

CHAPTER TWELVE

THE HTML LIBRARY

The features of this library are dealt with in great detail in Chapter 2. Thus, we will simply discuss the code here.

Creating a Product Page Header

The HTML Library begins with the **product_page_header** subroutine. **product_page_header** is used to display the shared HTML header used for database-based product pages. It takes one argument, **\$page_title**, which will be used to fill the data between the `<TITLE>` and `</TITLE>`. Typically, this value is determined by **\$sc_product_display_title** in **web_store.setup**. The subroutine is called with the following syntax:

```
&product_page_header("Desired Title");
```

The subroutine begins by assigning the incoming argument to the local variable **\$page_title**.

```
sub product_page_header
{
    local ($page_title) = @_;
```

Then, it assigns the text of all of the hidden fields that may need to be passed as state information to **\$hidden_fields** using the **make_hidden_fields** subroutine, which will be discussed later in this chapter.

```
local ($hidden_fields) = &make_hidden_fields;
```

Next, the HTML code is sent to the browser, including the page title and the hidden fields dynamically inserted.

```
print qq~
<HTML>
<HEAD>
<TITLE>${page_title}</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<FORM METHOD = "post" ACTION = "${sc_main_script_url}">

$hidden_fields
~;
```

Next, we will grab `$sc_product_display_header` which is a preformatted string defined in `web_store.setup` and use `printf` to put the entire contents of `@sc_db_display_fields` in place of the format tags (`%s`). The purpose of this will be to display the header categories which products will follow. Consider the following example from `web_store.setup.db`:

```
$sc_product_display_header = qq!
<TABLE BORDER = "0">
<TR>
<TH>Quantity</TH>
<TH>%s</TH>
<TH>%s</TH>
</TR>
<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>!;

@sc_db_display_fields = ("Image (If appropriate)",
                        "Description");
```

In this case, the strings **“Image (If appropriate)”** and **“Description”** will be substituted by the `printf` function for the two **“%s”** tags in the TABLE header defined in `$sc_product_display_header`.

```
printf($sc_product_display_header,
      @sc_db_display_fields);
}
```

Creating a Product Page

The `product_page_footer` subroutine is used to generate the HTML page footer for database-based product pages. It takes two arguments, `$db_status` and `$total_rows_returned` and is called with the following syntax:

```
&product_page_footer($status,$total_row_count);
```

The subroutine begins by defining a few local variables. `$db_status` gives us the status returned from the database search engine and `$total_rows_returned` gives us the actual number of rows returned. `$warn_message`, which is first initialized to be blank, will be used to generate a warning that the customer should narrow their search in case too many rows were returned.

```
sub product_page_footer
{
    local($db_status, $total_rows_returned) = @_;
    local($warn_message);
    $warn_message = "";
```

If the database returned a status, the script checks to see if it was like the string `max.*row.*exceed`. If so, the script lets the customer know that they need to narrow their search.

```
if ($db_status ne "")
{
    if ($db_status =~ /max.*row.*exceed.*i)
    {
        $warn_message = qq!
<CENTER>
<BLOCKQUOTE>
Your query returned $total_rows_returned. This is
more than the maximum we allow
($sc_db_max_rows_returned). You will need to
restrict your query further.
</BLOCKQUOTE></CENTER><P>!;
    }
}
```

Then the script displays the footer information defined with `$sc_product_display_footer` in `web_store.setup` and adds the final basic HTML footer. Notice that one of the Submit buttons, “Return to Frontpage,” is isolated into the `$sc_no_frames_button` variable. This is because in the Frames version, we do not want that option as it will cause an endlessly fracturing Frame system. Thus, in a Frame store, you would simply set `$sc_no_frames_button` to “” and nothing would print here. Otherwise, you may include that button in your footer for ease of navigation. The variable itself is defined in `web_store.setup`. The script also will print the warning message if there is a value for it.



NOTE

A discussion of recursive Frames appears in Chapter 6 along with an example.

```
print qq~
$sc_product_display_footer
<P>
$warn_message

<CENTER>
<INPUT TYPE = "submit" NAME = "add_to_cart_button"
      VALUE = "Add Items to my Cart">
<INPUT TYPE = "submit" NAME = "modify_cart_button"
      VALUE = "View/Modify Cart">
$sc_no_frames_button
<INPUT TYPE = "submit" NAME = "order_form_button"
      VALUE = "Checkout Stand">
</FORM>
</CENTER>
</BODY>
</HTML>~;
}
```

Defining the HTML Search Page Footer

The `html_search_page_footer` subroutine is used to generate the HTML footer for HTML-based product pages when the script must perform a keyword search and generate a list of hits. It is called with no arguments with the following syntax:

```
&html_search_page_footer;
```

Notice again the use of `$sc_no_frames_button` in place of the “Return to Frontpage” button as discussed in the last section.

```
sub html_search_page_footer
{
  print qq!
  <CENTER>
  <INPUT TYPE = "submit" NAME = "modify_cart_button"
        VALUE = "View/Modify Cart">
  $sc_no_frames_button
  <INPUT TYPE = "submit" NAME = "order_form_button"
        VALUE = "Checkout Stand">
  </FORM>
  </CENTER>
  </BODY>
  </HTML>;
}
```

Defining the Standard Page Header

The `standard_page_header` subroutine is used to generate a standard HTML header for pages within either the HTML-based or Database-based stores. It takes a single argument, the title of the page to be displayed, and is called with the following syntax:

```
&standard_page_header("TITLE");
```

Note, as in the case of `product_page_header`, all state variables must be passed as hidden fields. These hidden fields are generated by `make_hidden_fields` discussed later.

```
sub standard_page_header
{
  local($type_of_page) = @_;
  local ($hidden_fields) = &make_hidden_fields;
  print qq!
  <HTML>
  <HEAD>
  <TITLE>$type_of_page</TITLE>
  </HEAD>
  <BODY>
  <FORM METHOD = "post" ACTION = "$sc_main_script_url">
  $hidden_fields!;
}
```

Defining the Modify Form Footer

The `modify_form_footer` subroutine is used to generate the HTML footer code for the Modify Quantity of Items in the Cart form page. It takes no arguments and is called with the following syntax:

```
&modify_form_footer;
```

As usual, we will admit the “Return to Frontpage” button only if we are not using frames by defining it with the `$sc_no_frames_button` in `web_store.setup`.

```
sub modify_form_footer
{
  print qq!
  <P>
  <INPUT TYPE = "submit" NAME = "submit_change_quantity_button"
        VALUE = "Submit Quantity Changes">
  <INPUT TYPE = "submit" NAME = "continue_shopping_button"
        VALUE = "Continue Shopping">
  $sc_no_frames_button
  <INPUT TYPE = "submit" NAME = "order_form_button"
        VALUE = "Checkout Stand">
  </FORM>
  </CENTER>
  </BODY>
  </HTML>!;
}
```

Defining the Delete Form Footer

The `delete_form_footer` subroutine is used to generate the HTML footer code for the Delete Items from the Cart form page. It takes no arguments and is called with the following syntax:

```
&delete_form_footer;
```

As usual, we will admit the “Return to Frontpage” button only if we are not using frames by defining it with the `$sc_no_frames_button` in `web_store.setup` .

```
sub delete_form_footer
{
  print qq!
  <P>
  <INPUT TYPE = "submit"
        NAME = "submit_deletion_button"
        VALUE = "Submit Deletion">
  <INPUT TYPE = "submit"
        NAME = "continue_shopping_button"
        VALUE = "Continue Shopping">
  $sc_no_frames_button
  <INPUT TYPE = "submit" NAME = "order_form_button"
        VALUE = "Checkout Stand">
  </FORM>
  </CENTER>
  </BODY>
  </HTML>!;
}
```

Defining the Cart Footer

The `cart_footer` subroutine is used to generate the HTML footer code for the View Items in the Cart form page. It takes no arguments and is called with the following syntax:

```
&cart_footer;
```

As before, we will display the “Return to Frontpage” button only if we are not using frames by defining it with the `$sc_no_frames_button` in `web_store.setup` .

```
sub cart_footer
{
  print qq!
```

```

<INPUT TYPE = "submit"
        NAME = "change_quantity_button"
        VALUE = "Change Quantity">
<INPUT TYPE = "submit" NAME = "delete_item_button"
        VALUE = "Delete Items">
<INPUT TYPE = "submit"
        NAME = "continue_shopping_button"
        VALUE = "Continue Shopping">
$sc_no_frames_button
<INPUT TYPE = "submit" NAME = "order_form_button"
        VALUE = "Checkout Stand">
</FORM>
</CENTER>
</BODY>
</HTML!>;
}

```

Defining the Bad Order Note

The `bad_order_note` subroutine generates an error message for the customer in the case that they have not submitted a valid number for a quantity. It takes no arguments and is called with the following syntax:

```
&bad_order_note;
```

The code is examined below:

```

sub bad_order_note
{
  &standard_page_header("Wooopsy");
  print qq!

  <CENTER><H2>Wooopsy</H2></CENTER>

  <BLOCKQUOTE>
  I'm sorry, it appears that you did not enter a valid
  numeric quantity (whole numbers greater than zero)
  for one or more of the items you ordered and I am not
  allowed to modify your cart unless you do so. Would
  you try again? Thanks<P>

```

```
<CENTER>
<INPUT TYPE = "submit" NAME = "try_again"
      VALUE = "Try Again">
</CENTER>

</BLOCKQUOTE>
</BODY>
</HTML>!;
exit;
}
```

Defining the Cart Table Header

The `cart_table_header` subroutine is used to generate the header HTML for views of the cart. It takes one argument, the type of view we are requesting, and is called with the following syntax:

```
&cart_table_header([TYPE OF REQUEST]);
```

First the subroutine takes `$modify_type` and makes it into a table header if it has a value. If it does not have a value, then we don't want to output a needless column. There are really only four values that modify type should be equal to:

1. "" (View/Modify Cart or Order Form Screen)
2. "New Quantity" (Change Quantity Form)
3. "Delete Item" (Delete Item Form)
4. "Process Order" (Order Form Process Confirmation)

These four types distinguish the different types of pages on which a cart will be displayed. Note that although there are more than four reasons to display the cart, the header itself is only added onto for three of the reasons. For most of the reasons such as being at the order form screen or simply viewing the cart, the header itself will stay the same without any additions. We need to know the "modify type" value in order to determine if there will be an extra table header in the cart display. In the case of quantity changes or delete item forms, there must be an extra table cell for the checkbox and text field inputs so that the customer can select items. In the View/Modify cart screen (`$modify_type ne ""`), no extra cell is necessary.

```

sub cart_table_header
{
    local ($modify_type) = @_;

    if ($modify_type ne "")
    {
        $modify_type = "<TH>$modify_type</TH>";
    }
    print qq!
<CENTER>
<TABLE BORDER = "1">
<TR>
$modify_type!;

```

@sc_cart_display_fields is the list of all of the table headers to be displayed in the cart display table and is defined in **web_store.setup** .

```

foreach $field (@sc_cart_display_fields)
{
    print qq!<TH>$field</TH>\n!;
}

```

The script also adds on table headers for **Quantity** and **Subtotal**.

```

print qq!<TH>Quantity</TH>\n
        <TH>Subtotal</TH>\n</TR>\n!;
}

```

Defining the Display Cart Table

The job of **display_cart_table** is to display the current contents of the customer's cart for several different types of screens, all of which display the cart in some form or another. The subroutine takes one argument, which is the reason that the cart is being displayed. It is called with the following syntax:

```
&display_cart_table("reason");
```

There are really only five values that **\$reason_to_display_cart** should be equal to:

1. “” (View/Modify Cart Screen)
2. “change quantity” (Change Quantity Form)
3. “delete” (Delete Item Form)
4. “order form” (Order Form)
5. “process order” (Order Form Process Confirmation)

Notice that this corresponds closely to the list in **cart_table_header** because the goal of this subroutine is to fill in the actual cells of the table created by **cart_table_header**. In this list, however, there is one more explicit reason listed for displaying the actual cart table. This is because the cart table display is a little more complex than the cart header display routine. There are more types of changes made to the way the cart table is displayed. These various reasons and how they affect the code will be discussed below.

```
sub display_cart_table
{
```

Working variables are initialized and defined as local to this subroutine.

```
    local($reason_to_display_cart) = @_;
    local(@cart_fields);
    local($cart_id_number);
    local($quantity);
    local($unformatted_subtotal);
    local($subtotal);
    local($unformatted_grand_total);
    local($grand_total);
    local($price);
    local($text_of_cart);
    local($total_quantity) = 0;
    local($total_measured_quantity) = 0;
    local($display_index);
    local($counter);
    local($hidden_field_name);
    local($hidden_field_value);
    local($display_counter);
    local($product_id, @db_row);
```

Next, the script determines which type of cart display it is being asked to produce. It uses pattern-matching to look for key phrases in the **\$reason_to_display_cart**

defined as an incoming argument. Whatever the case, the subroutine calls **cart_table_header** to begin outputting the HTML cart display.

```

if ($reason_to_display_cart =~ /change*quantity/i)
{
    &cart_table_header("New Quantity");
}

elsif ($reason_to_display_cart =~ /delete/i)
{
    &cart_table_header("Delete Item");
}

else
{
    &cart_table_header("");
}

```

Next, the customer's cart is read line by line. File open errors are handled by **file_open_error** as usual.

```

open (CART, "$sc_cart_path" ||
&file_open_error("$sc_cart_path",
    "display_cart_contents", __FILE__,
    __LINE__);
while (<CART>)
{

```

Since every line in the cart will be displayed as a cell in an HTML table, we begin by outputting an opening `<TR>` tag.

```
print "<TR>";
```

Next, the current line has its final newline character chopped off.

```
chop;
```

Then, the script splits the row in the customer's cart and grabs the unique product i.d. number, the unique cart i.d. number, and the quantity. We will use those values while processing the cart.

```

@cart_fields = split (/\/, $_);
$cart_row_number = pop(@cart_fields);
push (@cart_fields, $cart_row_number);

```

```
$quantity = $cart_fields[0];
$product_id = $cart_fields[1];
```

Next, we will need to begin to distinguish between types of displays we are being asked for, because each type of display is slightly different. For example, if we are being asked to display a cart for the Delete Item form, we will need to add a checkbox before each item so that the customer can select which items to delete. If, on the other hand, we are being asked for Modify the Quantity of an Item form, we need to add a text field instead, so that the customer can enter a new quantity.

The first case we will handle is if we are being asked to display the cart as part of order processing.

```
if (($reason_to_display_cart =~ /process.*order/i) &&
    ($sc_order_check_db =~ /yes/i))
{
```

If we are displaying the cart for order processing *and* we are checking the database to make sure that the product being ordered is OK, then we need to load the database libraries if they have not required already.

```
if (!(($sc_db_lib_was_loaded =~ /yes/i))
{
    &require_supporting_libraries (__FILE__, __LINE__,
                                   "$sc_db_lib_path");
}
```

Then, we call the **check_db_with_product_id** in the database library. If it returns “false,” then we output a footer complaining about the problem and exit the program.

```
if (!(&check_db_with_product_id($product_id,*db_row)))
{
    print qq~
    </TR></TABLE>
    Product ID: $product_id not found in database. Your
    order will NOT be processed without this validation!
    </BODY>
    </HTML>~;
    exit;
}
```

Otherwise, we check the returned row with the price of the product in the cart. If the prices do not match, then another complaint message is printed and we exit the program.

```
else
{
  if ($db_row[$sc_db_index_of_price] ne
      $cart_fields[$sc_cart_index_of_price])
  {
    print qq~
    </TR></TABLE>
    Price for product id:$product_id did not match
    database! Your order will NOT be processed without
    this validation!
    </BODY>
    </HTML>~;
    exit;
  }
} # End of Else
} # End of if (($reason_to_display...
```

Remember, we need to use the **display_table_cart** to keep track of totals such as quantity, subtotal, and total measured quantity. Directly below, we keep track of total quantity.

```
$total_quantity += $quantity;
```

Now, we need to fill in the table row for every cart database row. **@sc_display_numbers** defined in the database-specific Setup file will give us the array numbers associated with the fields that we want displayed on this table. Then we will get the value of number from the **cart_fields** array. Hidden fields are generated with the items in the cart if we are displaying the order form and wish the form to be submitted to a different CGI script that merely takes in all form values indiscriminately. The use of hidden fields to store cart information is discussed in Chapter 9.

```
if (($reason_to_display_cart =~ /order*form/i) &&
    ($sc_order_with_hidden_fields =~ /yes/i))
{
  $counter++;
  $hidden_field_name = "cart-"
    . substr("000", length($counter))
```

```

    . $counter;
    $hidden_field_value = join("\|", @cart_fields);
    $hidden_field_value =~ s/\\"/~qq~/g;
    $hidden_field_value =~ s/\>/~gt~/g;
    $hidden_field_value =~ s/\</~lt~/g;

    print qq!
    <INPUT TYPE = "HIDDEN"
        NAME = "$hidden_field_name"
        VALUE = "$hidden_field_value">
    !;
}

```

In the case of a Quantity Change form, we will need to create a cell for the text field into which the customer can input a new quantity. The NAME value is set equal to the unique cart i.d. number of the current item so that when we submit this information, the items will be associated with the new quantities.

```

if ($reason_to_display_cart =~ /change*quantity/i)
{
    print qq!
    <TD ALIGN = "center">
    <INPUT TYPE = "text" NAME = "$cart_row_number"
        SIZE = "3"></TD>!;
}

```

Similarly, in the case of a Delete Item form, we must include a cell with a checkbox so that the customer can select items to delete from their cart. The NAME value is set equal to the unique cart i.d. number of the current item so that when we submit this information, the items will be associated with the checked checkboxes.

```

elsif ($reason_to_display_cart =~ /delete/i)
{
    print qq!
    <TD ALIGN = "center">
    <INPUT TYPE = "checkbox" NAME = "$cart_row_number">!;
}

```

\$display_counter is set equal to zero. This variable will be used to keep track of the number of displayed fields.

\$text_of_cart is initialized with two newlines. This variable will be used to

hold the entire formatted cart contents in one string so that we will be able to send a neatly formatted copy of the cart as plain ASCII to a log file or as email to the admin. We will be using the `.=` operator to append to the variable rather than overwrite it.

```
$display_counter = 0;
$text_of_cart .= "\n\n";
```

Now, for every item in the cart row that should be displayed as defined in the Setup file, we will do two things. First, we will append the data to the **\$text_of_cart** variable (formatting it neatly). Then we will display the data as a table cell. However, there are three types of data that must be displayed in table cells but each must be formatted slightly differently. The first type of cell is a cell with no data. To give the table a three-dimensional look, we will substitute all occurrences of no data for the ** ** character in order to get a blank but indented table cell. Of course, this routine simply overwrites the empty value of the data with the ** ** character. It does not actually display the cell; it passes that job on to the next “if” test.

Another case is when a table cell must reflect a price. In that case we must format the data with the monetary symbol defined in **web_store.setup**, using **display_price** as discussed in **web_store.cgi**. Finally, nonprice table cells are displayed (including those passed down from the first case).

```
foreach $display_index (@sc_cart_index_for_display)
{

Reformat blank cells.
if ($cart_fields[$display_index] eq "")
{
```

The text of the cart is entered into a buffer. The actual item being purchased is formatted inside a 25-character-width field.

```
$text_of_cart .= $sc_cart_display_fields[$display_counter] .
    substr(" " x 25,
length($sc_cart_display_fields[$display_counter])) .
    "= nothing entered\n";
$cart_fields[$display_index] = "&nbsp;";
}
```

Then the script displays the price or price after options cells.

```
if (($display_index ==
    $sc_cart_index_of_price_after_options)||
    ($display_index ==
    $sc_cart_index_of_price))
{
    $price = &display_price($cart_fields[$display_index]);
    print qq!<TD ALIGN = "center">$price</TD>\n!;
    $text_of_cart .= $sc_cart_display_fields[$display_counter] .
        substr((" " x 25),
            length($sc_cart_display_fields[$display_counter])) .
            "= $price\n";
}
}
```

Next, the script displays all other cells (blank cells have already been reformatted).

```
else
{
    print qq!<TD ALIGN =
        "center">$cart_fields[$display_index]</TD>\n!;
    $text_of_cart .= $sc_cart_display_fields[$display_counter] .
        substr((" " x 25),
            length($sc_cart_display_fields[$display_counter])) .
            "= $cart_fields[$display_index]\n";
}
}
```

If the current display index happens to be a cell that must be measured, we will add the value to **\$total_measured_quantity** for later calculation and display.

```
if ($display_index == $sc_cart_index_of_measured_value)
{
    $total_measured_quantity += $cart_fields[$display_index];
}

$display_counter++;
} # End of foreach $display_index...
```

Then, we must use the quantity value we earlier grabbed to fill the next table cell, and, after using another database specific setup variable, calculate the subtotal for that database row and the final cell and close out the table row and the cart file once we have gone all the way through it.

```

$unformatted_subtotal =
    ($quantity*$cart_fields[$sc_cart_index_of_price_after_options]);
$subtotal = &format_price($unformatted_subtotal);
$unformatted_grand_total = $grand_total + $subtotal;
$grand_total = &format_price($unformatted_grand_total);

$price = &display_price($subtotal);
print qq!<TD ALIGN = "center">$quantity</TD>
      <TD ALIGN = "center">$price</TD>
      </TR>!;

$text_of_cart .= "Quantity" .
    substr(" " x 25),
    length("Quantity")) .
    "= $quantity\n";
$text_of_cart .= "Subtotal For Item" .
    substr(" " x 25),
    length("Subtotal For Item")) .
    "= $price\n";

} # End of while (<CART>)
close (CART);

```

Finally, print out the footer with the `cart_table_footer` subroutine in `web_store.html`.

```

$price = &display_price($grand_total);
&cart_table_footer($price);

```

In the case of an order form, we will also have to create a hidden input tag with which to transfer the subtotal state information to the order processing routines.

This is necessary only if the order is being submitted to another server that does not have access to the cart file for processing against.

```

if (($reason_to_display_cart =~ /order*form/i) &&
    ($sc_order_with_hidden_fields =~ /yes/i))
{
    $hidden_field_name = "subtotal";
    $hidden_field_value = $subtotal;
    print qq!
    <INPUT TYPE = "HIDDEN" NAME = "$hidden_field_name"
            VALUE = "$hidden_field_value">!;
}

```

The **Subtotal** information is also added to `$text_of_cart`:

```
$text_of_cart .= "\n\nSubtotal:" . substr(" " x 25),  
length("Subtotal:")) . "= $price\n\n";
```

We need to return the subtotal for routines such as ordering calculations. We also need to return the text of the cart in case we are logging orders to email or to a file:

```
return($grand_total,  
        $total_quantity,  
        $total_measured_quantity,  
        $text_of_cart);  
} # End of display_cart_table
```

Defining the Cart Table Footer

The `cart_table_footer` subroutine is used to display the footer for cart table displays. It takes one argument, the preshipping grand total and is called with the following syntax:

```
&cart_table_footer(PRICE);
```

The code is examined below:

```
sub cart_table_footer  
{  
  local($price) = @_;  
  print qq!  
</TABLE>  
<P>  
  Pre-shipping Grand Total = $price  
<P>!;  
}
```

Automatically Generating Hidden Fields

The `make_hidden_fields` subroutine is used to generate the hidden fields necessary for maintaining state. It takes no arguments and is called with the following syntax:

```
&make_hidden_fields;  
The code is explained below:
```

```

sub make_hidden_fields
{
    local($hidden);
    local($db_query_row);
    local($db_form_field);

```

\$hidden is defined initially as containing the **cart_id** and page **hidden** tags that are necessary state variables on *every* page in the cart. The script then goes through checking to see which optional state variables it has received as incoming form data. For each of those, the script adds a **hidden** input tag.

```

$hidden = qq!
<INPUT TYPE = "hidden" NAME = "cart_id"
        VALUE = "$cart_id">
<INPUT TYPE = "hidden" NAME = "page"
        VALUE = "$form_data{'page'}">!;
if ($form_data{'keywords'} ne "")
{
    $hidden .= qq!
    <INPUT TYPE = "hidden" NAME = "keywords"
            VALUE = "$form_data{'keywords'}">!;
}

if ($form_data{'exact_match'} ne "")
{
    $hidden .= qq!
    <INPUT TYPE = "hidden" NAME = "exact_match"
            VALUE = "$form_data{'exact_match'}">!;
}
if ($form_data{'case_sensitive'} ne "")
{
    $hidden .= qq!
    <INPUT TYPE = "hidden" NAME = "case_sensitive"
            VALUE = "$form_data{'case_sensitive'}">!;
}

foreach $db_query_row (@sc_db_query_criteria)
{
    $db_form_field = (split(/\|/, $db_query_row))[0];
    if ($form_data{$db_form_field} ne "" &&
        $db_form_field ne "keywords")
    {
        $hidden .= qq!

```

```
        <INPUT TYPE = "hidden" NAME = "$db_form_field"
            VALUE = "$form_data{$db_form_field}">!;
    }
}
return ($hidden);
} # End of make_hidden_fields
```

Displaying Results for No Hits Found in HTML Search

The `PrintNoHitsBodyHTML` subroutine is utilized by the HTML-based store search routines to produce an error message in case no *hits* (successful matches) were found, based on the customer-defined keywords. It is called with no arguments and the following syntax:

```
&PrintNoHitsBodyHTML;
```

The code is shown below:

```
sub PrintNoHitsBodyHTML
{
    print qq!
    <P>
    <CENTER>
    <H2>Sorry, No Pages Were Found With Your
        Keyword(s).</H2>
    </CENTER>
    <P>!;
}
```

Defining the Body in the HTML Search

The `PrintBodyHTML` subroutine is utilized by the HTML-based store search routines to produce a list of hits. These hits will be those pages that had the customer-defined keywords within them. The subroutine takes two

arguments, the filename as it will appear in the URL link as well as the text that should be visibly hyperlinked and is called with the following syntax:

```
&PrintBodyHTML("file.name", "Title to be linked");
```

The code is shown below:

```
sub PrintBodyHTML
{
    local($filename, $title) = @_;
    print qq!
    <LI><B>
    <A HREF =
"$sc_main_script_url?page=$filename&cart_id=$cart_id">$title</A>
    </B>(/$filename)!;
}
1;
```