RabbitWeb

To Web-Enable Embedded Applications

This document explains the ease with which you can now create a web interface to your Rabbit-based device. An add-on module available starting with Dynamic C 8.50 introduces an enhanced HTTP server that, in most cases, eliminates the need for complicated CGI programming while giving the developer complete freedom in the design of their web page.

The enhanced HTTP server is called RabbitWeb and uses:

- A simple scripting language consisting of server-parsed tags added to the HTML page that contains the form.
- Dynamic C language enhancements, which includes new compiler directives that can be added to the application calling the HTTP server.

Section 1.0 presents a simple example to show the power and ease of developing a RabbitWeb server that presents a web interface to your device. This example gives step-by-step descriptions of the HTML page and the Dynamic C code. New features will be briefly explained, then linked to their comprehensive descriptions in Section 2.0 and Section 3.0. These sections are followed by a more complex example in Section 4.0, which in turn is followed by quick reference guides for both the Dynamic C language extensions and the new scripting language, which is called ZHTML (Appendix A).

1.0 Getting Started: A Simple Example

In this example, we pretend that a humidity detector is connected to your Rabbit-based controller. Your controller runs a web server that displays a page showing the current reading from the humidity detector. From this monitoring page there is a link to another page that contains an HTML form that allows you to remotely change some configuration parameters. This example introduces web variables and user groups. It also illustrates some new security features and the use of error checking.

This example assumes you have already installed Dynamic C 8.50 (or later) and hooked up the RCM3700. Instructions for doing so are in the *RabbitCore RCM3700 User's Manual*. This user's manual describes network connections for your core module, as well as setting IP addresses for running sample programs.

1.1 Dynamic C Application Code for Humidity Detector

This section describes the application for our example. The program is shown in its entirety for convenience. It is broken down into manageable pieces on the following pages.

File Name: `/Samples/tcpip/rabbitweb/humidity.c`

```
#define TCPCONFIG 1
#define USE_RABBITWEB 1
#memmap xmem
#use “dcrtcp.lib”
#use “http.lib”
#ximport “samples/tcpip/rabbitweb/pages/humidity_monitor.zhtml”
   monitor_zhtml
#ximport “samples/tcpip/rabbitweb/pages/humidity_admin.zhtml” admin_zhtml
SSPEC_MIMETABLE_START
  SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler),
  SSPEC_MIME(".html", "text/html")
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
  SSPEC_RESOURCE_XMEMFILE("/index.zhtml", monitor_zhtml),
  SSPEC_RESOURCE_XMEMFILE("/admin/index.zhtml", admin_zhtml)
SSPEC_RESOURCETABLE_END
#web_groups admin
int hum;
#web hum groups=all(ro)
int hum_alarm;
#web hum_alarm ((0 < $hum_alarm) && ($hum_alarm <= 100))\
    groups=all(ro),admin
int alarm_interval;
char alarm_email[50];
#web alarm_interval ((0 < $alarm_interval) && ($alarm_interval < 30000)) \
    groups=all(ro),admin
#web alarm_email groups=all(ro),admin
void main(void){
  int userid;
  hum = 50;
  hum_alarm = 75;
  alarm_interval = 60;
  strcpy(alarm_email, "somebody@nowhere.org");
  sock_init();                  // initialize TCP/IP stack
  http_init();                  // initialize web server

  http_set_path("/", "index.zhtml");
  tcp_reserveport(80);
  sspec_addrule (“/admin”, “Admin”, admin, admin,
                 SERVER_ANY, SERVER_AUTH_BASIC, NULL);
  userid = sauth_adduser(“harpo”, “swordfish”, SERVER_ANY);
  sauth_setusermask(userid, admin, NULL);
  while(1) {
    http_handler();
  }
}
```

The source code walk-through consists of blocks of code followed by line-by-line descriptions. Particular attention is given to the RabbitWeb #web and #web_groups statements, which are new compiler directives.

```
#define TCPCONFIG 1
#define USE_RABBITWEB 1

#memmap xmem

#use “dcrtcp.lib”
#use “http.lib”
```

The macro TCPCONFIG is used to set network configuration parameters. Defining this macro to 1 sets 10.10.6.100, 255.255.255.0 and 10.10.6.1 for the board’s IP address, netmask and gateway/nameserver respectively. If you need to change any of these values, read the comments at the top of \lib\tcpip\tcp_config.lib for instructions.

The USE_RABBITWEB macro must be defined to 1 to use the HTTP server enhancements. The #define of USE_RABBITWEB is followed by a request to map functions not flagged as root into xmem. The two #use statements allow the application the use of the main TCP/IP libraries (all brought in by dcrtcp.lib) and the HTTP server library (which also brings in the resource manager library, zserver.lib).

```
#ximport “samples/tcpip/rabbitweb/pages/humidity_monitor.zhtml”
   monitor_zhtml

#ximport “samples/tcpip/rabbitweb/pages/humidity_admin.zhtml” admin_zhtml

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

SSPEC_RESOURCETABLE_START
  SSPEC_RESOURCE_XMEMFILE("/index.zhtml", monitor_zhtml),
  SSPEC_RESOURCE_XMEMFILE("/admin/index.zhtml", admin_zhtml)
SSPEC_RESOURCETABLE_END
```

The HTML pages are copied to Rabbit memory using #ximport. The first one is a status page and the second one is a configuration interface.

Next the MIME type mapping table is set up. This allows zhtml_handler() to be called when a file with the extension .zhtml is processed. Then the static resource table is set up, which gives the server access to the files that were just copied in using #ximport. The first parameter is the name of the resource and the second parameter is its address.

```
#web_groups admin
```

The RabbitWeb server has a concept of user groups, which are created using the compiler directive, `#web_groups`. Users can be added to and removed from these groups at runtime by calling the API functions `sauth_adduser()` and `sauth_removeuser()`.

The purpose of the user groups is to protect directories and variables from unauthorized access. User groups are fully described in the section titled, Security Features.

```
int hum;
#web hum groups=all(ro)
```

This declares a variable named `hum` of type integer using normal Dynamic C syntax. It will be used to store the current humidity reported by the humidity detector. The `#web` expression registers this C variable with the web server. The read-only attribute is assigned by the "groups=all(ro)" part which gives read-only access to this variable to all user groups.

More information on registering variables is given in the section titled, Registering Variables, Arrays and Structures.

```
int hum_alarm;
#web hum_alarm \
   ((0 < $hum_alarm) && ($hum_alarm <= 100)) groups=all(ro),admin
```

This code creates a variable called `hum_alarm` to indicate the level at which the device should send an alarm. Unlike the #web statement for `hum`, there is a *guard* added when `hum_alarm` is registered. A guard is an error-checking expression used to evaluate the validity of submissions for its variable. The guard given for `hum_alarm` ensures only the range of values from 1 to 100 inclusive are accepted for this variable. More information on the syntax of the error-checking expression is in the section titled, Web Guards. The way error information is used in the HTML form is described in the section titled, Error Handling.

The dollar sign symbol in `$hum_alarm` specifies the latest submitted value of the variable, not necessarily the latest committed value of the variable. The difference between, and the importance of, the latest submitted value and the latest committed value of a variable will make more sense when you have read Section 2.2. Also, $-variables in web guards must be simple variables: for example, int, long, float, char, or string (char array). They cannot be structures or arrays.

The "admin" group is given full access to the variable (access is read and write by default), while all other users are limited to read-only access. If no "group=" parameter is given, then anyone can read or write `hum_alarm`. The order of group names is important. If "admin" came before "all(ro)" then the admin group would not have write access.

```
int alarm_interval;
char alarm_email[50];

#web alarm_interval \
   ((0 < $alarm_interval) && ($alarm_interval < 30000))\
   groups=all(ro),admin
#web alarm_email groups=all(ro),admin
```

These lines declare and register an integer variable and a string. The variable `alarm_interval` gives the minimum amount of time in minutes between two alarms, thus preventing alarm flooding. The variable `alarm_email` gives the email address to which alarms should be sent.

This concludes the compile-time initialization part of the code.

```
void main(void)
{
  int userid;
  hum = 50;
  hum_alarm = 75;
  alarm_interval = 60;
  strcpy(alarm_email, "somebody@nowhere.org");

  sock_init();                    // initialize TCP/IP stack
  http_init();                    // initialize web server


  http_set_path("/", "index.zhtml");
  tcp_reserveport(80);

  sspec_addrule (“/admin”, “Admin”, admin, admin, SERVER_ANY,
    SERVER_AUTH_BASIC, NULL);
  userid = sauth_adduser(“harpo”, “swordfish”, SERVER_ANY);
  sauth_setusermask(userid, admin, NULL);

  while(1) {
    http_handler();               // call the http server
  }
}
```

In `main()`, after the local variable `userid` is declared, there is run-time initialization of the variables that will be visible on the HTML page. Then the stack and the web server are initialized with calls to `sock_init()` and `http_init()`, respectively.

The function `http_init()` sets the root directory to “/” and sets the default file name to `index.html`. The call to `http_set_path()` can be used to change these defaults. We only want to change the default filename, so in the function call we keep the default root directory by passing “/” as the first parameter and change the default filename by passing `index.zhtml` as the second parameter. The reason we want to do this is for when the browser specifies a directory (instead of a proper resource name) we want to default to using `index.zhtml` in that directory, if it exists. If we don't use the set path function, the default is `index.html` which won't work for this sample because the file `index.html` doesn’t exist.

The call to `sspec_addrule()` configures the web server to give write and read access to the directory `/admin` to any members of the `admin` group and to require basic authentication for any access to this directory. The call to `sauth_adduser()` adds the user named `harpo` with a password of `swordfish` to the list of users kept by the server. The next function call, `sauth_setusermask()`, adds the user named `harpo` to the user group named `admin`. This sequence of calls allows you to restrict access to the file `humidity_admin.zhtml`. Only members of the user group `admin`,which in this case is the one user named `harpo`, can get the server to display a file resource that starts with `/admin`. Recall that the file `humidity_admin.zhtml` was copied to memory by the `#ximport` directive and given the label `admin_zhtml`. The file was then added to the static resource table and given the name `/admin/index.zhtml`. This is the name by which the server recognizes the file and the name by which access to it is restricted.

The web server is driven by the repeated call to `http_handler()`.

The second part of our example requires additions to the HTML page that is served by our web server. The use of the new scripting language will be explained as it is encountered in the sample pages. Regular HTML code will not be explained, as it is assumed the reader has a working knowledge of it. If that is not the case, refer to one of the numerous sources that exist (on the web, etc.) for information on HTML.

1.2 HTML Pages for Humidity Detector

This sample requires two HTML files: one to display the current humidity to all users, and another page that contains the form that allows some parameters to be changed.

1.2.1 The Monitor Page

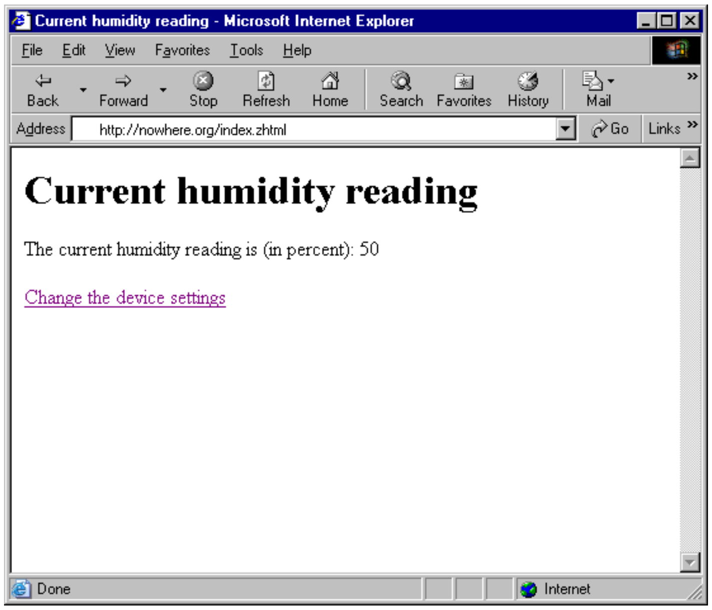
The first HTML file is `humidity_monitor.zhtml`. The "`.zhtml`" suffix indicates that it contains special server-parsed HTML tags. That is, the server must inspect the contents of the HTML file for special tags, rather than just sending the file verbatim.

File name: `humidity_monitor.zhtml`

```
<HTML>
  <HEAD><TITLE>Current humidity reading</TITLE></HEAD>
  <BODY>

    <H1>Current humidity reading</H1>
      The current humidity reading is (in percent):
        <?z print($hum) ?>
    <P>
    <A HREF="/admin/index.zhtml">Change the device settings</A>
  </BODY>
</HTML>
```

The above code displays the current humidity reading. The new server-parsed tags begin with "<?z" and end with "?>". "print ($hum)" displays the given variable (that must have been registered with #web).

This code sets up a hyperlink that the user can click on to change the device settings. Note that since it is in the "`/admin`" directory, the user will need to enter a username and password ("harpo" and "swordfish") to access the file. The username and password requirement was determined by the call to `sspec_addrule()` in `humidity.c`. Also note that the reference to the second HTML file uses the name that was given to `humidity_admin.zhtml` when it was entered into the static resource table in `humidity.c`.


Current humidity reading - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://nowhere.org/index.zhtml
Go Links
Done
Internet

Current humidity reading

The current humidity reading is (in percent): 50

Change the device settings

Figure 1. Web Page Served by RabbitWeb

This web page is very simple, as shown in Figure 1, but you are free to create more complex web pages (probably containing more variables to monitor). HTML editors such as Netscape's Composer, Hotdog Professional, and Macromedia Dreamweaver can be used to create these web pages.

1.2.2 The Configuration Page

The second HTML file is known to the server as "`/admin/index.zhtml`." Using `error()` and some conditional code allows multiple display options with the same HTML file. Again, the file is shown in its entirety for convenience. It is broken down on the following pages.

File name: `Samples/tcpip/rabbitweb/pages/humidity_admin.zhtml`

```
<HTML>
<HEAD><TITLE>Configure the humidity device</TITLE></HEAD>
<BODY>
  <H1>Configure the humidity device</H1>
  <?z if (error()) { ?>
    ERROR! Your submission contained errors. Please correct
    the entries marked in red below.
  <?z } ?>
  <FORM ACTION="/admin/index.zhtml" METHOD="POST">
    <P><?z if (error($hum_alarm)) { ?>
        <FONT COLOR="#ff0000">
      <?z } ?>
      Humidity alarm level (percent):
      <?z if (error($hum_alarm)) { ?>
        </FONT>
      <?z } ?>
      <INPUT TYPE="text" NAME="hum_alarm" SIZE=3
        VALUE="<?z print($hum_alarm) ?>">
    <P><?z if(error($alarm_email)) { ?>
        <FONT COLOR="#ff0000">
      <?z } ?>
      Send email alarm to:
      <? if (error($alarm_email)) { ?>
        </FONT>
      <?z } ?>
      <INPUT TYPE="text" NAME="alarm_email" SIZE=50
        VALUE="<?z print($alarm_email) ?>">

    <P><?z if (error($alarm_interval)) { ?>
        <FONT COLOR="#ff0000">
      <?z } ?>
      Minimum time between alarms (minutes):
      <?z if (error($alarm_interval)) { ?>
        </FONT>
      <?z } ?>
      <INPUT TYPE="text" NAME="alarm_interval" SIZE=5
        VALUE="<?z print($alarm_interval) ?>">
    <P><INPUT TYPE="submit" VALUE="Submit">
  </FORM>
  <A HREF="/index.zhtml">Return to the humidity monitor page</A>
</BODY>
</HTML>
```

After the usual opening lines of an HTML page, is our first server-parsed tag.

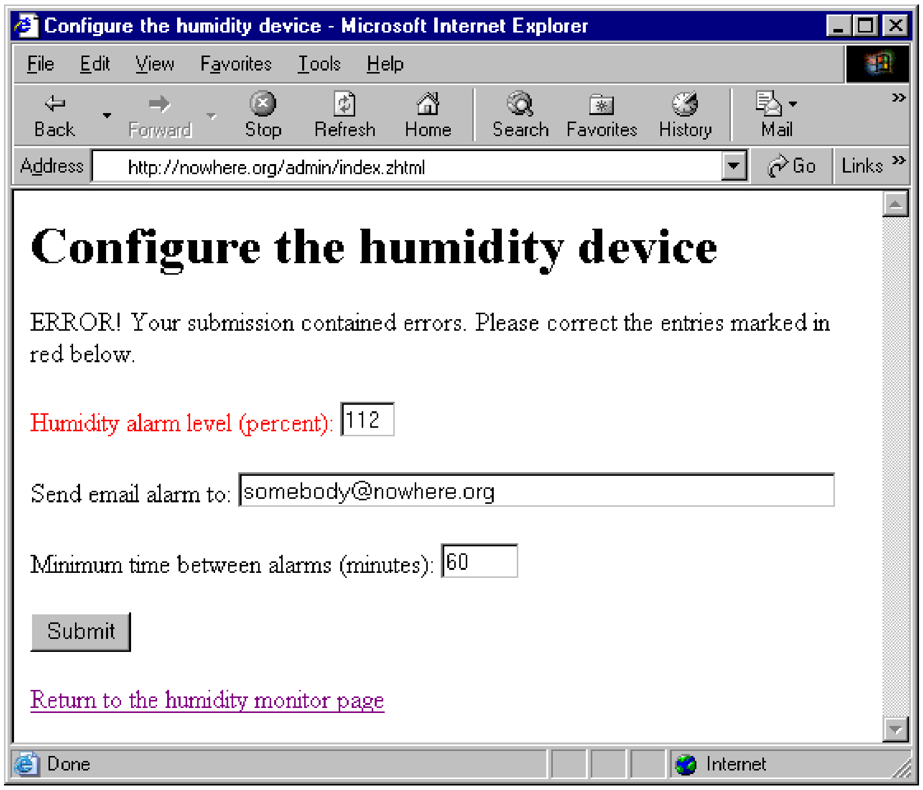
```
<?z if (error()) { ?>
  ERROR! Your submission contained errors. Please correct
  the entries marked in red below.
<?z } ?>
```

Without any parameters, `error()` returns TRUE if there were any errors in the last form submission. When the submit button for the form is clicked, the POST request goes back to the zhtml page specified by the line:

```
<FORM ACTION="/admin/index.zhtml" METHOD="POST">
```

Since this refers back to itself, if there were any errors in the last form submission, the page is redisplayed and along with it the error message inside the `if` statement.

Figure 2. Web Page with Error Message


Configure the humidity device - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://nowhere.org/admin/index.zhtml
Go Links
Done
Internet

Configure the humidity device

ERROR! Your submission contained errors. Please correct the entries marked in red below.

Humidity alarm level (percent): 112

Send email alarm to: somebody@nowhere.org

Minimum time between alarms (minutes): 60

Submit

Return to the humidity monitor page

There are five actions the user can take on this page. The Submit button was discussed above and the link to the monitor page is a common HREF link. The other three actions are the input fields of the form. These are text fields created by the INPUT tags.

```
<INPUT TYPE="text" NAME="hum_alarm" SIZE=3
   VALUE="<?z print($hum_alarm) ?>">
```

Notice how, with the use of `print()`, the value of the text fields are filled in by the server before the page is given to the browser.

Before the INPUT tag there is some code that displays text to describe the input field, along with some error checking:

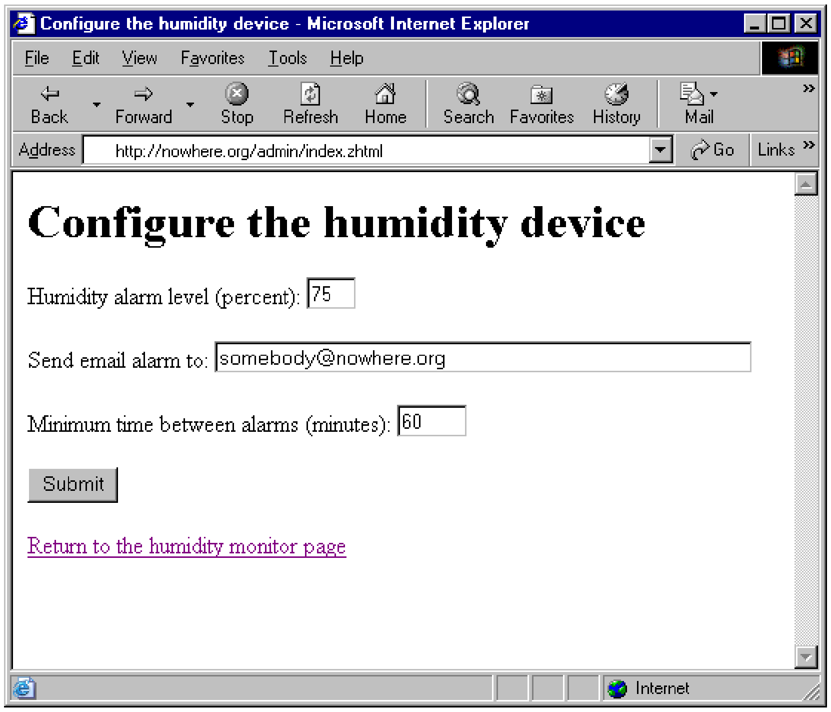
```
<?z if (error($hum_alarm)) { ?>
  <FONT COLOR="#ff0000">
  <?z } ?>
  Humidity alarm level (percent):
  <?z if (error($hum_alarm)) { ?>
     </FONT>
  <?z } ?>
```

Instead of calling `error()` with no parameters, the variable whose input field we are considering is passed to `error()`. Used with an if statement, this call to `error()` lets us change the font color to red if the value for that variable was invalid in the last form submission. Note that it is the text we have used to describe the web variable on the HTML page that is shown in red, not the value of the web variable itself. Also note that it is necessary to call `error()` twice, the second call is to close the FONT tag.

If in the last form submission the web variable had a valid value, the code above will still display the descriptive text but its font color will not be changed.

If there were no errors with any of the web variables in the last form submission, the page display reflects this status.

Figure 3. Web Page with No Error Message


Configure the humidity device - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address http://nowhere.org/admin/index.zhtml
Go Links
Internet

Configure the humidity device

Humidity alarm level (percent): 75

Send email alarm to: somebody@nowhere.org

Minimum time between alarms (minutes): 60

Submit

Return to the humidity monitor page

2.0 Dynamic C Language Enhancements for RabbitWeb

This section describes the language enhancements of the RabbitWeb module and how to make use of them to create a RabbitWeb server. These language enhancements are designed to interact with the ZHTML scripting language, (described in the section titled ZHTML Scripting Language). They work together to provide an easy-to-program web-enabled interface to your device.

2.1 Registering Variables, Arrays and Structures

Registering variables, arrays or structures with the web server is easy. First, we'll look at the simple case of an integer variable.

```
int foo;
#web foo
```

The variable foo is declared as an integer in the first expression and then registered with the web server in the second. Variable registration can only be done at compile-time.

Arrays and structures are registered in the same way as variables.

```
int temps[20];
#web temps
```

Strings, which are character arrays, can also be registered:

```
char mystring[20];
#web mystring
```

Strings receive special handling by RabbitWeb. The bounds are always checked when updating a string through a RabbitWeb interface, which means that the character buffer will not overflow.

It is permissible to register an array element without registering the entire array. For example,

```
int temps[20];
#web temps[0]
```

will register `temps[0]` but not temps[1], temps[2], etc. The same holds true for structure members.

```
struct foo2 {
   int a;
   int b;
};
struct foo2 bar;
#web bar
```

The above #web statement is functionally the same as:

```
#web bar.a
#web bar.b
```

Registering structure members or array elements separately lets you assign separate error-checking expressions to them, a topic covered in the section titled, Web Guards.

It is also possible to have arrays of structures:

```
struct foo2 bar[3];
#web bar
```

Arrays of structures can contain structures that contain arrays:

```
struct foo {
   int a;
   int b[2];
};
struct foo bar[3];
#web bar
```

And so on, and so on...

2.1.1 Selection-type Variables

Defining variables that can take on one of a list of variables is done with the `select()` feature at compile time.

```
      int color;
      #web color select("blue", "red", "green")
```

The `select()` feature is useful when creating a drop-down menu or a set of radio buttons. It is similar to an enumerated type. In this case, the actual variable, `color`, is an `int` and holds one of the values 0, 1, or 2 corresponding to the strings “blue,” “red” and “green,” respectively. To specify starting numbers other than zero, do the following:

```
      int color
      #web color select("blue" = 5, "red", "green" = 10)
```

This causes “blue” to be 5, “red” to be 6, and “green” to be 10. Unlike an `enum`, a selection-type variable can be of type `long` as well as `int`.

2.2 Web Guards

Registering variables, arrays and structures with the server is not enough—when data is received from the user, it should be checked for errors before being committed to the actual C variables. The `#web` syntax allows an optional expression to be added that is evaluated when the user submits a new value for that variable.

```
int foo;
#web foo (($foo > 0) && ($foo < 16))
```

If the C expression evaluates to TRUE (i.e., !0), the new value is accepted. If it evaluates to FALSE (i.e., 0), the new value is rejected. The new values are not applied until all variables in a submission have been checked.

To reference the old, committed (and therefore guaranteed correct) value, reference the variable directly:

```
int foo;
#web foo ((0 < $foo) && ($foo < foo))
```

One variable can reference another variable in an error-checking expression:

```
int low;
int high;

#web low
#web high ($high > $low)
#web low ($low < $high)
```

Notice that the variable `low` is registered with the web server before it is used in the error-checking expression for the variable `high`. This ordering lets the guard for `high` know that `low` is a web variable.

Arrays also need to be considered when doing error checking. The "@" character represents a wild-card for the index value. It is replaced with the index being checked in the expression:

```
int temps[20];
#web temps[@] ((50 <= $temps[@]) && ($temps[@] <= 100))
```

For example, if `temps[0]` is being checked for errors, the error-checking expression becomes:

```
((50 <= $temps[0]) && ($temps[0] <= 100))
```

Alternatively, it is possible to give each array element its own error-checking expression:

```
int temps[3];
#web temps ((50 <= $temps[0]) && ($temps[0] <= 100) && \
            (60 <= $temps[1]) && ($temps[1] <= 90) && \
            (70 <= $temps[2]) && ($temps[2] <= 80))
```

Note that the above statement spans lines. The statement is continued on the next line by escaping the end of the line.

It is also possible to register and check array variables individually:

```
int temps[3];
#web temps[0] ((50 <= $temps[0]) && ($temps[0] <= 100))
#web temps[1] ((60 <= $temps[1]) && ($temps[1] <= 90))
#web temps[2] ((70 <= $temps[2]) && ($temps[2] <= 80))
```

Structures are also supported with error checking.

```
struct foo {
   int a;
   int b;
};
struct foo bar;
#web bar ((0 < $bar.a) && ($bar.a < 10) && \
          (-5 < $bar.b) && ($bar.b < 5))
```

Alternatively, each structure element can be specified separately (using the same structure definition as above) and given its own error-checking expression.

```
struct foo bar;
#web bar.a ((0 < $bar.a) && ($bar.a < 10))
#web bar.b ((-5 < $bar.b) && ($bar.b < 5))
```

In the two code sections shown above, two similar methods for registering a structure are presented. The difference between these two methods is that the first one registers the entire structure as a single web variable, and the second one registers each element as separate web variables. In the first case, a change to any element of the structure causes the guard expression to be evaluated. In other words, changing either `bar.a` or `bar.b` will cause the guard expression to be evaluated and so both variables will be checked. In the second case, `bar.a` and `bar.b` are registered as independent web variables and so changing one does not cause the guard expression of the other one to be evaluated.

Structure elements can be specified separately in arrays of structures, as well:

```
struct foo bar[3];
#web bar[@].a (0 < $bar[@].a)
#web bar[@].b ($bar[@].a > 10)
```

The same holds true for arrays of structures, in which the structures themselves contain arrays.

```
struct foo {
   int a;
   int b[2];
};
struct foo bar[3];
#web bar[@].a (0 < $bar[@].a)
```

Of special note are variables with more than one array index. Which index is “@” referring to? Consider the following example:

```
#web bar[@].b[@] ((0 < $bar[@[0]].b[@[1]]) && ($bar[@[0]].b[@[1]] < 10))
```

In this case, the “@” in the guard is not enough. Instead, a different syntax is used, “@[#]”, where # is the #th index being referenced. If the user uses a simple “@” for a wildcarded index, it is implicitly replaced with “@[0]” (since, in general, @ is a shorthand notation for @[0]).

If the error-checking expression is not flexible enough, a user-defined function can be specified instead:

```
struct (
   int a;
   intb;
}foo;

#web foo (check_foo($foo.a, foo.b))
```

Remember that a $-variable must be a simple variable or a string (char array). It would be illegal to call the above function, `check_foo()`, with “`$foo`” since `foo` is a structure.

Consider the order of evaluation of each of the variable error checks to be undefined, that is, do not depend on the order. Also, only changed variables are checked for errors. This must be taken into account when writing guards. For example, in the following code:

```
#web low
#web high ($high > $low)
```

let us say the value of `high` is 60 and the value of `low` is 40. If these variables are presented in an HTML form and a value of 65 for `low` is submitted while the value for `high` is kept the same, it would be accepted because `low` has no guard. Since the value of `high` did not change, its guard was not activated. Hence, the guards for interdependent variables must be symmetric.

2.2.1 Reporting Errors

When a variable fails its error-check, the reason for the failure can be displayed in an HTML page by using the `WEB_ERROR()` function:

```
#web foo ((0 < $foo)?1:WEB_ERROR("too low"))
#web foo (($foo < 16)?1:WEB_ERROR("too high"))
```

Note that the checks for the variable `foo` have been split into two parts; both checks are done during the error-checking phase. If the check (such as “(0 < $foo)”) succeeds, then the expression evaluates to 1. If the check fails, then the special `WEB_ERROR()` function is triggered, which will associate the given error string with the variable and will return 0.

`RWEB_WEB_ERROR_MAXBUFFER`, which is 512 by default, defines the size of the buffer for the error strings. The buffer must be large enough to hold all error strings for a single form submission. To change it, `#define` this macro before the `#use "http.lib"` statement in the application code.

Go to the section titled Error Handling to see how the error string passed to `WEB_ERROR()` is displayed in an HTML page.

2.3 Security Features

Various HTTP authentication methods (basic, digest, and SSL) are supported to protect web pages and variables. Each method requires a username and password for access to the resource. Permissions are granted based on user groups rather than on individual user ids. User groups are defined at compile-time; however, users can be added to or removed from a user group at run-time.

The groups are defined at compile-time in this manner:

```
#web_groups admin,users
```

This statement creates the two groups, “admin” and “users.” The symbols “admin” and “users” are added to the global C namespace. These represent unsigned 16-bit numbers. Each group has one of the 16 bits set to 1, so that the groups can be ORed together when multiple groups need to be referenced. Note that this limits the number of groups to 16.

The web server does not directly know that “admin” is for administrative users and “users” is for everyone else. This distinction is made by how the programmer assigns protection to server resources. For example,

```
#web foo ($foo > 0) groups=users(ro),admin
```

limits access to the variable `foo`. This variable is read-only for those in the “users” group. “(rw)” can be specified to mean read-write for the “admin” group, but this is the default so it is not necessary. The group “all” is a valid group, which will give access to a variable to all users regardless of group affiliation. By default, all groups have access to all variables. The “groups=” is used to limit access. Consider the line:

```
#web foo ($foo > 0) groups=users(ro)
```

This line causes the `admin` group to have no access to the variable `foo`. In other words, if there is a “groups=” clause then any group that is not mentioned explicitly in it will have no access to the variable to which it applies.

Also the order of the groups is important if the “all” group is mentioned. For example, the line:

```
#web foo ($foo > 0) groups=all(ro),admin
```

gives read/write access to the admin group. But the line

```
#web foo ($foo > 0) groups=admin,all(ro)
```

limits the admin group to read-only access.

To add a user to a group, you must first add the user to the list kept by the server by calling `sauth_adduser()`. The value returned by `sauth_adduser()` identifies a unique user. This value is passed to `sauth_setusermask()` to set the groups that the user will be in. For example:

```
id = sauth_adduser(“me”, “please”, HTTP_SERVER);
sauth_setusermask(id, admin|users, NULL);
```

The user `me` is now in both the “admin” group and the “users” group. The groups determine what server resources the user can access. The user information only determines what username and password must be provided for the user to gain access to that group’s resources.

The web server has no concept of which variables are located on which forms. By allowing certain variables to be available to certain user groups, it doesn’t matter which variables are located on which forms—

any user can update variables through any POST-capable resource as long as a group the user is a member of has access to that variable.

It may also be important to update certain variables only through certain authentication methods. For instance, if the data must be secret, you can require that it only be updated via SSL. You can also make certain variables be read-only for certain user groups.

Valid user groups and authentication methods can be specified as follows:

```
#web foo(foo > 0) auth=basic,digest,ssl groups=admin,users(ro)
```

By default, all authentication methods and user groups are allowed to update the variable. That is, to limit access to the variable, you must include the applicable auth= or groups= parameters when registering the variable.

"none" is a valid authentication method.

If `foo` is a structure or array, the protection modes are inherited by all members of the structure or array unless specifically overridden with another `#web` statement.

If a received variable fails a security check, then the client browser will be given a "Forbidden access" page.

2.4 Handling Variable Changes

Receiving, checking, and applying the variable changes works well when the program does not immediately need to know the new values. For instance, if we are updating a value that represents the amount of idle time needed on a serial port before sending the queued data over the Ethernet, the program does not need to know the new interval value immediately—it can just use the new value the next time it needs to do the calculation. But sometimes the program must perform some action when values have been updated. For example, if a baud rate is changed on a serial port, then that serial port likely needs to be closed and reopened. To handle this, and similar situations, a callback function can be associated with an arbitrary group of variables:

```
    #web_update foo, bar, baz user_callback
```

If any variable within a group is changed, then the callback function for that group is called. The user program, through the callback function, can then take the proper action based on the new value. The above statement means that if any of the variables `foo`, `bar`, or `baz` are updated, then the function `user_callback()` will be called to notify the application. If variables in more than one group are updated at once, each group's callback function will be called in turn (with no guarantees on order of calls). If a variable is in multiple groups and that is the only variable updated, then all update callback functions are called, although the order in which they are called is unspecified.

There is an important restriction on the use of `#web_update` for arrays and structures: for an array element or structure member registered explicitly (that is, with its own `#web` statement), the callback function associated with the array or structure as a whole will not be called when the variable is updated. For example, consider:

```
struct foo2 {
   int a;
   int b;
};
struct foo2 bar;

#web bar
#web bar.a       //#web_update variables must be explicitly #web registered

#web_update bar user_callback()
#web_update bar.a differentuser_callback()
```

If `bar.b` is updated, `user_callback()` is called, but if `bar.a` is updated, the function `differentuser_callback()` is called and not `user_callback()`.

2.4.1 Interleaving Problems

Consider the following scenario: Users A and B are operating a web interface on Device C. User A gets a form page from Device C and then leaves the computer for a while. User B then gets the same form page from Device C, updates the data, and then the new values are committed on Device C. Then, User A comes back to his computer, makes changes to the form that was left on his screen from earlier, and submits those values. Keep in mind that User A never saw the update done by User B. What should Device C do? Should it allow A's update? Or should it tell User A that an interim update has been made, and that he should thus review his changes in light of that fact?

Ideally, the developer should be in control of how this scenario is handled since different applications have different needs. One way to avoid trashing a valid update is given here:

```
int foo;
#web foo ($foo == foo + 1)
#web_update foo increment_value

void increment_value(void) {
   foo++;
}
```

Some client-side JavaScript is needed in the ZHTML file where the `foo` value is included:

```
<SCRIPT>
   document.write('<INPUT TYPE="hidden" NAME="foo"
     VALUE="' + (<?z echo($foo) ?> + 1) + '">')
</SCRIPT>
```

This causes the variable `foo` to be updated whenever a successful update is made. Here's how it works:

- The developer gives `foo` an initial value
- Included in a form is a hidden field that represents the value of `foo` (see the HTML in the SCRIPT tags above)
- In the JavaScript above, the "`<?z echo($foo) ?>`" is first replaced by the HTTP server with the current value of `foo`.
- In the browser, the JavaScript is executed, which takes the value of `foo` and adds one to it. This is the value of the hidden input field.
- When the form data is submitted, the automatic error-checking will recognize that the value of `foo` has been updated along with the other data. If all data passes the error-checking, then the value of `foo` is incremented by the user's `increment_value()` function.
- If, when the form data is submitted, the current value of `foo` plus one does not match the submitted value of foo, then we know that an interim update has occurred. The value of `foo` is marked as an error by the server, which can be handled by the developer's ZHTML page. Note that none of the updated form values will be committed, since an error was triggered.

3.0 ZHTML Scripting Language

This section describes the ZHTML scripting language: a set of features that can be included in HTML pages. These features interact with some new features in Dynamic C (described in the section titled, Dynamic C Language Enhancements for RabbitWeb) to create an enhanced HTTP server.

3.1 SSI Tags, Statements and Variables

The new server-parsed tag is similar to SSI tags prior to Dynamic C 8.50, in that they are included in HTML pages and are processed by the server before sending the page to the browser.

The new tags all have the following syntax:

```
<?z statement ?>
```

That is, a valid `statement` is preceded by `<?z` and followed by `?>`. This follows the naming scheme for PHP and XML tags ("<?php" and "<?xml", respectively) and so follows standard conventions.

To surround a block of HTML, do the following:

```
<?z statement { ?>
<H1>HTML code here!</H1>
<?z } ?>
```

The <?z ... ?> tags delimit statements. This means that you cannot put two statements in a single set of tags. For example,

```
<?z if (error($foo)) {
   echo($foo)
} ?>
```

is not valid. The `if`, `echo`, and "}" statements must be separated by the <?z ... ?> tags like the following:

```
<?z if (error($foo)) { ?>
   <?z echo($foo) ?>
<?z } ?>
```

Note that "}" is considered a statement in ZHTML (the "close block" statement), and must always be in its own <?z ... ?> tags.

The simplest use of the new SSI tag simply prints the given variable:

```
<?z print($foo) ?>
```

The value of the given variable is displayed using a reasonable default `printf()` specifier. For example, if `foo` is an `int`, `print()` uses the conversion specifier %d. `foo` must be registered with the web server. How to register a variable with the web server is described under Registering Variables, Arrays and Structures.

A variable must begin with a "$" character to access its last submitted value. This value may or may not have been committed. The last committed value is accessible using "@," as in:

```
<?z print(@foo) ?>
```

Why is there a distinction between the last submitted value and the last committed one? In other words, does it matter in the HTML code whether the submission value is valid? It can. See the section titled Error Handling for more information.

To specify a printf-style conversion specifier, the `printf()` function can be used, but with the limitation that it will only accept one variable as an argument:

```
<?z printf("%ld", $long_foo) ?>
```

Note that the `print` function does not generate the code for the form widget itself—this is done with the INPUT tag, an HTML tag that generates a specific form element. Here is an example of using the `print` command to display a value in a form widget:

```
<INPUT TYPE="text" NAME="foo" SIZE=10
   VALUE= "<?z print($foo) ?>">
```

For the value to be updateable, the NAME field must be the name of the variable. Otherwise, when the form is submitted, the web server will not know where to apply the new value. This is not true of arrays. When referencing arrays the name must differ somewhat from the C name because the '[' and ']' symbols are not valid in the NAME parameter of the INPUT tag due to limitations in HTTP.

The `varname()` function must be used to make the variable name safe for transmission.

```
NAME="<?z varname($foo[3]) ?>"
```

That is, `varname()` automatically encodes the variable name correctly.

3.2 Flow Control

In addition to simply displaying variables in your HTML documents, the new ZHTML tag allows some simple looping and decision making.

3.2.1 Looping

A `for` loop, when combined with arrays, makes it easy to display lists of variables. The format of the `for` loop is as follows:

```
<?z for ($A = start; $A < end; $A += step) { ?>
   <H1>HTML code here!</H1>
<?z } ?>
```

where:

- **A**: A single-letter variable name from A-Z. These loop-control variables take on an `unsigned int` value.
- **start**: The initial value of the for loop. The value of the variable will start at this value and count to the `end` value.
- **end**: The upper value of the `for` loop. The operator may be any one of the following: `<`, `>`, `==`, `<=`, `>=`, `!=`.
- **step**: The number by which the variable will change for each iteration through the loop. The operator may be any one of the following: `++`, `--`, `+=step`, `-=step`. Note that `$A++` will increment the variable by 1.

Note that although this `for` loop looks like the regular Dynamic C `for` loop, its use is restricted to what is documented here.

To display a list of numbers in HTML using a `for` loop, you can do something like this:

```
<TABLE><TR>
<?z for ($A = 0; $A < 5; $A++) { ?>
   <TD><?z print($foo[$A]) ?>
<?z } ?>
</TR></TABLE>
```

This code will display the variables `foo[0]`, `foo[1]`, `foo[2]`, `foo[3]`, and `foo[4]` in an HTML table.

It is also possible to get the number of elements in a one-dimensional array by doing the following:

```
<?z for ($A = 0; $A < count($foo, 0); $A++) { ?>
```

The second parameter to `count()` indicates that we want the upper bound of the nth array index of `foo`. (From this you can infer that the first parameter must be an array!) For example, if `$foo` is a three-dimensional array, then count `($foo, 0)` yields the array bound for the first dimension, count `($foo, 1)` yields the array bound for the second dimension and count `($foo, 2)` yields the array bound for the third dimension.

3.2.2 Conditional Code

In addition to looping, you can have conditional code with `if` statements. The `if` statement is specified as follows:

```
<?z if ($A == 0) { ?>
   HTML code
<?z } ?>
```

where:

- **A**: The variable to check in the conditional. This can be anything that evaluates to a number, whether it be a normal integral #web-registered variable, a loop variable, a numeric literal, or a `count()` expression.
- **==**: The relational operator in the `if` statement. This can be "==", "!=", "<", ">", "<=", or ">=".
- **0**: The number to which the variable should be compared. This can be anything that evaluates to a number, whether it be a normal integral #web-registered variable, a loop variable, a numeric literal, or a `count()` expression.

For example:

```
<?z if ($foo == 0) { ?>
   HTML code
<?z } ?>
```

or

```
<?z if ($foo == @foo) { ?>
   HTML code
<?z } ?>
```

are both legal.

The table from the previous example was modified to allow one of the values to be displayed in an input widget, whereas all the other values are simply displayed.

```
<TABLE><TR>
   <?z for ($A = 0; $A < count($foo, 0); $A++) { ?>
      <TD>
      <?z if ($A == 3) { ?>
         <INPUT TYPE="text" NAME="<?z varname($foo[$A]) ?>"
            VALUE="<?z print($foo[$A]) ?>">
      <?z } ?>
      <?z if ($A != 3) { ?>
         <?z print($foo[$A]) ?>
      <?z } ?>
   <?z } ?>
</TR></TABLE>
```

`if` statements can be nested. Even `for` loops can be nested within other `for` loops. The nesting level has an upper limit defined by the macro `ZHTML_MAX_BLOCKS`, which has a default value of 4.

3.3 Selection Variables

Put together, `if` statements and `for` loops are useful for selection-type variables. To iterate through all possible values of a selection-type variable and output the appropriate "<OPTION>" or "<OPTION SELECTED>" tags, something like the following can be done:

```
<?z for ($A = 0; $A < count($select_var); $A++) { ?>
   <OPTION
      <?z if (selected($select_var, $A)) { ?>
         SELECTED
      <?z } ?>
   >
<?z print_opt($select_var, $A) ?>
```

This syntax allows for maximum flexibility. In this case, the `count()` function returns the number of options in a selection variable. The `selected()` function takes a selection variable and an index as parameters. It returns TRUE if that option matches the current value, and FALSE if it doesn't.

The `print_opt()` function outputs the $A-th possible value.

The following is a convenience function that automatically generates the option list for a given selection variable:

```
<?z print_select($select_var) ?>
```

3.4 Checkboxes and RadioButtons

This section describes how to add checkboxes and radiobuttons to your web page.

Checkboxes are a bit tricky because if a checkbox is not selected, then no information on that variable is sent to the server. Only if it is selected will the variable value ("on" by default, or whatever you have in the VALUE="this_is_the_value" attribute) be passed in. In particular this means that if a variable was checked, but then you uncheck it, the server will not be able to tell the difference between that variable being unchecked and that variable value simply not being sent. The server would need a notion of the full list of variables that should be in a specific form, information which RabbitWeb does not have.

However, there is a workaround. If a variable is included in a form multiple times, its value will be submitted multiple times. RabbitWeb will take the last value given as the true value, and ignore all previous ones. So, to force a default unchecked value, you can include a hidden variable before you do the checkbox INPUT field. Since you can do ZHTML comparisons with numbers, if you give the variable the value 0 or 1, it can be used in the checkbox INPUT tag.

```
<INPUT TYPE="hidden" NAME="<?z varname($checkbox[0]) ?>"
   VALUE="0">

<INPUT TYPE="checkbox" NAME="<?z varname($checkbox[0]) ?>
   VALUE="1"
   <?z if ($checkbox[0] == 1) { ?>
      CHECKED
   <?z } ?>
>
```

So, if the value of `$checkbox[0]` is 1, then the `CHECKED` attribute will be included and the checkbox will be checked. Otherwise, it will be blank. If it is checked when the form is displayed, but you clear the value, this still works, since the hidden field with a value of 0 will always be sent.

Since a list of radiobuttons is more likely to be subject to different formatting depending on user taste than something like a pulldown menu, there is no automatic way of generating the list. The best way to generate a list of radiobuttons is to use a for loop and the count function.

The following page displays both a checkbox and a list of radiobuttons.

```
<HTML>
<HEAD>
<TITLE>Radio button and checkbox</TITLE>
</HEAD>
<BODY>
<form action="./index.zhtml" method="post" >
  <INPUT TYPE="hidden" name="checkboxBoolean"  VALUE="0" >
  <INPUT TYPE="checkbox"
    <?z if($checkboxBoolean==1) { ?>
      CHECKED
    <?z } ?>
    NAME="checkboxBoolean" VALUE="1" >
  <br><br>

  <?z for ($A = 0; $A < count($radiobutton); $A++){ ?>
    <INPUT TYPE="radio" NAME="radiobutton"
      VALUE="<?z print_opt($radiobutton, $A) ?>"
      OPTION
      <?z if (selected($radiobutton, $A) ) { ?>
        CHECKED
      <?z } ?> >
  <?z } ?>
  <br><br>

  <INPUT TYPE="Submit"   VALUE="Submit" >
</form>
</BODY>
</HTML>
```

To take advantage of the above zhtml script, the server code would need something like the following:

```
int checkboxBoolean, radiobutton;
#web checkboxBoolean
#web radiobutton select("0" = 0, "1", "2", "3", "4", "5", "6", "7")

checkboxBoolean = 0;
radiobutton = 0;
```

3.5 Error Handling

One of the biggest benefits to the new server-parsed HTML tags is the ability to perform actions based on whether a user-submitted variable was in error. A natural way of creating an HTML user interface is to create the form on an HTML page. When the user enters (or changes) values and submits the result, the server should check the input for errors. If there are errors, then the same form can be redisplayed. This form can mark the values that are in error and allow the user to update them. With the use of conditionals, it is possible to create both the original form and the form that shows the errors in the same page.

The destination page of a submitted form can be any page. When the web server receives a POST request with new variable data, it checks the data using the error-checking expression in the `#web` statement that registered the variable. If there is an error, then the destination web page is displayed in error mode. The following text describes how error mode affects the display of the destination web page.

By default, the `print` statement displays the new value of the variable when in error mode. To override the default behavior and show the old, committed value (note that the erroneous value has not been committed), do the following:

```
<?z print(@foo) ?>
```

The "@" symbol specifies the old value of the variable.

To execute some code only when a certain variable has an error, do the following:

```
<?z if (error($foo)) { ?>
   This value is in error!
<?z } ?>
```

It is also possible to say: `!error($foo).`

If a value submitted for a variable has an error, then `error(var)` used in a `print` statement evaluates to an error string if one was defined using the method described in the section titled, Reporting Errors. Here is an example:

```
<?z if (error($foo)) { ?>
   This value is <?z print(error($foo)) ?>!
<?z } ?>
```

Although the ZHTML parser can output error messages into the HTTP stream, these messages may not be visible on a web page depending on how the browser is displaying pages. The surest way to find out exactly the result of a ZHTML page is to check the source of the page in the browser. For Internet Explorer, the user can choose the "View/Source" menu item. Other browsers have equivalent functionality.

To display some information if the page is being displayed in error mode, use `error()` with no parameter. If any variable in the form has an error, `error()` will return TRUE. Here is an example of its use:

```
<?z if (error()) { ?>
   Errors are in the submission! Please correct them below.
<?z } ?>
```

3.6 Security: Permissions and Authentication

To check if a user has authorization for a specific variable, call the `auth()` function:

```
<?z if (auth($foo, “rw”)) { ?>
   You have read-write access to the variable foo.
<?z } ?>
```

“ro” is also a valid second parameter.

To check if the current page is being displayed as the result of a POST request instead of a GET request, call the `updating()` function.

```
<?z if (updating()) { ?>
   <?z if (!error()) { ?>
      <META HTTP-EQUIV=”Refresh” CONTENT=”0;
          URL=http://yoururl.com/”>
   <?z } ?>
<?z } ?>
```

Both `auth()` and `updating()` may be preceded by “1” (the not operator).

4.0 TCP to Serial Port Configuration Example

This section is a step-by-step description of the sample program `ethernet_to_serial.c`. It is located in `Samples\tcpip\rabbitweb`.

This sample program can be used to configure a simple Ethernet-to-serial converter. For simplicity, it only supports listening on TCP sockets, meaning that Ethernet-to-serial devices can only be started by another device initiating the network connection to the Rabbit.

Each serial port is associated with a specific TCP port. The Rabbit listens on each of these TCP ports for a connection. It then passes whatever data comes in to the associated serial port, and vice versa.

4.1 Dynamic C Application Code

The program starts with a configuration section:

```
#define TCPCONFIG 1
```

This `#define` statement sets the predefined TCP/IP configuration for this sample. If the default network configuration of 10.10.6.100, 255.255.255.0 and 10.10.6.1 for the board’s IP address, netmask and gateway/nameserver respectively are not acceptable, change them before continuing. See `LIB\TCPIP\TCP_CONFIG.LIB` for instructions on how to change the configuration.

```
const char ports_config[] = { 'E', 'F' };

#define E2S_BUFFER_SIZE 1024
#define HTTP_MAXSERVERS 1

#define MAX_TCP_SOCKET_BUFFERS (HTTP_MAXSERVERS +
  sizeof(ports_config))

#define SERINBUFSIZE 127
#define SEROUTBUFSIZE 127
```

Each element in array `ports_config` corresponds to a serial port. In the following code, the size of this array will be used in `for` loops to identify, initialize and monitor the serial ports. A buffer is defined that will hold the data that is being passed from the Ethernet port to the serial port. The number of server instances is set to one and the number of socket buffers is set to the number of server instances plus the number of serial ports. The last two defines will be used later to allocate space for the receive and transmit buffers used by the serial port drivers.

This is the end of the configuration section.

```
#memmap xmem

#define USE_RABBITWEB 1

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

#ximport "samples/tcpip/rabbitweb/pages/config.zhtml"
    config_zhtml

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

SSPEC_RESOURCETABLE_START
  SSPEC_RESOURCE_XMEMFILE("/", config_zhtml),
  SSPEC_RESOURCE_XMEMFILE("/index.zhtml", config_zhtml)
SSPEC_RESOURCETABLE_END
```

This block of code asks the compiler to map functions not declared as root to extended memory. Setting the macro `USE_RABBITWEB` to one enables the use of the scripting language and the HTTP enhancements. (Other macros that affect these features are described in the reference section.) Next the TCP/IP libraries are brought in, as well as the HTTP library. The HTML page that contains the configuration interface to the serial ports is copied into memory with the `#ximport` directive.

HTTP servers require MIME type mapping information. This information is kept in the MIME table, which is set up by the `SSPEC_MIME_*` macros.

The `SSPEC_RESOURCE*` macros set up the static resource table for this server. The resource table is a list of all resources that the server can access. In this case, the server has knowledge of two resources named “/” and “/index.zhtml”. When either of these is requested, the config.zhtml file is served. The file extension (zhtml) identifies the file as containing server-parsed tags.

```
void restart_socket(int i);
void update_tcp(void);
void restart_serial(int i);
void update_serial(void);
void serial_open(int i);
void e2s_init(void);
void e2s_tick(void);
```

These are the function declarations. They will be defined later in the program.

```
struct SerialPort {
  word tcp_port;
  struct {
     char port;
     long baud;
     int databits;
     int parity;
     int stopbits;
  } ser;
};

struct SerialPort serial_ports[sizeof(ports_config)];
struct SerialPort serial_ports_copy[sizeof(ports_config)];
```

The SerialPort structure has fields for the configuration information for each serial port and TCP port pair. The serial_ports array (and its copy) stores configuration information about the serial ports. serial_ports_copy[] is used to determine which port information changed when the update function is called.

```
#web serial_ports[@].tcp_port ($serial_ports[@].tcp_port > 0)
#web serial_ports[@].ser.port
```

The first #web statement is registration for the TCP port. Note that the only rule in the guard is that the new value must be greater than zero. The next #web statement registers the character representing the serial port, in this case, “E” or “F.”

```
#web serial_ports[@].ser.baud(($serial_ports[@].ser.baud >= 300)? \
                              1:WEB_ERROR("too low"))

#web serial_ports[@].ser.baud(($serial_ports[@].ser.baud <= 115200)? \
                              1:WEB_ERROR("too high"))
```

These two #web statements correspond to the baud rate. The guards are split into two so that the WEB_ERROR() feature can be used. The string passed to WEB_ERROR() can later be used in the ZHTML scripting to indicate why the guard statement failed.

```
#web serial_ports[@].ser.databits select("7" = 7, "8" = 8)
#web serial_ports[@].ser.parity select("None" = 0, "Even", "Odd")
#web serial_ports[@].ser.stopbits select("1" = 1, "2" = 2)
```

These are selection variables. They limit the available options for serial port configuration parameters.

```
#web_update serial_ports[@].tcp_port update_tcp
#web_update serial_ports[@].ser.baud,serial_ports[@].ser.databits,\
        serial_ports[@].ser.stopbits update_serial
```

The `#web_update` feature will initiate a function call when the corresponding variables are updated. Note that `update_tcp()` will be called when the TCP port changes, and `update_serial()` will be called when any of the other serial port configuration parameters are updated.

```
#define AINBUFSIZE SERINBUFSIZE
#define AOUTBUFSIZE SEROUTBUFSIZE
#define BINBUFSIZE SERINBUFSIZE
#define BOUTBUFSIZE SEROUTBUFSIZE
#define CINBUFSIZE SERINBUFSIZE
#define COUTBUFSIZE SEROUTBUFSIZE
#define DINBUFSIZE SERINBUFSIZE
#define DOUTBUFSIZE SEROUTBUFSIZE
#define EINBUFSIZE SERINBUFSIZE
#define EOUTBUFSIZE SEROUTBUFSIZE
#define FINBUFSIZE SERINBUFSIZE
#define FOUTBUFSIZE SEROUTBUFSIZE
```

These set the receive and transmit buffer sizes for the serial ports. In this example only serial ports “E” and “F” are being used, but here, as well as in the function `e2s_init()`, code is included for all possible serial ports. In this way it is relatively easy to change the serial ports being used simply by changing the character array, `ports_config[]`.

```
enum {
  E2S_INIT,
  E2S_LISTEN,
  E2S_PROCESS
};
```

These are symbols representing different states in the Ethernet-to-serial state machine.

```
struct {
  int state;                      // Current state of the state machine
  tcp_Socket sock;                // Socket associated with this serial port

  // The following members are function pointers for accessing this serial port
  int (*open)();
  int (*close)();
  int (*read)();
  int (*write)();
  int (*setdatabits)();
  int (*setparity)();
} e2s_state[sizeof(ports_config)];
```

The `e2s_state` array of structures holds critical information for each socket/serial port pair, namely the socket structures that are used when calling TCP/IP functions and the various serial port functions that access the serial ports or set serial port parameters.

The first member of the structure (`state`) is the value of the variable that determines which state of the Ethernet-to-serial state machine will execute the next time `e2s_tick()` is called.

```
char e2s_buffer[E2S_BUFFER_SIZE];
```

This is a temporary buffer for copying data between the serial port buffers and the socket buffers.

Now we will look at the functions that were declared earlier in the program.

```
void restart_socket(int i)
{
  printf("Restarting socket %d\n", i);
  // Abort the socket
  sock_abort(&(e2s_state[i].sock));
  // Set up the state machine to reopen the socket
  e2s_state[i].state = E2S_INIT;
}
```

The function `restart_socket()` displays a screen message and then aborts the socket. The state variable for the Ethernet-to-serial state machine is set to the initialization state, which will cause the socket to be opened for listening the next time the state machine tick function is called.

```
void update_tcp(void){
  auto int i;

  // Check which TCP port(s) changed
  for (i = 0; i < sizeof(ports_config); i++) {
    if (serial_ports[i].tcp_port != serial_ports_copy[i].tcp_port)
    {
      // This port has changed, restart the socket on the new port
      restart_socket(i);

      // Save the new port, so we can check which one changed on the next update
      serial_ports_copy[i].tcp_port = serial_ports[i].tcp_port;
    }
  }
}
```

The function `update_tcp()` is called when a TCP port is updated via the HTML interface. It determines which TCP port(s) changed, and then restarts them with the new parameters.

```
void restart_serial(int i){
  printf("Restarting serial port %d\n", i);
  e2s_state[i].close();         // Close the serial port
  serial_open(i);               // Open the serial port
}
```

The function `restart_serial()` closes and then reopens the serial port specified by its parameter.

```
void update_serial(void){
  auto int i;

  // Check which serial port(s) changed
  for (i = 0; i < sizeof(ports_config); i++)
  {
    if (memcmp(&(serial_ports[i].ser),
           &(serial_ports_copy[i].ser),
           sizeof(serial_ports[i].ser)))
    {
      // This serial port has changed, so re-open the serial port with the new parms
      restart_serial(i);

      // Save the new parameters, so we can check which one changed on the next update
      memcpy(&(serial_ports_copy[i].ser),
           &(serial_ports[i].ser),
           sizeof(serial_ports[i].ser));
    }
  }
}
```

The function `update_serial()` is called when a serial port is updated via the HTML interface. It determines which serial port(s) changed, and then restarts them with the new parameters.

```
void serial_open(int i)
{
  //  Open the serial port
  e2s_state[i].open(serial_ports[i].ser.baud);

  // Set the data bits
  if (serial_ports[i].ser.databits == 7) {
    e2s_state[i].setdatabits(PARAM_7BIT);
  }
  else {
    e2s_state[i].setdatabits(PARAM_8BIT);
  }
  //  Set the stop bits
  if (serial_ports[i].ser.stopbits == 1) {
    if (serial_ports[i].ser.parity == 0) {          // No parity
      e2s_state[i].setparity(PARAM_NOPARITY);
    }
    else if (serial_ports[i].ser.parity == 1) {     // Even parity
      e2s_state[i].setparity(PARAM_EPARITY);
    }
    else {                                          // Odd parity (== 2)
      e2s_state[i].setparity(PARAM_OPARITY);
    }
  }
  else {                                            // 2 stop bits
    e2s_state[i].setparity(PARAM_2STOP);
  }
}
```

The function, `serial_open()`, is called from the function that initializes the Ethernet-to-serial state machine, `e2s_init()`. It does all of the work necessary to open a serial port, including setting the number of data bits, stop bits, and parity.

The first statement opens the serial port using the baud rate value that was initialized in `main()`. In the rest of the code, the values for the other serial port parameters, which are also initialized in `main()`, are used to determine the correct bitmask to send to the serial port functions `serXdatabits()` and `serXparity()`. (The bitmasks, `PARAM_*`, are defined in the serial port library, `RS232.lib`.)

```
void e2s_init(void)
{
  auto int i;
  for (i = 0; i < sizeof(ports_config); i++) {
    e2s_state[i].state = E2S_INIT;       // Initialize the state

    // Initialize the serial function pointers
    switch (ports_config[i]) {
      case 'A':
        e2s_state[i].open = serAopen;
        e2s_state[i].close = serAclose;
        e2s_state[i].read = serAread;
        e2s_state[i].write = serAwrite;
        e2s_state[i].setdatabits = serAdatabits;
        e2s_state[i].setparity = serAparity;
        break;
      . . .

      default:
        // Error--not a valid serial port
        exit(-1);
    }
    // Open each serial port
    serial_open(i);
  }
}
```

The above function initializes the Ethernet-to-serial state machine: first by setting the variable that is used to travel around the state machine (`e2s_state[i].state`), then by setting the function pointers used to access the serial ports. For example, `serAopen()` is a function defined in `RS232.lib` that opens serial port A.

The `switch` statement has cases for serial ports B, C and D that are not shown here. They are functionally the same as the above code for serial port A. If the chip on the target board is a Rabbit 3000, there are cases for serial ports E and F as well. The default case is an error condition that will cause a run-time error if encountered.

The last statement in the `for` loop is a call to `serial_open()`. This function, which was described earlier, makes calls to the appropriate serial port functions using the function pointers that were just initialized.

```
void e2s_tick(void)
{
  auto int i;
  auto int len;
  auto tcp_Socket *sock;

  for (i = 0; i < sizeof(ports_config); i++) {
    sock = &(e2s_state[i].sock);
    switch (e2s_state[i].state) {
      case E2S_INIT:
        tcp_listen(sock, serial_ports[i].tcp_port, 0, 0, NULL, 0);
        e2s_state[i].state = E2S_LISTEN;
        break;

      case E2S_LISTEN:
        if (!sock_waiting(sock)) {
           //  The socket is no longer waiting
           if (sock_established(sock)) {
              //  The socket is established
              e2s_state[i].state = E2S_PROCESS;
           }
           else if (!sock_alive(sock)) {
              //The socket was established but then aborted by the peer
              e2s_state[i].state = E2S_INIT;
           }
           else {
           //socket was opened, but is now closing. Go to PROCESS state to read any data.
              e2s_state[i].state = E2S_PROCESS;
           }
        }
        break;
```

The function, `e2s_tick()`, drives the Ethernet-to-serial state machine. Each time this tick function is called, it loops through all of the serial ports, first grabbing the socket structure that associates a particular serial port with a TCP port, then determining which state is active for that TCP port. There are three states in the Ethernet-to-serial state machine, identified by:

- `E2S_INIT`
- `E2S_LISTEN`
- `E2S_PROCESS`

The first state, `E2S_INIT`, opens the socket with a call to `tcp_listen()` and then sets the state variable to be in the listen state. The next time the tick function is called the `E2S_LISTEN` state will execute. The state machine will stay in this listen state until a connection to the socket is attempted, a condition determined by a call to `sock_waiting()`.

As noted in the code comments above, once a connection is attempted there are several stages it can be in, which one will determine the next state of the Ethernet-to-serial state machine.

```
      case E2S_PROCESS:
         // Check if the socket is dead
         if (!sock_alive(sock)) {
            e2s_state[i].state = E2S_INIT;
         }
         // Read from TCP socket and write to serial port
         len = sock_fastread(sock, e2s_buffer, E2S_BUFFER_SIZE);
         if (len < 0) {                            //Error
            sock_abort(sock);
            e2s_state[i].state = E2S_INIT;
         }
         if (len > 0) {
            // Write the read data to the serial port--Note that for simplicity,
            // this code will drop bytes if more data has been read from the TCP
            // socket than can be written to the serial port.
            e2s_state[i].write(e2s_buffer, len);
         }
         else { /* No data read, do nothing */ }

         // Read from the serial port and write to the TCP socket
         len = e2s_state[i].read(e2s_buffer, E2S_BUFFER_SIZE,
            (unsigned long)0);

         if (len > 0) {
            len = sock_fastwrite(sock, e2s_buffer, len);
            if (len < 0) {                         //Error
               sock_abort(sock);
               e2s_state[i].state = E2S_INIT;
            }
         }
      break;
    }
  }
}
```

The `E2S_PROCESS` state checks to make sure the user did not abort the connection since the last time the tick function was called. If there was no abort, an attempt is made to read data from the socket buffer. If an error is returned from `sock_fastwrite()`, the connection is aborted and we go back to the init state the next time the tick function is called. If data was read, it is written to the serial port. If no data was read, then nothing happens.

Next an attempt is made to read data from the serial port. If data was read, it is then written out to the TCP socket. If the data read from the serial port was not written successfully to the TCP socket, the connection is aborted and we go back to the init state the next time the tick function is called.

If no data was read from the serial port, the process state will execute again the next time the tick function is called.

```
void main(void)
{
  auto int i;

  // Initialize the serial_ports data structure
  for (i = 0; i < sizeof(ports_config); i++) {
    serial_ports[i].tcp_port = 1234 + i;
    serial_ports[i].ser.port = ports_config[i];
    serial_ports[i].ser.baud = 9600;
    serial_ports[i].ser.databits = 8;
    serial_ports[i].ser.parity = 0;
    serial_ports[i].ser.stopbits = 1;
  }

  // Make a copy of the configuration options to be compared against when
  // the update functions are called
  memcpy(serial_ports_copy, serial_ports, sizeof(serial_ports));

  // Initialize the TCP/IP stack, HTTP server, and Ethernet-to-serial state machine.
  sock_init();
  http_init();
  e2s_init();

  // This is a performance improvement for the HTTP server (port 80),
  // especially when few HTTP server instances are used.

  tcp_reserveport(80);

  while (1) {
    // Drive the HTTP server
    http_handler();

    // Drive the Ethernet-to-serial state machine
    e2s_tick();
  }
}
```

In the `main()` function, the configuration parameters for the serial ports are given initial values which are then copied for later comparison. After initialization of the stack, the web server and finally the state machine, the while loop allows us to wait for a connection.

4.2 HTML Page for TCP to Serial Port Example

The file `config.zhtml` that was copied into memory at the beginning of this program contains the HTML form that is presented when someone contacts the IP address of the Rabbit that is running the above application code. `config.zhtml` uses the ZHTML scripting language that interacts with the code above to create the web interface to a Rabbit-based controller board.

File name: `Samples/tcpip/rabbitweb/pages/config.zhtml`

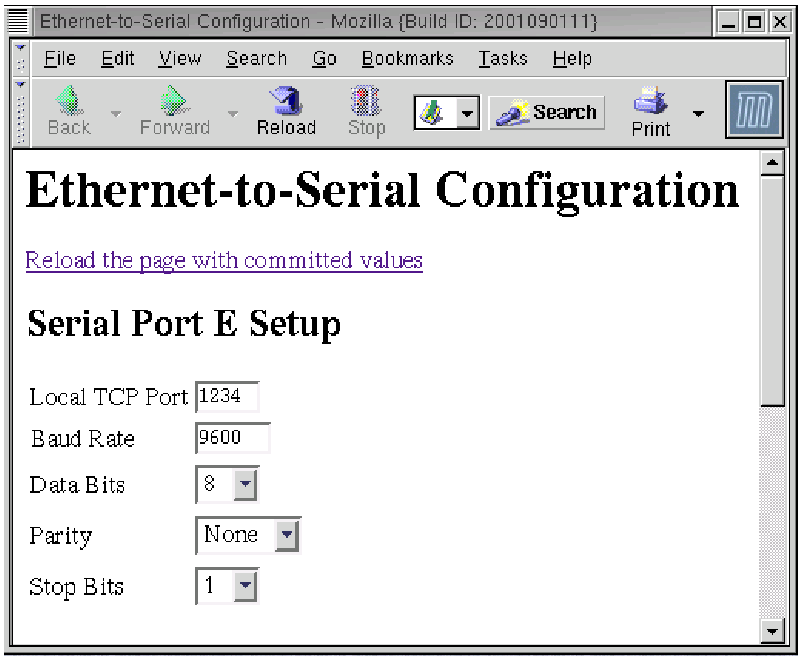
```
<HTML><HEAD>
<TITLE>Ethernet-to-Serial Configuration</TITLE></HEAD>
<BODY>

<H1>Ethernet-to-Serial Configuration</H1>

<A HREF="/index.zhtml">Reload the page with committed values</A>

<P>
<?z if (error()) { ?>
  There is an error in your data! The errors are both listed below
  and marked in <FONT COLOR="#ff0000">red</FONT>.
  <UL>
    <?z for ($A = 0; $A < count($serial_ports, 0); $A++)
    { ?>
      <?z if (error($serial_ports[$A].tcp_port))
      { ?>
        <LI>Serial Port <?z echo($serial_ports[$A].ser.port) ?>
        TCP port is in error (must be greater than 0)
        (committed value is
        <?z echo(@serial_ports[$A].tcp_port)?>)
      <?z } ?>

      <?z if (error($serial_ports[$A].ser.baud))
      { ?>
        <LI>Serial Port <?z echo($serial_ports[$A].ser.port) ?>
        baud rate is in error
        (<?z echo(error($serial_ports[$A].ser.baud)) ?>)
        (must be between 300 and 115200 baud)
        (committed value is
        <?z echo(@serial_ports[$A].ser.baud) ?>)
      <?z } ?>
    <?z } ?>
  </UL>
  <?z } ?>
```

After the usual opening lines of an HTML page, the first server-parsed tag we encounter is used with the call to `error()` to display a form submission error message, the same way we did in the humidity detector example in Section 1.2.2. Next is an example of a `for` loop used to print additional, more focused, error messages regarding the local TCP port number and the baud rate for each serial port. Again, `error()` is used with an `if` statement to verify the submission of particular web variables and display whatever error messages are chosen.

```
<FORM ACTION="/index.zhtml" METHOD="POST">
  <?z for ($A = 0; $A < count($serial_ports, 0); $A++) { ?>
    <H2>Serial Port <?z echo($serial_ports[$A].ser.port) ?> Setup
    </H2>
      <TABLE>
```

The form is defined next. Another `for` loop allows us to have the same form entries for each serial port in turn.When displayed without errors, the page looks like this:

Figure 4. Web Page Served by RabbitWeb



Ethernet-to-Serial Configuration

Reload the page with committed values

Serial Port E Setup

Local TCP Port	1234
Baud Rate	9600
Data Bits	8
Parity	None
Stop Bits	1

There are two tables, one for serial port E and, if you could scroll down in Figure 4, you would see that it is followed by a table for serial port F. Each table consists of five rows and two columns. Of the five rows, two have text entries and three have drop-down menus, one for each of the three selection variables defined in the Dynamic C application code shown above on page 30. We will not show the rest of the HTML code here because it is too repetitive and we have seen similar code in the humidity detector example. There are two lines, however, that are worth further discussion.

```
<INPUT TYPE="text"
  NAME="<?z varname($serial_ports[$A].tcp_port) ?>"
  SIZE=5 VALUE="<?z echo($serial_ports[$A].tcp_port) ?>">
```

The two text fields in the above form are created with INPUT tags like the one shown here. Recall that the NAME attribute does not allow the use of “[” or “].” The call to `varname()` solves that problem for us.

```
<SELECT NAME="<?z varname($serial_ports[$A].ser.parity) ?>">
  <?z print_select($serial_ports[$A].ser.parity) ?>
</SELECT>
```

The SELECT tag is used to create a drop-down menu in HTML, which is a convenient way to display a RabbitWeb selection-type variable.

Appendix A. RabbitWeb Reference

This appendix is the repository of some specialized details, such as the grammars that describe the scripting language and the Dynamic C enhancements. It is also intended as a way to quickly find descriptions of particular components of the RabbitWeb module.

A.1 Language Enhancements Grammar

Terminals are in bold, "[ ]" indicate optional parts, and "|" indicates an OR in the statement.

```
web-extension -> #web-statement |
                #web_groups-statement |
                #web_update-statement
```

```
#web-statement -> #web variable expression [authorization]
```

End-of-line escaping must be used for the #web statement to span lines.

```
variable -> case-insensitive-C-variable
expression -> modified-C-expression |
              select ( select-list )
select-list -> "string" [ = numeric-literal ] [, select-list]
```

`variable` is a C variable in the global scope. Due to details of how variables are transferred over HTTP, the variable name must be treated as case-insensitive. We can catch variables that conflict because of case-insensitivity at run-time.

`modified-C-expression` is a regular C expression, with an optional "$" symbol preceding C variables which is used to reference the newest value of the variable.

```
authorization -> [authorization] [auth-method] [valid-groups]
```

```
auth-method -> auth = auth-type-list
```

```
auth-type-list -> ssl | basic | digest [, auth-type-list]
```

```
valid-groups -> groups = valid-groups-list
```

```
valid-groups-list -> group [( group-rights )] [, valid-groups-list]
```

```
group-rights -> ro | rw
```

```
#web_groups-statement -> #web_groups groups-list
```

```
groups-list -> group-name [, groups-list]
```

`group-name` follows the same rules of a C variable name (it will, in fact, be in the namespace as a C variable).

```
#web_update-statement -> #web_update variable-list function-spec
```

```
variable-list -> variable [, variable-list]
```

`function-spec` is the name of a previously declared C function.

A.2 Configuration Macros

There are several macros that can be used when setting up a RabbitWeb server.

`USE_RABBITWEB`

Define to 1 to enable the HTTP extensions, including the ZHTML scripting language. Defaults to 0.

`RWEB_POST_MAXBUFFER`

This defines the size of a buffer that is created in xmem. This buffer stores POST requests that contain variable updates. Hence, this macro limits the size of POST requests that can be processed. Defaults to 2048.

`RWEB_POST_MAXVARS`

This macro defines the maximum number of variables that can be processed in a single POST request. That is, it limits the number of variables that can be updated in a single request. Each variable requires 20 bytes of root memory for bookkeeping information, so the total memory usage is 20 * `RWEB_POST_MAXVARS`. Defaults to 64.

`RWEB_ZHTML_MAXBLOCKS`

This macro determines the number of `if` and `for` blocks that can be nested in ZHTML. Each additional block allowed adds 11 bytes for each HTTP server instance (defined by `HTTP_MAXSERVERS`). Defaults to 4.

`RWEB_ZHTML_MAXVARLEN`

This defines the size of a root buffer that is used to store variable values, and hence limits the maximum length of a variable value. Only string values can be larger than 4 bytes, so realistically this macro only affects strings. Defaults to 256.

`RWEB_WEB_ERROR_MAXBUFFER`

This macro defines the size of a buffer in xmem that is used to hold `WEB_ERROR()` error messages. This buffer limits the total size of the error messages associated with a single form update. Defaults to 512.

A.3 Compiler Directives

The RabbitWeb compiler directives are summarized here.

`#web`

Registers a variable, array or a structure with the server. For more information, see Section 2.1.

The `#web` statement has several optional parts that can be used when a variable (or array or structure) is registered. The optional parts are:

- An error-checking expression to limit the acceptable values that are submitted. For more information see Section 2.2. A macro called `WEB_ERROR` can be included in the error-checking expression to associate a string with an error. For more information, see Section 2.2.1.

- The "auth=" parameter is a comma separated list of acceptable authentication methods. The possible choices are basic, digest and ssl. For more information, see Section 2.3.

- The "groups= " parameter is a comma separated list of user groups that are allowed access to the web variable (or array or structure). For more information, see Section 2.3.

One or more of the optional parts can be used in a `#web` statement.

`#web_groups`

This directive defines a web group. For more information, see Section 2.3.

`#web_update`

This directive identifies a user-defined function to call in response to a variable update. For more information, see Section 2.4.

A.4 ZHTML Grammar

Terminals are in bold, "[ ]" indicate optional parts, and "|" indicates an OR in the statement.

```
zhtml-tag -> <?z statement ?>
statement -> print-function | printf-function |
           varname-function | print_opt-function |
           print_select-function | if-statement | for-loop
```

```
print-function -> print( variable )
```

```
variable -> $ registered-variable | loop-variable
```

`registered-variable` is an array, structure or variable that is registered with the web server.

```
loop-variable -> $ A-Z
```

`loop-variable` is a one-letter variable (A-Z) defined in the `for` loop, and can be used as the index for an array.

```
printf-function -> printf ( printf-specifier , variable )
```

The `printf-specifier` is like a C printf specifier, except that it is limited to a single variable.

```
varname-function -> varname( variable )
print_opt-function -> print_opt( variable , number ) |
                      print_opt( variable , loop-variable )
print_select-function -> print_select( variable )
```

```
count-expression -> count( variable, number ) | count( variable )
```

Note that in the first option `variable` is an array; in the second option it is a selection-type variable.

```
numeric-expression -> loop-variable | integral-variable |
                    count-expression | numeric-literal
```

`integral-variable` refers to a registered `#web` variable of integral (`int` or `long`, `signed` or `unsigned`) type.

```
if-statement -> if ( if-expression ) { html-code }
```

```
if-expression ->numeric-expression operator numeric-expression |
                 [ ! ] error( variable ) |
                 [ ! ] auth( variable , " group-rights " ) |
                 [ ! ] updating( )
```

```
operator -> == | != | > | < | >= | <=
```

```
group-rights -> ro | rw
```

```
for-loop -> for ( loop-variable = numeric-expression ;
                loop-variable operator numeric-expression ;
                loop-variable for-inc ) { html-code }
for-inc -> ++ | -- | += numeric-expression | -= numeric-expression
```

`ro` stands for read-only

`rw` stands for write-only

A.5 RabbitWeb Functions

This section lists all of the functions that can be called within ZHTML tags.

auth()

This function is used to check if a user has authorization for accessing a specific variable.

```
<?z if (auth($foo, "rw")) { ?>
   You have read-write access to the variable foo.
<?z } ?>
```

This function can be preceded by "!" (the not operator).

count()

This function is for arrays and selection-type variables.

If the first parameter is an array, the second parameter specifies an array dimension. For a one-dimensional array, the second parameter must be zero. For a two-dimensional array, the second parameter must be zero or one. And so on. If the first parameter is an array, the return value of the function is the upper bound for the specified array dimension.

If the first parameter is a selection variable, there is no second parameter. The `count()` function returns the number of options for a selection variable.

The return value of `count()` can be used in a `for` loop to cycle through all elements of an array.

```
<?z for ($A = 0; $A < count($foo, 0); $A++) { ?>
```

echo(), print()

These are display functions to make web variables visible on an HTML page. `They` display the variable passed to them using a default conversion specifier. The function `echo()` is an alias for `print()`.

```
<?z print($foo) ?>
```

error()

The `error()` function can be called both with and without a parameter. If it is called without a parameter it will return TRUE if there were any errors in the form submission and FALSE otherwise. To call `error()` with a parameter, you must pass it the name of a web variable. The function will return TRUE if that variable did not pass its error check, and FALSE otherwise.

It can be used to print out the `WEB_ERROR()` message:

```
print(error($foo))
```

`printf()`

This is a display function to make web variables visible on an HTML page. With `printf()` you can display a variable of type `int` or `long`:

```
<?z printf(“%ld”, $long_foo) ?>
```

`print_opt()`

This is a display function to make selection-type web variables visible on an HTML page. It takes two parameters. The first parameter is a selection-type variable and the second parameter is the index into the list of possible values for the selection-type variable.

```
<?z print_opt($select_var, $A) ?>
```

`print_select()`

This is a display function to make selection-type web variables visible on an HTML page. `It` automatically generates the option list for a given selection variable:

```
<?z print_select($select_var) ?>
```

`selected()`

The `selected()` function takes two parameters. The first parameter is a selection variable and the second parameter is an integer index into the array of options for the specified selection variable. The function returns TRUE if the option indicated by the index parameter matches the currently selected option, and FALSE if it doesn’t.

For example, to iterate through all possible values of a selection-type variable and output the appropriate “<OPTION>” or “<OPTION SELECTED>” tags, something like the following can be done:

```
<?z for ($A = 0; $A < count($select_var, 0); $A++) { ?>
   <OPTION
   <?z if (selected($select_var, $A)) { ?>
      SELECTED
   <?z } ?>
   >

<?z print_opt($select_var, $A) ?>
```

The page `Samples/tcpip/rabbitweb/pages/selection.zhtml` uses the `selected()` function.

`updating()`

This function can be used with an `if` statement to test whether the current page is being displayed as the result of a POST request instead of a GET request. This is useful to redirect to another page on a successful form submission. Use this function as follows:

```
<?z if (updating()) { ?>
   <?z if (!error()) { ?>
      <META HTTP-EQUIV="Refresh" CONTENT="0;
          URL=http://yoururl.com/">
   <?z } ?>
<?z } ?>
```

This function can be preceded by "!" (the not operator).

`varname()`

This is a convenience function that gets around the limitation of no square brackets in the NAME attribute of the INPUT tag in HTML.

```
<INPUT TYPE="text" NAME="<?z varname($foo[3]) ?>
   "VALUE=" <?z echo($foo[3]) ?>">
```