CHAPTER 19

The Form Processor

OVERVIEW

Our form processing application allows a site administrator to process the results of Web-based forms in a number of ways. The script can be used to generate dynamic E-mail or to create and build a log file or database to track usage of HTML-based forms.

For example, a site administrator could use this script to help process customer feedback. The administrator would first create an HTML form–based questionnaire or other feedback form using standard HTML. Clients would fill out and submit feedback to the form processing script. The script would then send the feedback results to the administrator via E-mail as well as add it to a growing log of all feedback results, which could then be imported to a database application for further processing.

The processing of forms is configured by taking advantage of various hidden form tags included in the HTML form created by the administrator, so very little coding knowledge is needed to use this script. Furthermore, because

the workings of the script are defined in the HTML form rather than in the Perl script, the HTML form author can use this one script to process all of the forms, no matter how different they are. The administrator need only modify the hidden variables that define how the script should act. Thus, we need not write separate CGI form processors for each of our forms. Instead, we can provide one centralized solution for all the forms on our system.

Installation and Usage

Once unarchived, the application should expand into the root directory **Form_processing**. Figure 19.1 shows the directory structure, including a summary of the correct permissions for files and directories.

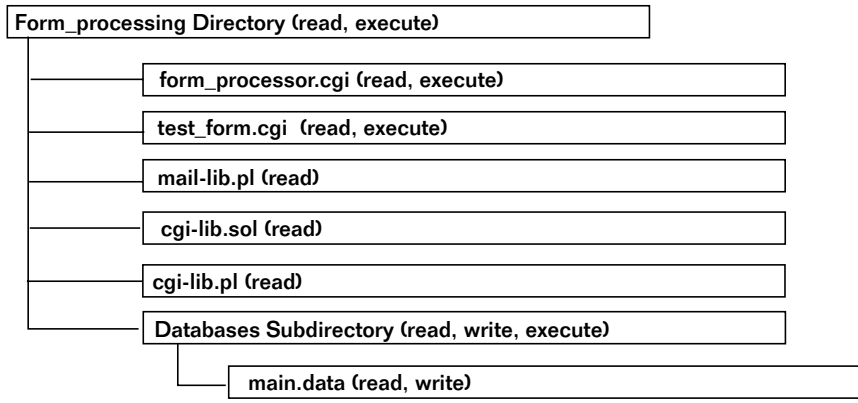

Figure 19.1 *Form processor directory structure.*

Form_processing, the root directory, contains five files (**form_processor.cgi**, **cgi-lib.pl**, **cgi-lib.sol**, **mail-lib.pl**, **test_form.cgi**) and one subdirectory (**Databases**), which contains one file (**main.data**). **Form_processing** must be executable and readable by the Web server.

form_processor.cgi is the main application and must be readable and executable by the Web server. This is the script that processes incoming data.

cgi-lib.pl, **cgi-lib.sol**, and **mail-lib.pl** are all discussed in Part Two. **form_processor.cgi** uses **cgi-lib.pl** to gather form data, **cgi-lib.sol** to create a

lock file, and **mail-lib.pl** to E-mail on request. Each of these files must be readable by the Web server.

test_form.cgi is not necessary for the working of the program, but it is included on the accompanying CD-ROM as an example of a form interface to the script. It must be readable and executable by the Web server.

It is not necessary to have the HTML form generated by a CGI script. However, to keep everything in one directory structure for the example, we have made a script to generate a form instead of using basic HTML. Recall that Web browsers may not read HTML files in some cgi-bin directories.

Finally, **Databases** is the subdirectory in which **form_processor.cgi** will write when it is asked to append to database files that log the usage of the forms. Its permissions should allow the Web server to read, write, and execute. The directory can contain as many database files as you have HTML forms that write to them.

main.data is a sample database that we have included as an example of a datafile that logs usage of the script. It should be readable and writable by the Web server.

Server-Specific Setup and Options

The HTML Form

Before you run this script, you must prepare your HTML forms to send enough information so that the script can be most efficient. Specifically, you must send the information as hidden variables along with the client-defined information. The script uses these hidden variables to answer internal questions about how it should process the data it is receiving.

This script takes 17 hidden variables, which must be included in any HTML form (or, in the case of our example, **test_form.cgi**) that uses the script. The hidden variables are as follows.

`mailto` is the E-mail address of the person who should receive the results of form submission. This field is required.

`html_response` is the text of the HTML response that you want the client to receive after submitting information. This response is not required, but it's a good idea because it makes your site more user-friendly.

`email_subject` is the subject that you want to appear on the E-mail sent to the person who receives the results of form submission. This is not required but is also a good touch.

`variable_order` is the order in which you want all the variables to appear on the E-mail sent to the person who receives the results of form submission. The format of this tag is a pipe-delimited list, and every input field in your form must be represented in the list (see the following example). This is a required field.

`required_variables` is a list of variables that are required. If clients do not fill out information for these input fields, they get an error message requesting that they go back and fill out all the form fields. This field is not required.

`url_of_this_form` is the URL of the form that is being used to submit information; if clients do not submit all the required fields, they can link back to the form to try again. This variable is not required unless you have set a value for `required_variables`.

`background`, `bgcolor`, `text_color`, `link_color`, `vlink_color`, and `alink_color` relate to the body tag for all the HTML responses. None of them is required, but what is the good of having a fancy Web-based GUI without cool colors and stuff?

`response_title` is the title that you want to appear on the HTML response for successes. This title is not required, but it is suggested because it is good HTML style.

`return_link_url` is the URL of the page that defines the hyperlink that people click from the HTML response page after they have submitted their information. Again, this is not required, but it's recommended because it is best to provide the client with a simple way out of the form routine and back into your Web site.

`return_link_name` is the text of the clickable link related to `return_link_url`. This text is required only if you set `return_link_url` equal to something.

`database_name` is the path (relative to this script) of the datafile being used to store information submitted by this form.

`database_delimiter` is the delimiter your database uses to divide fields. Common ones are comma (,), colon (:), or pipe (|). We recommend the pipe symbol because it is rarely used in client-submitted data.

These hidden variables define how this script should perform, to whom it should send E-mail, how to sort the variables in the E-mail message, how to respond to the client, and so on. Following is an example of the short form, contained in the file **test_form.cgi**, that is displayed by pointing to a URL such as the following:

```
http://www.foobar.com/cgi-bin/Form_processing/test_form.cgi
```

You can use this form as a base example from which to create others.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
print <<" end_of_html";
<HTML><HEAD><TITLE>Test Form</TITLE></HEAD><BODY>
<FORM ACTION = "http://www.foobar.com/cgi-bin/form_processor.cgi"
METHOD = "POST">
<INPUT TYPE = "hidden" NAME = "mailto"
       VALUE = "selena@foobar.com">
<INPUT TYPE = "hidden" NAME = "html_response"
       VALUE = "Thank you for submitting your personal
                information.">
<INPUT TYPE = "hidden" NAME = "email_subject"
       VALUE = "Form Test (From Scripts Page)">
<INPUT TYPE = "hidden" NAME = "variable_order"
       VALUE = "name|email|sign|purpose">
<INPUT TYPE = "hidden" NAME = "required_variables"
       VALUE = "name|email">
<INPUT TYPE = "hidden" NAME = "url_of_this_form"
       VALUE = "/cgi-
       bin/Selena/Form_processing/test_form.cgi">
<INPUT TYPE = "hidden" NAME = "background" VALUE = "">
<INPUT TYPE = "hidden" NAME = "bgcolor"
       VALUE = "FFFFFF">
<INPUT TYPE = "hidden" NAME = "text_color"
       VALUE = "000000">
<INPUT TYPE = "hidden" NAME = "link_color" VALUE = "">
<INPUT TYPE = "hidden" NAME = "vlink_color" VALUE = "">
```

```
<INPUT TYPE = "hidden" NAME = "alink_color" VALUE = "">
<INPUT TYPE = "hidden" NAME = "response_title"
       VALUE = "Thank You">
<INPUT TYPE = "hidden" NAME = "return_link_name"
       VALUE = "Selena Sol's Script Archive">
<INPUT TYPE = "hidden" NAME = "return_link_url"
       VALUE = "http://www.eff.org/~erict/Scripts/">
<INPUT TYPE = "hidden" NAME = "database_name"
       VALUE = "Databases/main.data">
<INPUT TYPE = "hidden" NAME = "database_delimiter"
       VALUE = ",">
<P><B>Name (Required):</B><BR>
<INPUT TYPE = "text" NAME = "name" SIZE = "35"
       MAXLENGTH = "35">
<P><B>Email (Required):</B><BR>
<INPUT TYPE = "text" NAME = "client_email" SIZE = "35" MAXLENGTH =
"35">

<P><B>Astrological Sign:</B><BR>
<INPUT TYPE = "text" NAME = "sign" SIZE = "35"
       MAXLENGTH = "35">

<P><B>Purpose in Life: </B><BR>
<INPUT TYPE="text" NAME="purpose" SIZE="35" MAXLENGTH="35">

<P><CENTER>
<INPUT TYPE = "submit" VALUE = "Tell me about yourself!">
</CENTER>
</FORM></BODY></HTML>
end_of_html
```

On the Web, the previous HTML form code looks like Figure 19.2.

In this example, the script E-mails selena@foobar.com when someone submits form data. It uses "Form Test (From Scripts Page)" as the subject of the E-mail and returns the client a little blurb with a pointer back to Selena Sol's Script Archive. Finally, the script appends the database **main.data** with a comma-delimited database row representing the four fields created here.

The body of the E-mail message contains the variables and their client-defined values in a specific order: name, E-mail address, sign, and purpose of life. I'll explain later how the script determines the order, but

I want to draw your attention to the way the order is predefined in the HTML form. It is pipe-delimited, and all variables must appear in the order you want them to be processed.

Figure 19.2 *HTML form interface to form_processor.cgi.*

The Setup Variables

Just as the HTML form must be coded to send specific bits of information, the main script, **form_processor.cgi**, must be prepared to process those bits. Specifically, you must define certain variables within the main script. For ease, these variables are gathered near the top of the script.

`$email_of_sender` is the E-mail address of the account from which mail should be sent to the form processing administrator. This variable is

not necessarily required, because the script is set to understand the variable name `client_email`, which can be included in the HTML form. If `client_email` is defined, the script will send the mail as if it were coming from the client-submitted E-mail. (You may want to remind the client that this must be a valid E-mail address.) If `client_email` is not defined, however, you should define `email_of_sender` here so that `sendmail` will be able to function.

`$your_server_name` is the name of your Web server. We use this value to make sure that the form using this script is actually on your server. After all, we wouldn't want everyone in the universe using your server for form processing. If you are not sure what this value should be, it is probably the "www.foobar.com" in your URL address. This variable is required only if you want the greater security.

Be careful if you are using a virtual server. The name may be an alias, and the script might not even process your forms! If you are unsure of what all this means, the best idea is to check with your site administrator, explain what you are trying to do, and ask for the valid server name relative to your Web server.

`$restricted_use` is a flag that we use to activate the security just described. If you set this variable to `no`, it means that this script will process forms from any Web server on the net. This variable is required only if you want greater security.

`$location_of_cgi_lib`, `$location_of_cgi_sol`, and `$location_of_mail_lib` are the locations of the library files that should accompany this script.

`$should_i_mail` and `$should_I_append_a_database` are flags that you set depending on how you want the results of the form processing to be gathered. If you set `$should_i_mail` to `yes`, the form data will be sent to the administrator you have defined in the hidden form variables.

Similarly, if you set `$should_i_append_a_database` to `yes`, the form information will be saved in a database style that you can import to whatever database application you use. The name of the database is defined with hidden variables, as is the database delimiter.

Running the Script

Once you have configured the setup file to the specifics of your server setup and have prepared a form to be processed, you can try out the scripts by creating a hyperlink to the location of the form you are using (in this case, we reference the form-generating script):

```
<A HREF = "http://www.foobar.com/cgi-
bin/Form_processing/test_form.cgi">Sample Feedback form</A>
```

Design Discussion

Once the client has typed the information into the form fields you created in your HTML form, the client will press the **submit** button and **form_processor.cgi** will be responsible for handling the input as you have instructed it with the hidden variables. The logic of the script is depicted in Figure 19.3.

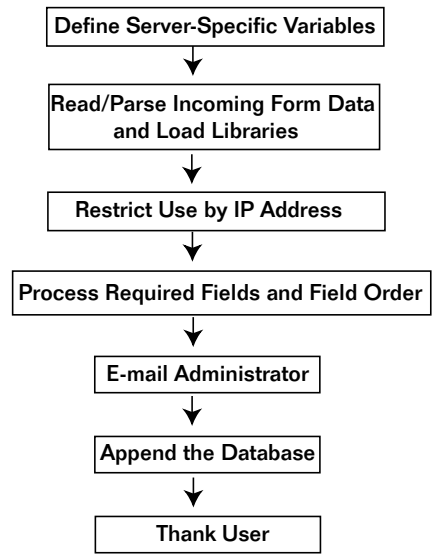

Figure 19.3 *Script logic.*

The script begins by starting the Perl interpreter and printing the HTTP header.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
```

Defining Server-Specific Variables

Next, the script defines the server-specific variables discussed in the "Installation and Usage" section.

```
$email_of_sender = "admin\@foobar.com";
$your_server_name = "www.foobar.com";
$restricted_use = "no";
$location_of_cgi_lib = "./cgi-lib.pl";
$location_of_cgi_sol = "./cgi-lib.sol";
$location_of_mail_lib = "./mail-lib.pl";
$should_i_mail = "yes";
$should_I_append_a_database = "yes";
```

Loading Supporting Files and Reading and Parsing Form Data

Next, the script loads the supporting files and uses **cgi-lib.pl** to read the incoming form data. The script sends `*form_data` as a parameter to the subroutine `ReadParse` in **cgi-lib.pl** so that the associative array of form keys and values comes back with a descriptive name rather than `%in`.

```
require "$location_of_cgi_lib";
require "$location_of_cgi_sol";
require "$location_of_mail_lib";
&ReadParse(*form_data);
```

Restricting Use by IP Address

Next, the script determines the location of the form that is requesting its attention. It does this by accessing the environment variable `$ENV{'HTTP_REFERER'}`, which is equal to the URL of the form in question (http://www.foobar.com/Feedback/feedback.html). The script then `splits` that

value into separate variables for every occurrence of "/" to isolate "www.foo-bar.com," which it compares to the value of `$your_server_name`. Thus, `$referring_server` is the only variable in the split that we give a hoot about. If `$referring_server` and `$your_server_name` are not the same, the script knows that it is being accessed by a form on another server.

```
($http, $empty, $referring_server, @path) = split (/\//,
$ENV{'HTTP_REFERER'});
```

If the `$restricted_use` has been set to `yes`, and if the `$referring_server` is not the same as `$your_server_name`, it means that we have had an illegal attempted access and the script should stop processing the form.

```
if ($restricted_use eq "yes")
  {
  if ($referring_server ne "$your_server_name")
    {
    &html_header("Form Error - Wrong Server");
    print "I'm sorry, you are not allowed to use this
           form processing script from a server other
           than $your_server_name<P>";
    print "As far as I can tell, you are coming from:
           $referring_server";
    print "<P>$restricted_use = restricted use<BR>";
    print "</BODY></HTML>";
    exit;
    }
  }
```

Processing Required Fields and Field Order

Once the script is sure that the form is within the security wall, the script breaks up the `$variable_order` variable that was sent from the form. `$variable_order` should look something like this:

```
name|email|sign|purpose|
```

This variable has been defined by the person who wrote the form that calls this script. The script then `splits` the variable into array elements every time it sees a pipe (|) so that `@form_variables` might look like this:

```
("name", "email", "sign", "purpose").
```

The script splits the variable using the following code:

```
@form_variables = split (/\|/, $form_data{'variable_order'});
```

It then repeats the same thing it did for variable_order, but this time for required_variables.

```
@required_variables = split (/\|/, $form_data{'required_variables'});
```

Once it has the list of required variables, the script checks to make sure that the client has submitted values for each of those variables. If any values are missing, the script produces a note explaining the problem along with a list of required variables. The script also adds a pointer back to the form.

```
foreach $variable (@required_variables)
  {
  if ($form_data{$variable} eq "")
    {
    &html_header("Form Error - Missing Data");
    print "Woops, I'm sorry, the following fields are
           required: ";
    print "<BLOCKQUOTE>";
    foreach $variable (@required_variables)
      {
      print "$variable<BR>";
      } # End of foreach $variable (@required_variables)
    print "</BLOCKQUOTE>";
    print "Please <A HREF =
         \"$form_data{'url_of_this_form'}\">go back to
          the form</A> and make sure you fill out all
          the required information.";
    print "</BODY></HTML>";
    exit;
    } # End of if ($form_data{$variable} eq "")
  } # End of foreach $variable (@required_variables)
```

If you edit this note, make sure to escape any occurrences of @ or " with a backslash:

```
print "<A HREF = \"mailto:admin\@foobar.com\">admin\@foobar.com</A>";
```

E-Mailing Form Results to the Administrator

If the client submitted information for all the required fields, the script begins processing the data. It first checks to see whether it is supposed to generate E-mail.

```
  if ($should_i_mail = "yes")
    {
```

If `$should_i_mail` has been set to `yes` in the define variables area, the script sends the results of the form to the administrator defined in the HTML hidden form data.

```
  if ($form_data{'mailto'} ne "")
    {
    $email_to = "$form_data{'mailto'}";
    }
$email_subject = "$form_data{'email_subject'}";
```

Then the script begins building the body of the E-mail message. `$email_body` is used to store the information that the script will mail. First, the script notes the time using a little routine written by Matt Wright.

```
$email_body = "This data was submitted on: ";
$email_body .= &get_date;
$email_body .="\n\n";
```

The use of ".=" tells the script to append the new information to the end of the old. Thus, `$email_body` keeps getting longer and longer as new information is tagged to the end of the old.

Next, for every form variable, the script adds to `$email_body` the variable name and its values in the order specified by `$form_data{'variable_order'}`.

```
foreach $variable (@form_variables)
  {
  $email_body .= "$variable = $form_data{$variable}\n";
  }
```

Also, if the author of the HTML form has used the special form variable name `client_email` as one of the hidden input field names, the script changes `$email_of_sender` to the client-submitted E-mail address.

```
if ($form_data{'client_email'} ne "")
  {
  $email_of_sender = "$form_data{'client_email'}";
  }
```

This change is not necessary, but from experience we've learned that it is nice to have the E-mail sent "from" the client submitting the information so that reply E-mail goes straight to the client rather than the form administrator.

Now the script uses the `send_mail` routine in **mail-lib.pl** to send the data.

```
&send_mail("$email_of_sender", "$email_to",
           "$email_subject", "$email_body");
} # End of  if ($should_i_mail = "yes")
```

The administrator will receive an E-mail message as shown in Figure 19.4.

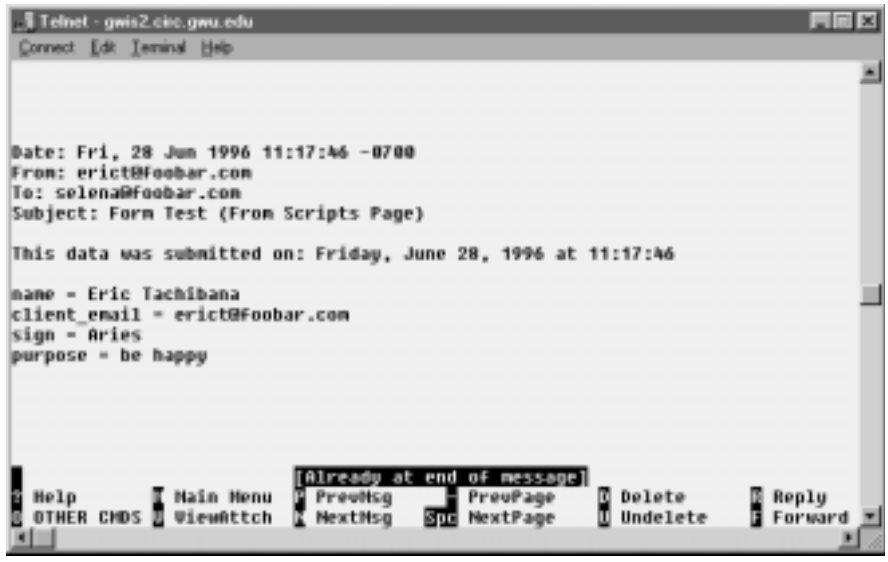

Figure 19.4 *Administrative E-mail.*

Appending to the Form Results Database

If the $should_I_append_a_database has been set to yes, the script also needs to append, to the database specified in the hidden field, the database_name specified in the form.

```
if ($should_I_append_a_database = "yes")
  {
```

The script first checks to see whether the database exists.

```
$database = "$form_data{'database_name'}";
if (-e $database)
  {
```

If the database exists, the script sets the $counter variable to zero. $counter is used to keep track of the number of fields sent from the form so that the script will know when the database row ends.

```
$counter = "0";
```

Then, for every field sent from the form, the script increments counter by 1.

```
foreach $variable (@form_variables)
   {
$counter++;
```

Next, the script appends the value of the variable to the growing $database_row variable.

```
$database_row .= "$form_data{$variable}";
```

If this is not the last item in the row, the script also divides each field with the database delimiter. When counter equals the number of elements in @form_variables, the script knows that it is at the end of the row and need not append another delimiter.

```
if ($counter <= @form_variables)
  {
  $database_row .= "$form_data{'database_delimiter'}";
```

```
  }
} # End of foreach $variable (@form_variables)
```

Finally, the script appends the new row, ending it with a newline character. To do so, it first creates a lock file using `GetFileLock` in **cgi-lib.sol**. It then opens the database for appending (`>>`), prints to the database, and closes it. Then the script releases the lock file.

```
 &GetFileLock ("$database.lock");
 open (DATABASE, ">>$database");
 print DATABASE "$database_row\n";
 close (DATABASE);
 &ReleaseFileLock ("$database.lock");
} # End of if (-e $database)
```

If the database file did not exist, however, the script sends an error message to the user. Most likely, the hidden form variable is not correct (the path is wrong), the permissions of the file are not set to be readable and writable by the Web server, or its directory is not to be readable, writable, and executable by the Web server.

```
else
  {
  &html_header("Form Error - Database Does Not Exist");
  print "I'm sorry, I am having trouble finding the
         database that this information should be sent
         to.  Please contact <A HREF =
\"mailto:$form_data{'mailto'}\">$form_data{'mailto'}</A>
         and let them know that there has been a
         problem.  Thank you very much.";
  print "</BODY></HTML>";
  exit;
  }
} # End of if ($should_I_append_a_database = "yes")
```

Thanking the User

Then the script prints a response to the client.

```
&html_header($form_data{'response_title'});
print "$form_data{'html_response'}";
print "<P>You sent us the following data:<P>";
foreach $variable (@form_variables)
```

```
  {
  print "$variable = $form_data{$variable}<BR>";
  }
print "<P>Please return to ";
print "<A HREF =
\"$form_data{'return_link_url'}\">$form_data{'return_link_name'}</A>";
print "</BLOCKQUOTE>";
print "</BODY></HTML>";
exit;
```

On the Web, the client will get something like the message shown in Figure 19.5.

Figure 19.5 *Response to client.*

The get_date Subroutine

The `get_date` subroutine, written by Matt Wright, is used to find the current date.

```
sub get_date
  {
  @days = ('Sunday', 'Monday', 'Tuesday', 'Wednesday',
           'Thursday', 'Friday', 'Saturday');
  @months = ('January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'November',
             'December');
```

The subroutine uses the `localtime` command to get the current time, splitting it into variables.

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
localtime(time);
```

It then formats the variables and assigns them to the final `$date` variable.

```
if ($hour < 10) { $hour = "0$hour"; }
if ($min < 10) { $min = "0$min"; }
if ($sec < 10) { $sec = "0$sec"; }
```

The months and days arrays are used to format the date into a more human-readable form.

```
$date = "$days[$wday], $months[$mon] $mday, 19$year at
$hour\:$min\:$sec";
}
```

The html_header Subroutine

`html_header` is used to generate the header for all the HTML outputs.

```
sub html_header
  {
```

The subroutine begins by assigning to the local variable `$title` the `title` variable coming in from the subroutine call.

```
local($title) = @_;
```

It then prints the header.

```
print "<HTML><HEAD><TITLE>$title</TITLE></HEAD>";
print "<BODY";
```

Next, it creates the <BODY> tag according to the instructions defined in the hidden form variables.

```
if ($form_data{'background'} ne "")
    {
    print " BACKGROUND = \"$form_data{'background'}\"";
    }
if ($form_data{'bgcolor'} ne "")
    {
    print " BGCOLOR = \"$form_data{'bgcolor'}\"";
    }
if ($form_data{'text_color'} ne "")
    {
    print " TEXT = \"$form_data{'text_color'}\"";
    }
if ($form_data{'link_color'} ne "")
    {
    print " LINK = \"$form_data{'link_color'}\"";
    }
if ($form_data{'vlink_color'} ne "")
    {
    print " VLINK = \"$form_data{'vlink_color'}'}\"";
    }
if ($form_data{'alink_color'} ne "")
    {
    print " ALINK = \"$form_data{'alink_color'}\"";
    }
 print "><H2><CENTER>$title</CENTER></H2><BLOCKQUOTE>";
 }
```

Notice in the last line that we include the end of the <BODY> tag as the first character printed.