state of “1” for 30 seconds. After 30 seconds, the value is toggled to “0” to trigger an alert. Every 30 seconds, the value in My\_Data will alternate between “1” and “0”.

```
void main()
{
    My_Data = 1;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();

        costate //Costate to toggle between 1 and 0 every 30 seconds
        {
            waitfor( DelaySec (30));
            My_Data = My_Data ^ 1; //bitwise XOR My_Data (0 or 1)
        }
    }
}
```

Notice that the web server is driven as fast as possible while I have restricted the execution of the My\_Data toggle to once every 30 seconds by using the DelaySec() function with the costate.

## HTML Code - Two Web Pages in One File

Knowing that the My\_Data variable can equal either “1” or “0” we can use RabbitWeb’s mighty **IF** statement to include or exclude blocks of code. Note that this page includes a refreshing meta tag in the header that will reload the page every 5 seconds. (I have placed the dynamic HTML code in **bold** text and the RabbitWeb code in **bold green** text.)

```
<html>
<head>
<meta http-equiv="refresh" content="5" >
</head>

<body>

    <?z if ($My_Data == 0){ ?>
        ALERT! ALERT! ALERT! (Could be any HTML code)
    <?z } ?>

    <?z if ($My_Data == 1){ ?>
        HAPPY HAPPY JOY JOY!!... (Could be any HTML code)
    <?z } ?>

</body>
</html>
```

In our first operating condition, RabbitWeb will parse the HTML file and the served file will look like this:

```
<html>
<head>
<meta http-equiv="refresh" content="5" >
</head>

<body>
    HAPPY HAPPY JOY JOY!! (Could be any HTML code)
</body>
</html>
```

The browser will display the “Happy” message:

```
HAPPY HAPPY JOY JOY!! (Could be any HTML code)
```



When the alert condition occurs, we have to wait for the client's refresh request to occur. When the client requests the new page, it will be reparsed and the HTML file will look like this:

```
<html>
<head>
<meta http-equiv="refresh" content="5" >
</head>

<body>
    ALERT! ALERT! ALERT! (Could be any HTML code)
</body>
</html>
```

And the page will display the "Alert" message:

```
ALERT! ALERT! ALERT! (Could be any HTML code)
```

### Tip – Want More?

You could also use this with any text file passed through the RabbitWeb pre-parser. This could include JavaScript files and XML files too. With a little creativity, you could create dynamic data files for any situation.

## Chapter 10 - Interactive Web Pages Made Easy

Displaying data is great but in order to use a web page as a user interface, we need the ability to both display and change data from the web page. Fortunately, RabbitWeb makes this very easy as well.

### Adding a Submit Button

Here is a variation on our web page with a **Submit** button added.

By using an HTML **form** element we allow the user to enter information and we send this back to the web server as an HTTP **post** transaction.

```
<form action="/index.zhtml" method="post">
```

The HTML code has an **input** element which is defined as a **text** box. The **value** attribute allows us to display an editable text string automatically when the page loads. The RabbitWeb **echo** tag shown in **green bolded text** copies the current value of the `My_Data` variable into the editable text field.

```
<input type="text" name="My_Data" value="<z echo($My_Data) ?>">
```

We have also defined a submit button as input type and the submit button. You can change the text displayed on the submit button by editing the quoted text after the **value** attribute.

```
<input type="submit" value="Submit">
```

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

<form action="/index.zhtml" method="post">

My_Data = <input type="text" name="My_Data" value="<z echo($My_Data) ?>"> <br>

<input type="submit" value="Submit"> <br>

</form>

</body>
</html>
```

With a submit button on the page any changes will be passed back to the server automatically using a **post** transaction when the user presses **Submit**.



### Updating the Dynamic C code

If we change the main code in the previous example, we can watch the value of the My\_Data variable change in Dynamic C's STDIO window in response to the changes entered by the user on the web page. (I have placed the new line of code in **bold** text.)

```
void main()
{
    My_Data = 2009;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
        printf("My_Data = %d\r", My_Data); //Continously print My_Data's value
    }
}
```

### Want More?

For another examples that demonstrate how to update a variable on the embedded controller from a web interface, see the UPDATING.C example program in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\RabbitWeb directory.

## Chapter 11 - Detecting User Input from the Web Page

With an embedded controller, it isn't unusual to want to trigger an interrupt service routine based on an external feedback. In a similar manner, RabbitWeb allows you to directly call a function when a user changes a value with the web interface.

In order to demonstrate this, we can use the same HTML code with the Submit button as before. (I have left the RabbitWeb tag in **green bold** letters so it is easy to see.)

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

<form action="/index.zhtml" method="post">

My_Data = <input type="text" name="My_Data" value="<b>?z echo($My_Data) ?>"> <br>
<input type="submit" value="Submit"> <br>

</form>

</body>
</html>
```

### Adding a Callback Function to the Dynamic C Code

Our previous code needs to be changed slightly to handle RabbitWeb.

```
#define USE_RABBITWEB 1
#use "dcrtcp.lib"
#use "http.lib"

#ximport "samples/tcpip/http/pages/mypage.zhtml" mypage_zhtml

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler)
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", mypage_zhtml),
    SSPEC_RESOURCE_XMEMFILE("/index.zhtml", mypage_zhtml)
SSPEC_RESOURCETABLE_END

int My_Data; //Create a global variable
#web My_Data //Register it as a Web Variable
```

```

int My_Data_update(void)    //Callback function that prints the change
{
    printf("My_Data changed to %d!\n", My_Data);
}
#web_update My_Data My_Data_update

void main()
{
    My_Data = 2009;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
    }
}

```

Just as before we create a global variable and register it with the web server.

```

int My_Data;    //Create a global variable
#web My_Data    //Register it as a Web Variable

```

Then, we create a function we would like to call when the web registered variable changes. This function is called a **callback function**. This function is simple, but it could be anything you like.

```

int My_Data_update(void)    //Callback function that prints the change
{
    printf("My_Data changed to %d!\n", My_Data);
}

```

Finally, we register our callback function to tie it to the state of the variable. This will cause our callback function to be called when the value changes.

```

#web_update My_Data My_Data_update

```

Here is a snapshot of the callback function printing in Dynamic C's STDIO window:

A screenshot of the Dynamic C STDIO window. The window has a blue title bar with the text "Stdio". The main area is white and contains three lines of text: "Network default interface up at IP=192.168.1.197 mask=255.255.255.0", "My\_Data changed to 2008!", and "My\_Data changed to 2009!".

```
Stdio
Network default interface up at IP=192.168.1.197 mask=255.255.255.0
My_Data changed to 2008!
My_Data changed to 2009!
```

### Other Possibilities?

See the WEB\_UPDATE.C sample program in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\RabbitWeb directory for another example. Remember that you can create complicated behavior with your callback functions. When the client changes a value, the callback function could:

- Set a flag to trigger some behavior
- Send data from a serial port, TCP socket, or UDP socket
- Change the state of a digital output
- Set a pin as either a digital input or an output
- Anything else the Rabbit could do with a normal function call...

## Chapter 12 - Creating Loops in RabbitWeb to Display Arrays

We can create a web registered array in our code like this:

```
#define USE_RABBITWEB 1
#use "dcrtcp.lib"
#use "http.lib"

#ximport "samples/tcpip/http/pages/mypage.zhtml" mypage_zhtml

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler)
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", mypage_zhtml),
    SSPEC_RESOURCE_XMEMFILE("/index.zhtml", mypage_zhtml)
SSPEC_RESOURCETABLE_END

int My_Data[5];           //Create a global array
#web My_Data              //Register entire arrays as an array of web variables
//#web My_Data[0]         //Registers only element 0 in the array

void main()
{
    My_Data[0] = 2009;
    My_Data[1] = 2010;
    My_Data[2] = 2011;
    My_Data[3] = 2012;
    My_Data[4] = 2013;

    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);
    while (1)
    {
        http_handler();
    }
}
```

You can register the entire array in one statement by using the array name.

```
#web My_Data //Register the entire array of web variables
```

If you prefer you can register individual elements in the array instead.

```
#web My_Data[0] //Registers only element 0 in the array
```

## The HTML Code

You have multiple options for displaying the values in your array. This example using an HTML table will work, but requires more code on your part. (Only the **bold green** text is related to RabbitWeb.)

```
<html>
<head>
  <meta http-equiv="refresh" content="5" >
</head>
<body>
<table border="1">
<tr>
  <td><?z echo($My_Data[0]) ?> </td>
  <td><?z echo($My_Data[1]) ?> </td>
  <td><?z echo($My_Data[2]) ?> </td>
  <td><?z echo($My_Data[3]) ?> </td>
  <td><?z echo($My_Data[4]) ?> </td>
</tr>
</table>
</body>
</html>
```

You could also use a RabbitWeb **for** loop to display the My\_Data array. RabbitWeb scripting includes a series of canned loop-control indexes you can use in **for** loops like the one above. These cover the range of 'A' to 'Z' and assume an unsigned integer value.

```
<html>
<head>
  <meta http-equiv="refresh" content="5" >
</head>
<body>
<table border="1">
<tr>
  <?z for ($A = 0; $A < 5; $A++) { ?>
    <td><?z echo($My_Data[$A]) ?> </td>
  <?z } ?>
</tr>
</table>
</body>
</html>
```

These two pages are functionally identical and both will refresh the page every five seconds as well using the refreshing meta tag we discussed earlier.



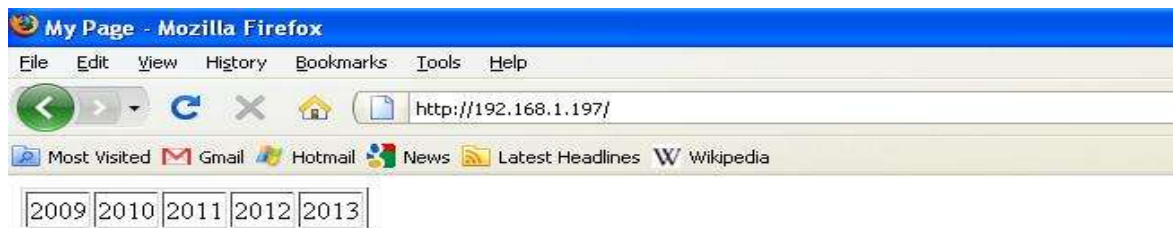
### TIP - A Sneaky RabbitWeb Trick for Arrays

Another advantage is that even if I don't know the array size, RabbitWeb can detect it automatically by using the **count()** RabbitWeb scripting function in my **for** loop. The code shown below will automatically loop through any size array when parsing the file. (The <td> tags are part of the HTML table.)

```
<?z for ($A = 0; $A < count($My_Data, 0); $A++) { ?>
    <td><?z echo($My_Data[$A]) ?> </td>
<?z } ?>
```

I have set the second parameter of **count()** to '0' because My\_Data is a one dimensional array. What if My\_Data was a multi-dimensional array like My\_Data[5][5][5]? If I wanted to get to my second dimension I would pass a '1' and a '2' to get to the third dimension into **count()** instead of the '0'.

Here is the HTML page created by the RabbitWeb loop and it will be identical for all the examples I described above.



### Want to Try More?

This is a good example of how you can list a large number of values from the controller very easily, however you can also combine this with a conditional **if** statements to produce different effects. For example, you can set the background of each table cell to green when the My\_Data value is good and change it to red when there is an alarm.

Take a look at the ARRAYS.C program in the C:\DCRABBIT\_XX.XX\Samples\TCPIP\RabbitWeb directory for another example using arrays.

## Chapter 13 - Adding JavaScript with RabbitWeb to your HTML Pages

It is important to remember that RabbitWeb is not restricted to parsing HTML code and it can also be useful when working with JavaScript.

### Initializing JavaScript Variables with RabbitWeb

This Javascript shown below doesn't do anything but initialize variables; however it does provide a useful demonstration. (JavaScript items are in **bold black** text and the RabbitWeb is in **bold green** text.)

```
<html>
<head>
<script type="application/x-javascript">
function my_script()
{
    var Analog_input1 = <?z echo($My_Data[0]) ?>;
    var Analog_input2 = <?z echo($My_Data[1]) ?>;
    var Analog_input2 = <?z echo($My_Data[2]) ?>;
    var Analog_input3 = <?z echo($My_Data[3]) ?>;
    var Analog_input4 = <?z echo($My_Data[4]) ?>;
}
</script>
</head>
<body onload="my_script();">
    Some HTML CODE
</body>
</html>
```

If we use the same Dynamic C program as before, the RabbitWeb pre-parser would create this new file with a changed JavaScript function. (I have shown the changes to the JavaScript in **bold** text.)

```
<html>
<head>
<script type="application/x-javascript">
function my_script()
{
    var Analog_input1 = 2009;
    var Analog_input2 = 2010;
    var Analog_input2 = 2011;
    var Analog_input3 = 2012;
    var Analog_input4 = 2013;
}
</script>
</head>
<body onload="my_script();">
    Some HTML CODE
</body>
</html>
```

## How to Create the Most Annoying Pop-up Box in History

You can combine RabbitWeb with Javascript code to make a pop-up box that is shown every time the page loads. Because we have a refreshing meta tag in the HTML head element, this pop-up will reappear every 5 seconds.

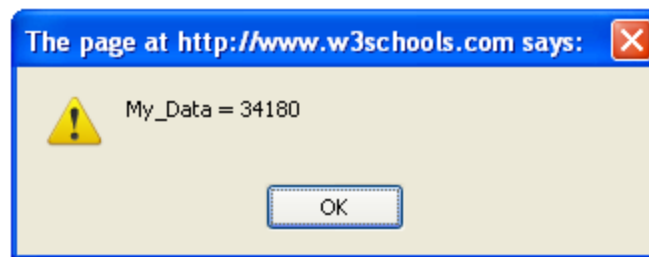
Why is this interesting? Our alert box is showing the value of our My\_Data variable. (JavaScript items are in **bold** text and the RabbitWeb in **bold green** text.)

```
<html>
<head>
<meta http-equiv="refresh" content="5" >
<script type="application/x-javascript">

function disp_alert()
{
    alert("My_Data = <?z echo($My_Data) ?>");
}

</script>
</head>
<body onload="disp_alert();">
    Some HTML code here...
</body>
</html>
```

## The Annoying Pop-up Box



This Pop-up is guaranteed to be terribly annoying, but it is also terribly interesting to see data from the Rabbit appear in the alert box of your browser. With a little more work, we might be able to make this more useful.

## Saving the Annoying Pop-Up Box

Using the RabbitWeb **if** statement you can dramatically change the behavior of a web page very easily. You can add or remove large chunks of code based on the value of a variable on the embedded controller.

If I reuse this Dynamic C main code from our previous example to toggle the value of My\_Data, we should be able to trigger my alert pop-up box.

```
void main()
{
    My_Data = 0;
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        costate          //Costate to drive the web server
        {
            http_handler();
        }

        costate          //Costate to toggle between 1 and 0 every 30 seconds
        {
            waitfor( DelaySec (30));
            My_Data = My_Data ^ 1;      //bitwise XOR My_Data
        }
    }
}
```

Then I can add a simple **if** statement to my Javascript and trigger the alert based on the value of my RabbitWeb variable. (JavaScript items are in **bold** text and the RabbitWeb in **bold green** text.)

```
html>
<head>
<meta http-equiv="refresh" content="5" >
<script type="application/x-javascript">

function disp_alert()
{
    if ( <?z echo($My_Data) ?> );
    {
        alert("My_Data = <?z echo($My_Data) ?>");
    }
}

</script>
</head>
<body onload=" disp_alert();">
    Some HTML Code Here...
</body>
</html>
```

As the page refreshes every 5 seconds, RabbitWeb will insert the value of the My\_Data variable into the file creating either this non-alerting Javascript which hits an “if (0)” and does nothing. (Changes are in **bold** text.)

```
html>
<head>
<meta http-equiv="refresh" content="5" >
<script type="application/x-javascript">

function disp_alert()
{
    if (0);
    {
        alert("My_Data = 0");
    }
}

</script>
</head>
<body onload=" disp_alert();">
    Some HTML Code Here...
</body>
</html>
```

Or this alerting Javascript which hits an “if (1)” and creates the pop-up box:

```
html>
<head>
<meta http-equiv="refresh" content="5" >
<script type="application/javascript">

function disp_alert()
{
    if (1);
    {
        alert("My_Data = 1");
    }
}

</script>
</head>
<body onload=" disp_alert();">
    Some HTML Code Here...
</body>
</html>
```

With just a bit of imagination, you can do almost anything.

## Chapter 14 - Basic XML with the Rabbit Embedded Web Server

XML is a method of transmitting and presenting data in an organized way. It uses XML tags the same way you or I might use the handles on a suitcase to grab onto it. If you know the XML tag, you can get to the data very easily and the best thing is you will create your own tags for your file. (I have put the data in **bold** text.)

For example:

```
<!-- My XML file -->
<Famous_Animals>
  <Dogs>
    <Scoobie_Doo>Great Dane</Scoobie_Doo>
    <Lassie>Collie</Lassie>
    <Rin_Tin_Tin>German Sheppard</Rin_Tin_Tin>
    <Stimpy>Chihuahua</Stimpy>
  </Dogs>
</Famous_Animals>
```

Each defined tag must have a closing tag with a "/" like this:

```
<my_nifty_xml_tag>some nifty data</my_nifty_xml_tag>
```

With this understanding, we can add an XML data file into our embedded web server by editing the MIME table and the Resource table. Note that we have also imported the **dogs.xml** file into extended memory with the **#ximport** keyword. (I have placed the changes in **bold** text.)

### Dynamic C Code

```
#class auto

#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"

#ximport "C:/Pux Dyn C Files/Webinar/dogs.xml"          dogs_xml

SSPEC_MIMETABLE_START
    SSPEC_MIME(".xml","text/xml")                        //Here is my XML MIME type!
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/dogs.xml", dogs_xml)      //Here is my XML file!
SSPEC_RESOURCETABLE_END
```

This Dynamic C code doesn't do anything but serve the XML file.

```
void main()
{
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();
    }
}
```

If we assume my web server has an IP address of "192.168.1.197", I should be able to access the XML file from a browser by typing the following URL into my browser:

<http://192.168.1.197/dogs.xml>



```
<!-- My XML file -->
- <Famous_Animals>
  - <Dogs>
    <Scoobie_Doo>Great Dane</Scoobie_Doo>
    <Lassie>Collie</Lassie>
    <Rin_Tin_Tin>German Sheppard</Rin_Tin_Tin>
    <Stimpy>Chihuahua</Stimpy>
  </Dogs>
</Famous_Animals>
```

This demonstrates that the Rabbit can serve up an XML file in a way that makes some sense to the browser. Now we need a web page that can do something with it.



## Putting together a JavaScript to Read your XML data

Different browsers will handle an XML file in different ways, but the process will break down into four basic steps:

1. Create an XML document.
2. Do you want your read to be synchronous or asynchronous?
3. Load the XML file.
4. Grab a specific piece of data from the file.

## Scary Looking JavaScript Code

I have put together a small HTML file with Javascript code to access the dogs.xml file. This may look intimidating so we're going to break it down piece by piece. (You will notice the **XML tag** in **bold blue** text and the **ID attribute** referenced in **bold red** text.)

```
<html>
<head>
<script type="text/javascript">
function Get_XML_Dog()
{
    var Breed; //Store the Breed of the dog here
    try      //Test for Internet Explorer
    {
        xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    }
    catch(err)    //If ActiveX isn't going to work, so we use a more codified standard
    {
        try      //Every browser but Internet Explorer
        {
            xmlDoc=document.implementation.createDocument("", "", null);
        }
        catch(err) //If that doesn't work, we produce an error mocking our user.
        {
            alert("No XML for you!");
            return;
        }
    }
    xmlDoc.async=false;           //We want a synchronous connection
    xmlDoc.load("dogs.xml");       //Load the XML file

    //Get the XML data from Scoobie_Doo tag and stash it in the Breed variable
    Breed = xmlDoc.getElementsByTagName("Scoobie_Doo")[0].childNodes[0].nodeValue;

    //Write the contents of Breed at the span tag with the id="Dog_Breed"
    document.getElementById("Dog_Breed").innerHTML=Breed;
}
</script>
</head>
```

## Continued - The HTML body that works with the JavaScript

```
<body>
<b>What Kind of Dog is Scoobie Doo?</b>

<p>Scoobie Doo is a <span id="Dog_Breed"></span></p>

<p><button onclick="Get_XML_Dog()">Snag an XML Dog Tag!</button></p>

</body>
</html>
```

## Breaking down the Scary Looking Javascript – HTML DOM

First, we are going to store the breed of the dog in a Javascript variable. If we wanted to manipulate the data with our Javascript, this is useful. If we just want to output the data, we might not need this variable.

```
var Breed; //Store the Breed of the dog here
```

In this document, I am going to treat the complicated topic of the HTML Document Object Model (DOM) as we would a hammer. I'm going to show you a basic method of using these tools and I'm not going to get into extensive detail about what makes them work. If we use the hammer analogy, we are interested in hitting our nails into a board, but we don't need to know equations for kinetic energy or how Newtonian physics explains a lever.

If you want more information on HTML DOM, you can find a tutorial here:

<http://www.w3schools.com/html/dom/default.asp>

We need to have a plan for handling different web browsers. As usual, Microsoft's Internet Explorer wants to go off in its own direction so we can look at their case first. If we are using Internet Explorer, our xmlDoc will be created by this code. If we are not using Internet Explorer it will produce an error.

```
try    //Test for Internet Explorer
{
    xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
}
```

The error will be handled by our **catch** statement. Inside the catch statement, we **try** creating the xmlDoc again using a more common method. If that doesn't work, it will throw another error and we hand it off to another **catch** statement. In the final **catch** statement, we mock people using outdated browsers with a message that taunts them.

```
catch(err)    //If ActiveX isn't going to work, so we use a more codified standard
{
    try        //Every browser but Internet Explorer
    {
        xmlDoc=document.implementation.createDocument("", "", null);
    }
    catch(err) //If that doesn't work, we produce an error mocking our user.
    {
        alert("No XML for you!");
        return;
    }
}
```

If your XML file is very large, you may want the code in the Javascript to wait until the XML file is completely loaded. You can make the Javascript wait for the file to load by specifying that you don't want an asynchronous load.

```
xmlDoc.async=false;    //We want a synchronous connection
```

Now you can load the XML file into your xmlDoc object and the Javascript will wait for you to finish.

```
xmlDoc.load("dogs.xml");    //Load the XML file
```

With the XML file loaded, we can use the XML tag to latch onto a key piece of data by specifying the tag name. (I have placed the unique XML tag in **bold blue** text to make it easy to spot.)

```
//Get the XML data from Scoobie_Doo tag and stash it in the Breed variable
Breed = xmlDoc.getElementsByTagName("Scoobie_Doo")[0].childNodes[0].nodeValue;
```

An HTML element can have **attributes**. You can recognize an attribute because it will be in quotes. Here you can see an HTML paragraph with a unique ID. You can add an ID attribute to any HTML element. (I have placed the ID attribute in **bold red** text to make it easy to see.)

```
<p id="my_unique_paragraph_id">Some Stuff</p>
```

Attributes allow us to provide additional information about the element and we can use these as handles for plugging things into our page with XML. In the HTML code for handling our XML file, I have included an ID.

```
<span id="Dog_Breed"></span>
```

If I wanted to directly set the value between my two span tags, I could use the following Javascript statement:

```
document.getElementById("Dog_Breed").innerHTML="Some Dog";
```

By using our **Breed** Javascript variable, we can set this value with data from the XML file.

```
//Write the contents of Breed at the span tag with the id="Dog_Breed"  
//(See the HTML body below.)  
document.getElementById("Dog_Breed").innerHTML=Breed;
```

We could skip the **Breed** variable completely and use a single line of code to assign the XML data to the ID attribute just as easily if we wished. (I apologize for the small font.)

```
document.getElementById("Dog_Breed").innerHTML=xmlDoc.getElementsByTagName("Scoobie_Doo")[0].childNodes[0].nodeValue;
```

I chose to break this single line up, because it's easier to read and explain if we take it in smaller steps.

In the body of the HTML there are two lines that are important. The first is our **span** element with its ID attribute.

```
<p>Scoobie Doo is a <span id="Dog_Breed"></span></p>
```

The last key piece is HTML code to add a button. When the button is clicked, our Javascript function **Get\_XML\_Dog()** will execute.

```
<p><button onclick="Get_XML_Dog()">Snag an XML Dog Tag!</button></p>
```

## Dynamic C Code

The only real changes to the Dynamic C code are the additions of the HTML file.

```
#class auto

#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"

#import "C:/Pux Dyn C Files/Webinar/basic_xml.html"      index_html
#import "C:/Pux Dyn C Files/Webinar/dogs.xml"            dogs_xml

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".xml", "text/xml")                        //Here is my XML MIME type!
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", index_html),
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/dogs.xml", dogs_xml)        //Here is my XML file!
SSPEC_RESOURCETABLE_END

void main()
{
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();
    }
}
```

## Let's See the Dog Tags Work!

When the page is loaded initially the browser will display this text:



When the user clicks the button the **Get\_XML\_Dog()** Javascript function will execute and our magic XML code will produce this:



## Where's the Magic? - AJAX

The important thing to notice is that all this occurred without a page refresh! This is a method of web programming known as Asynchronous JavaScript and XML or AJAX.

## Chapter 15 - RabbitWeb and XML – The Holy Grail of Embedded Web Developers

Any text file can be passed through the RabbitWeb pre-parser and an XML file is no exception. With XML and RabbitWeb we can create a powerful combination.

As an example, our embedded web server might have the following web resources that it needs to serve to a browser:

- HTML file - 500k
- 3 jpegs - 250k each
- JavaScript file - 100k
- Style sheet - 50k
- XML file - 1k

We want a dynamic page and we could use this HTML code in the <head> to force the entire page to refresh every 5 seconds.

```
<meta http-equiv="refresh" content="5" >
```

Unfortunately, in this case a single page refresh **requires 1401k of data!**

**That's not efficient at all!**

If we use the previous option of manually refreshing the entire page every 5 seconds with a meta tag we place a serious burden on the web server. If we could refresh only the XML file, we create the ability to update only the data in the web page without refreshing the entire page.

### The Teeny-Tiny and yet Magical my\_data.xml file

Consider this simple XML file using a RabbitWeb **echo** tag. (I have placed the RabbitWeb tag in **bold green** text.)

```
<!-- RabbitWeb XML file -->
<RabbitWeb>
  <My_Data1> <?z echo($My_Data1) ?> </My_Data1>
</RabbitWeb>
```

This file is very small and our embedded web server can serve this up much more quickly than the entire page with all the supporting files. Best of all, the HTML pages do not completely reload and the data will change before our eyes like magic!

## Dynamic C code

This Dynamic C code will serve up our HTML file and our RabbitWeb-enabled XML file. Notice that we are using the **SSPEC\_MIME\_FUNC** type for our MIME table to pass the XML file through the RabbitWeb pre-parser. We are also continuously changing the value of the My\_Data variable so we can see a change on our page.

```
#class auto

#define USE_RABBITWEB 1

#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"

#import "C:/Pux Dyn C Files/Webinar/Dynamic_W01.html"      index_html
#import "C:/Pux Dyn C Files/Webinar/my_data.xml"           my_data_xml

int      My_Data;
#web My_Data

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME_FUNC(".xml", "text/xml", zhtml_handler)
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", index_html),
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/my_data.xml", my_data_xml)
SSPEC_RESOURCETABLE_END

void main()
{
    My_Data1 = 1;

    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();
        My_Data++;    //Keep changing My_Data
    }
}
```



## The Magic HTML file with Special Kung Fu JavaScript

This looks almost the same as the previous HTML file. (You will notice the **XML tag** in **bold blue** text and the **ID attribute** referenced in **bold red** text.)

```
<html>
<head>
<script type="text/javascript">
function Read_XML_File()
{
    var XML_Data; //Storage for the value of My_Data from the XML file
    try           //Test for Internet Explorer
    {
        xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    }
    catch(err)     //If ActiveX isn't going to work we use a more codified standard
    {
        try       //Every browser but Internet Explorer
        {
            xmlDoc=document.implementation.createDocument("", "", null);
        }
        catch(err) //If that doesn't work, we produce an error mocking our user.
        {
            alert("No XML for you!");
            return;
        }
    }
    xmlDoc.async=false;           //We want a synchronous connection
    xmlDoc.load("my_data.xml");   //Load the XML file

    //Store the value of My_Data tag in the XML_Data variable
    XML_Data = xmlDoc.getElementsByTagName("My_Data")[0].childNodes[0].nodeValue;

    //Write the value of XML_Data into the body
    document.getElementById("My_Data_ID").innerHTML=XML_Data;
}
</script>
</head>

<body>
<p><b>My_Data1:</b><span id="My_Data_ID"></span></p>

<button onclick="Read_XML_File()">Get XML</button>

</body>
</html>
```

Clicking on the button will execute the **Read\_XML\_File()** JavaScript function which requests a fresh copy of the XML file every time the user clicks the “**Get XML**” button. Unlike our previous example, the XML file will be run through the RabbitWeb pre-parser each time the client requests the file and as the value of the My\_Data variable changes, so will the XML file.

Every time the user clicks the button, the data will be updated!

### The Magic Page in Action!

Here is the page before the button is pressed:



Here is the page after the button is pressed the first time without any page refresh:



Here is the page after the button is pressed the second time without any page refresh:



My\_Data1: 8101



### Is this Really Magic?

Each click of the button requests a new copy of the XML file which is updated by RabbitWeb. It's fast and effective! Can we make it more interesting?

## Chapter 16 - Loopy JavaScript

Updating the data on a button press is interesting, but it would be more impressive to continually refresh the data behind the scenes with an infinite loop. JavaScript has a mechanism for this called a timing event.

### The Joy of the `setTimeout()` Function

The **`setTimeout()`** function can be used to execute a JavaScript after an interval has expired. It accepts two parameters:

1. A JavaScript statement
2. A time interval in milliseconds

The function returns a unique ID number that will allow you to shut down the timed process. You can think of this as a way of putting a C-code style **`break`** into your infinite loop as an escape hatch.

A call to the `setTimeout()` might look like this:

```
var t = setTimeout("c=c+1;",5000);    //increment c in 5 seconds
```

### Easy `setTimeout()` Function Example

Here is a short HTML file that uses the `setTimeout()` Function to display an alert box 5 seconds after the user presses a button on the web page.

```
<html>
<head>
<script type="text/javascript">
function Delayed_Pop_Up()
{
    var timer_id = setTimeout("alert('TaDa!')",5000);
}
</script>
</head>

<body>
<button onclick="Delayed_Pop_Up ()">Trigger Delayed Pop Up</button>
</body>
</html>
```

### Using the setTimeout() Function to create an Infinite Loop

A more interesting construction would be to use the setTimeout() to call another function in an infinite loop. Here we are using the same ID attribute from the previous examples as a mechanism for creating an infinite loop that continues to update the value of the c variable as displayed on the web page twice per second.

```
<html>
<head>
<script type="text/javascript">

var c = 0;      //Simple counter
var timer_id;  //Global timeout ID so we can exit later if we wish

function Inc_C()
{
    c = c + 1;
    document.getElementById("c_variable").innerHTML=c;
    timer_id = setTimeout("Inc_C()",500); //Calling the function again to loop!
}

</script>
</head>

<body>

<p>c = <span id="c_variable"></span></p>
<p><button onclick="Inc_C()">Start C up!</button></p>

</body>
</html>
```

## Exiting the Loop

If we wanted to exit the infinite loop, we could use the `clearTimeout()` function like this:

```
clearTimeout(timer_id);
```

For example, we could create another button that exits the loop. (I have placed the code used to exit the loop in bold text below.)

```
<html>
<head>
<script type="text/javascript">
var c = 0;          //Simple counter
var timer_id;      //Global timeout ID so we can exit later if we wish
function Inc_C()
{
    document.getElementById("c_variable").innerHTML=c;
    c = c +1;
    timer_id = setTimeout("Inc_C()",500); //Calling the function again to loop!
}

function Done()
{
    clearTimeout(timer_id);
}
</script>
</head>

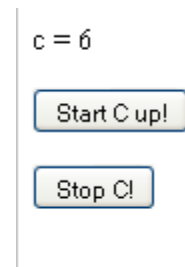
<body>

<p>c = <span id="c_variable"></span></p>
<p><button onclick="Inc_C()">Start C up!</button></p>
<p><button onclick="Done()">Stop C!</button></p>

</body>
</html>
```

## Looking at the Page

Here is a screenshot of how the page will look:



## Chapter 17 - Using a JavaScript Loop to Pass Dynamic XML Data in Real-Time

Now that we understand how to start and stop loops in JavaScript, we can apply this to our Dynamic XML files.

Here is a more interesting XML file. Each request for the file will automatically pass it through the RabbitWeb pre-parser. (I have placed the RabbitWeb tags in **bold green** text.)

```
<!-- RabbitWeb XML file -->
<RabbitWeb>
    <My_Data1> <?z echo($My_Data1) ?> </My_Data1>
    <My_Data2> <?z echo($My_Data2) ?> </My_Data2>
    <My_Data3> <?z echo($My_Data3) ?> </My_Data3>
    <My_Data4> <?z echo($My_Data4) ?> </My_Data4>
</RabbitWeb>
```

### Dynamic C Code

Here is the pre-main code:

```
#class auto

#define USE_RABBITWEB 1

#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"

#ximport "C:/Pux Dyn C Files/Webinar/Dynamic_W02.html"          index_html
#ximport "C:/Pux Dyn C Files/Webinar/my_data2.xml"              my_data2_xml

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME_FUNC(".xml", "text/xml", zhtml_handler)
SSPEC_MIMETABLE_END

int      My_Data1, My_Data2, My_Data3, My_Data4; //Create Global Variables

#web My_Data1 //Register Global Variables With RabbitWeb
#web My_Data2
#web My_Data3
#web My_Data4

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", index_html),
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/my_data2.xml", my_data2_xml)
SSPEC_RESOURCETABLE_END
```

And the main code:

```
void main()
{
    My_Data1 = 1;
    My_Data2 = 10;
    My_Data3 = 100;
    My_Data4 = 1000;

    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();

        costate //Only update the data every 500 ms
        {
            waitfor(DelayMs(500));
            My_Data1++;
            My_Data2++;
            My_Data3++;
            My_Data4++;
        }
    }
}
```

This code doesn't do much that is different from our previous examples with the exception that it has four variables and we are only updating the My\_Data variables every 500 ms.

Our goal is to request a refreshed version of this file from the embedded web server in an infinite loop. We can use the **setTimeout()** function to achieve this goal by combining it with the XML code we have already used.



## The Amazing JavaScript

I have created four buffer variables to make the code easier to read. In practice, I would set the id with the tag in a single line without using a buffer variable. (The looping statement is shown in **bold** text.)

```
<html>
<head>
<script type="text/javascript">

var My_Data1_var;
var My_Data2_var;
var My_Data3_var;
var My_Data4_var;

function Get_XML()
{
    try          //Test for Internet Explorer
    {
        xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    }
    catch(err)    //If ActiveX isn't going to work we use a more codified standard
    {
        try      //Every browser but Internet Explorer
        {
            xmlDoc=document.implementation.createDocument("", "", null);
        }
        catch(err) //If that doesn't work, we produce an error mocking our user.
        {
            alert("No XML for you!");
            return;
        }
    }
    xmlDoc.async=false;
    xmlDoc.load("my_data2.xml");

    My_Data1_var = xmlDoc.getElementsByTagName("My_Data1")[0].childNodes[0].nodeValue;
    My_Data2_var = xmlDoc.getElementsByTagName("My_Data2")[0].childNodes[0].nodeValue;
    My_Data3_var = xmlDoc.getElementsByTagName("My_Data3")[0].childNodes[0].nodeValue;
    My_Data4_var = xmlDoc.getElementsByTagName("My_Data4")[0].childNodes[0].nodeValue;

    document.getElementById("My_Data1").innerHTML=My_Data1_var;
    document.getElementById("My_Data2").innerHTML=My_Data2_var;
    document.getElementById("My_Data3").innerHTML=My_Data3_var;
    document.getElementById("My_Data4").innerHTML=My_Data4_var;
    setTimeout("Get_XML()",500);
}
</script>
</head>
```

(The second part of the HTML file is on the next page.)

## The HTML Body

This is very similar to the previous example with the exception that we are accessing four variables.

```
<body>
<h1>Dynamic XML Test!</h1>

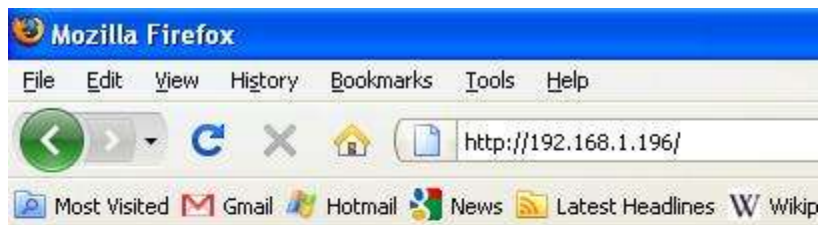
<p>
<b>My_Data1:</b> <span id="My_Data1"></span><br />
<b>My_Data2:</b> <span id="My_Data2"></span><br />
<b>My_Data3:</b> <span id="My_Data3"></span><br />
<b>My_Data4:</b> <span id="My_Data4"></span><br />
</p>

<p><button onclick="Get_XML()">Get XML</button></p>

</body>
</html>
```

## Watching it All Work

Here is the page before the button is pressed:



# Dynamic XML Test!

My\_Data1:

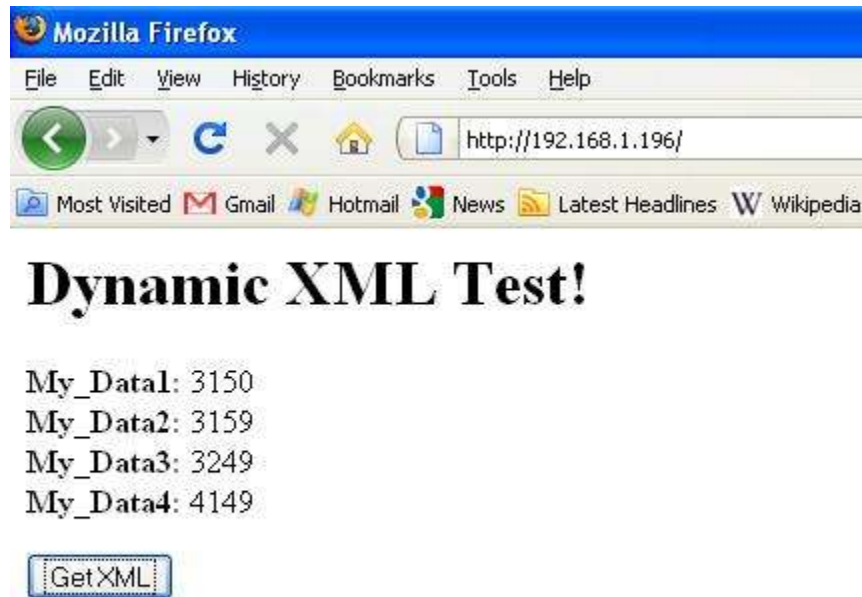
My\_Data2:

My\_Data3:

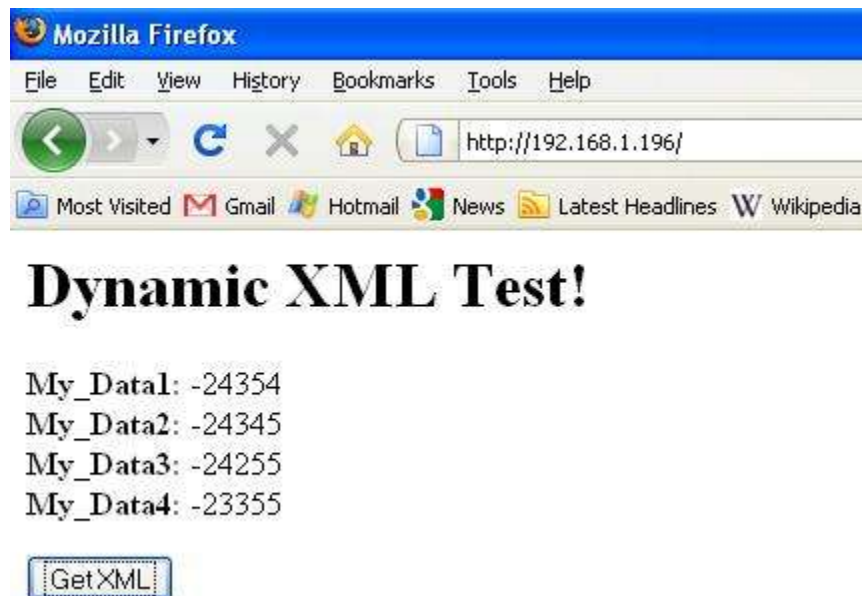
My\_Data4:

GetXML

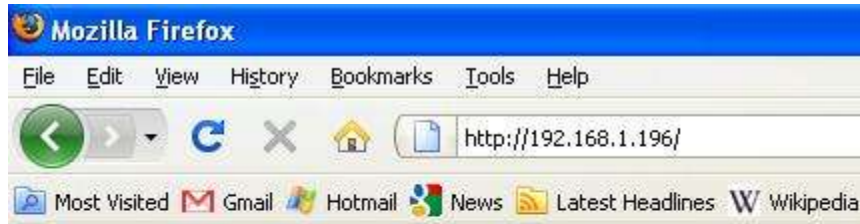
When the button is pressed, the client requests the XML file from the server and the page is automatically updated:



Every 500 milliseconds, it will continue to update automatically without any button presses.



And again:



## Dynamic XML Test!

My\_Data1: 31090

My\_Data2: 31099

My\_Data3: 31189

My\_Data4: 32089

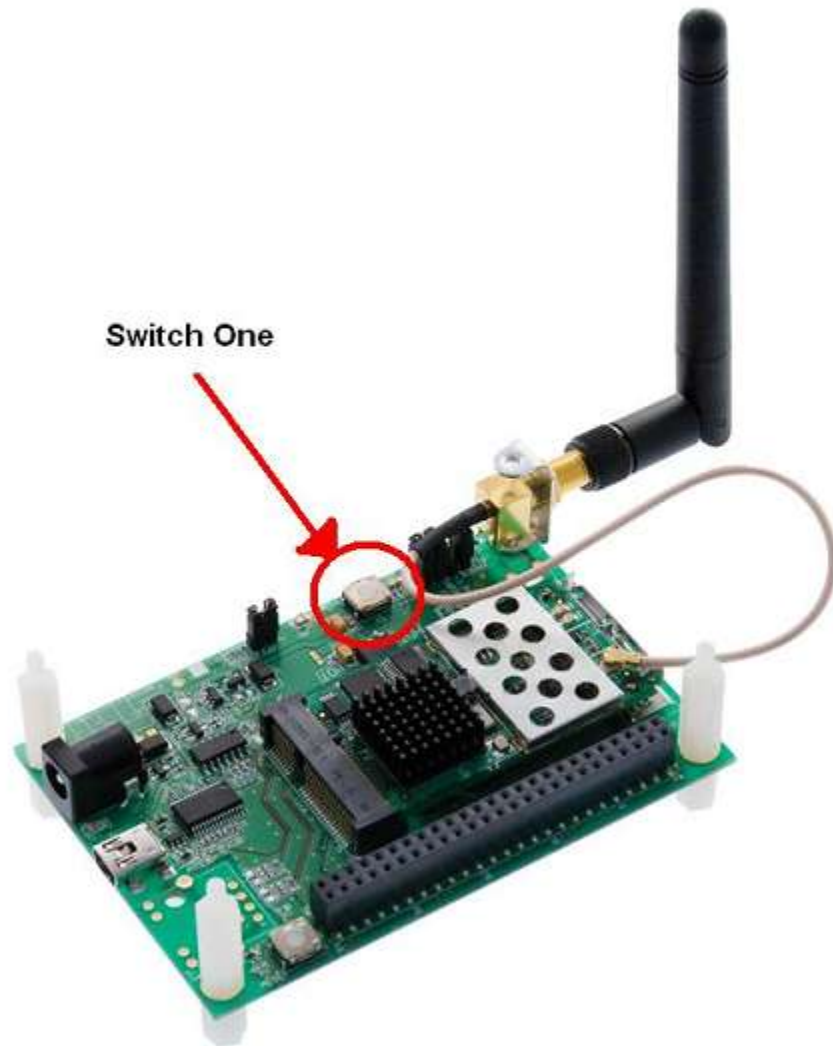


### Want more?

With the ability to swap data in real-time with your embedded controller you have the potential to create dynamic charts, graphs, and other animations with Flash or extensions to the JavaScript language like the Yahoo User Interface and Canvas.

## Chapter 18 - Dynamic Web Interface to Hardware in Real-Time

The next natural step is to move from an abstract variable like `My_Data` to a value defined by the hardware itself. The RCM5700 MiniCore and the RCM5600W MiniCore both have access to a push button switch on the prototyping board connected to pin 1 on Parallel Port D. You will find the push button labeled SW1 near the jumper array on the edge of the prototyping board.



If we assume I have created an integer named `Switch_1`, we can read the value of the switch with this statement:

```
Switch_1 = BitRdPortI(PDDR, 1);    //Read S1
```

We can then use the same method as before to create a dynamic web page to display the status of the hardware.

## A Simple XML file for Our Little Switch

Here is our dynamic XML file. Each request for the file will automatically pass it through the RabbitWeb pre-parser. (I have placed the RabbitWeb tags in **bold green** text.)

```
<!-- RabbitWeb XML file -->
<RabbitWeb>
    <Switch_1> <?z echo($Switch_1) ?> </Switch_1>>
</RabbitWeb>
```

## Dynamic C Code

Our Dynamic C code registers our Switch\_1 integer variable with the web server and passes our XML file through the ZHTML handler before we get to the main code. (Notice also that I have included the RCM56XX.LIB file.)

```
#class auto

#define USE_RABBITWEB 1

#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
#use "rcm56xxw.lib"

#ximport "C:/Pux Dyn C Files/Webinar/Dynamic_W04.html"      index_html
#ximport "C:/Pux Dyn C Files/Webinar/my_data3.xml"          my_data3_xml

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME_FUNC(".xml", "text/xml", zhtml_handler)
SSPEC_MIMETABLE_END

int Switch_1; //Create Global Variable
#web Switch_1 //Register Global Variable with RabbitWeb

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", index_html),
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/my_data3.xml", my_data3_xml)
SSPEC_RESOURCETABLE_END
```

In the main code, we poll the value of our switch and drive the web server. Notice that we include a **brdInit()** function which is specific for the RCM5600W from the [rcm56xxw.lib](#) we included earlier.

```
void main()
{
    Switch_1 = 0;

    brdInit();
    sock_init_or_exit(1);
    http_init();
    tcp_reserveport(80);

    while (1)
    {
        http_handler();
        Switch_1 = !BitRdPortI(PDDR, 1);  //Read S1
    }
}
```

## The JavaScript

Our JavaScript should merely be a variation on what we have already done. In this case, I am skipping the buffer variable in order to assign the ID attribute to the value from the XML tag in a single statement. (I have placed the single line of code that assigns the value in **bold** text.)

```
<html>
<head><script type="text/javascript">
function Get_XML ()
{
    try                //Test for Internet Explorer
    {
        xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    }
    catch(err)//If ActiveX isn't going to work we use a more codified standard
    {
        try            //Every browser but Internet Explorer
        {
            xmlDoc=document.implementation.createDocument("", "", null);
        }
        catch(err) //If that doesn't work, we produce an error mocking our user.
        {
            alert("No XML for you!");
            return;
        }
    }
    xmlDoc.async=false;        //We want a synchronous connection
    xmlDoc.load("my_data3.xml");    //Load the XML file

    //Set the ID attribute with the data from the XML file in a single line
    document.getElementById("Switch_1").innerHTML=xmlDoc.getElementsByTagName("Switch_1")[0].childNodes[0].nodeValue;
    //Infinite Loop every 500 milliseconds
    setTimeout("Get_XML ()",500);
}
</script></head>
```

## The HTML Body

The HTML body is again merely a variation on the same theme.

```
<body>

<h1>Press Switch 1 on the Protoboard!</h1>
<p><b>Switch_1:</b> <span id="Switch_1"></span><br /></p>
<p><button onclick="Get_XML ()">Start Dynamic Data</button></p>

</body>
</html>
```

## How Does it Look?

Here is the page before we turn on the infinite loop in our JavaScript.



]

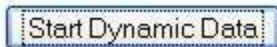


After we press the “**Start Dynamic Data**” button, we begin updating the page every 500 milliseconds by repeatedly calling the **Get\_XML()** function. The page below shows our page automatically refreshing after the “**Start Dynamic Data**” button press, but I have not pressed down on Switch 1 on the prototyping board.

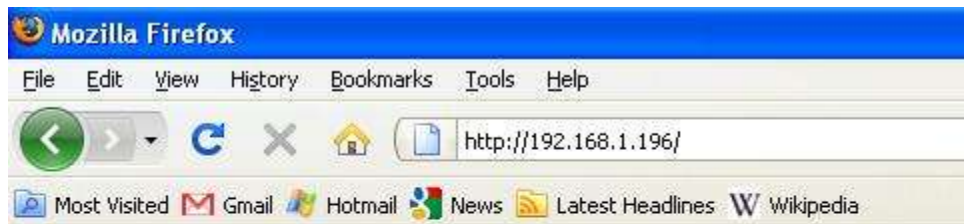


## Press Switch 1 on the Protoboard!

Switch\_1: 0

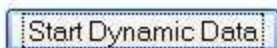


When I press down on Switch 1 on the prototyping board, the continuously refreshing stream of XML data updates the data on the web page to show the change.



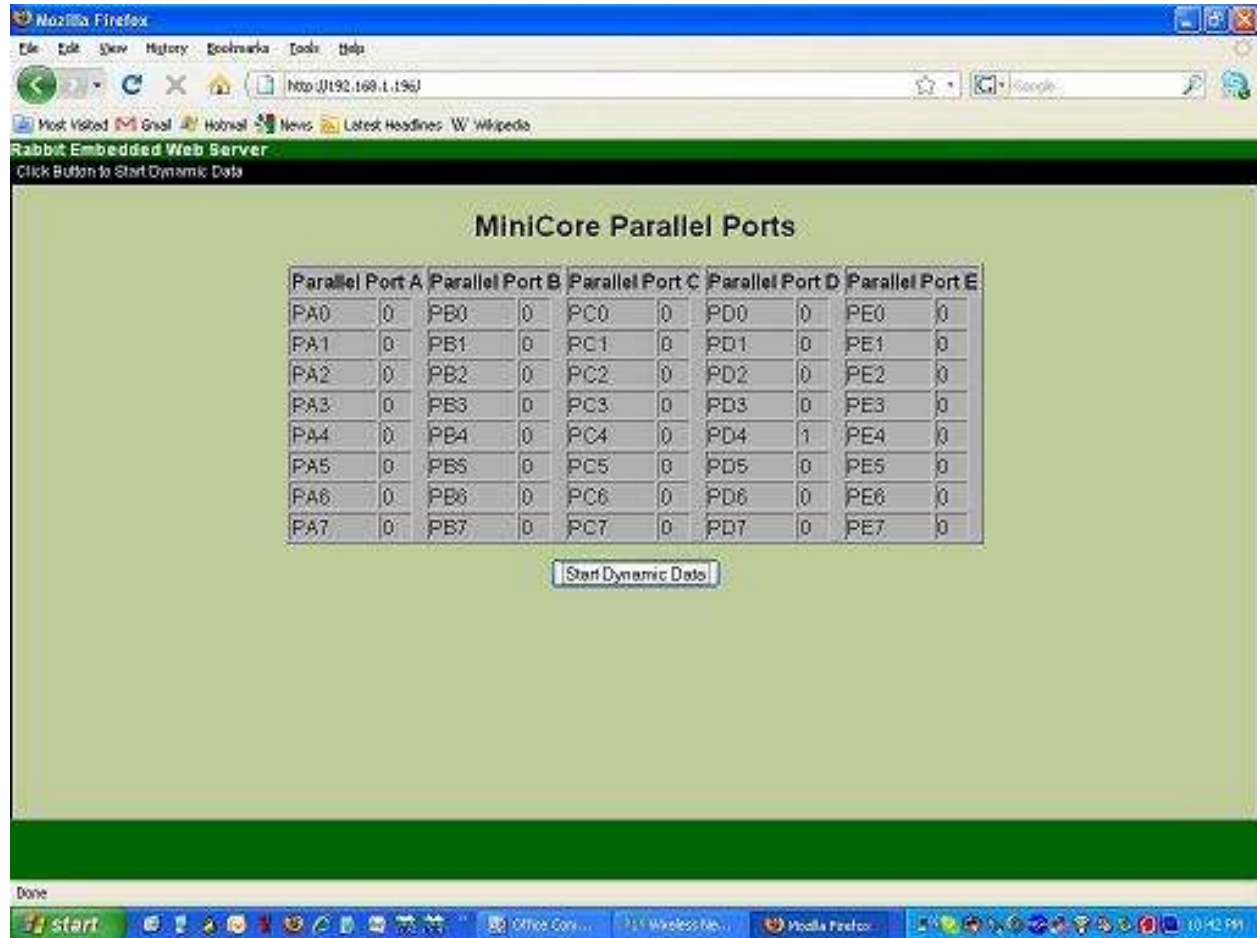
## Press Switch 1 on the Protoboard!

Switch\_1: 1



## Chapter 19 - Monitoring Every Parallel Port Pin on the MiniCore

We have already seen that we can monitor the status of a single pin. There is nothing preventing us from taking the idea further and getting the state of every parallel port pin on the Rabbit 5000 microprocessor.



You can make this table update in real-time using the same principles outlined earlier.