CHAPTER THREE

FOCUS ON THE HTML STORE

The HTML Store, which uses pre-designed HTML pages to display products for sale, is one of the most commonly used configurations of the Web store. In fact, over the last year, dozens of companies have chosen to use the HTML interface and have customized a wide array of storefronts. Many of these companies have registered their URL with us and are listed at Selena Sol's Script Archive at the following URL:

```
http://www.eff.org/~erict/Scripts/
```

We recommend that you browse through this list to get ideas as to how you might create your own "storefront" look.

The HTML store is popular because of its versatility and ease of use. The HTML store uses pre-designed HTML pages to create the store environment and as such, makes customizing a wide array of unique pages easy and less intimidating. Figure 3.1 shows a product page: **Vowels.html**.

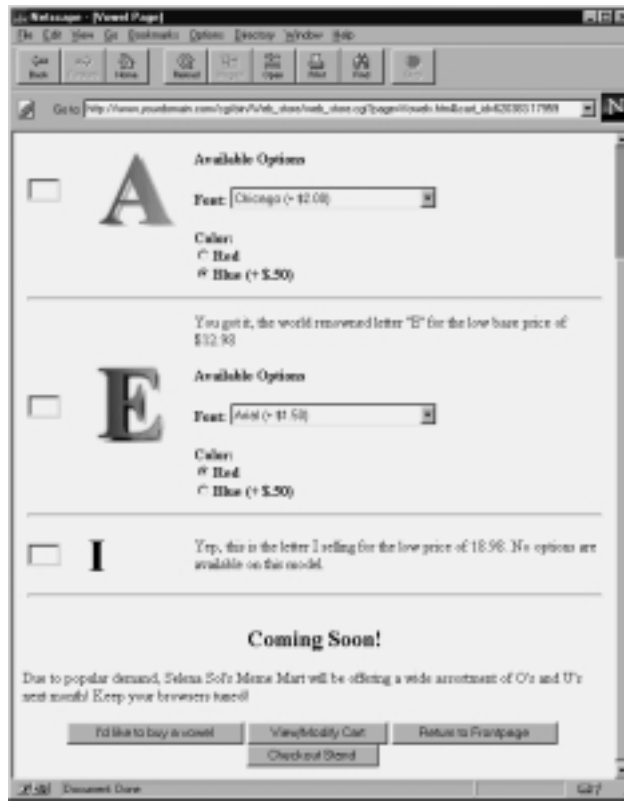

Figure 3.1 Vowels.html.

In fact, its two most cited benefits are the ease with which a CGI novice can customize the store without ever reading the code and the fact that creating the store with HTML pages gives the store designers an infinite amount of flexibility with the presentation of products. After all, the lack of a database back end, and the dependence upon HTML as the medium of programming, make the HTML Store just like doing straight HTML. The little bit of customization that is necessary can be summed up in a few simple tags that, when embedded in your HTML, will create the necessary link between your customers and the application. This chapter will discuss these tags to give you a firm basis for creating your own tagged HTML product pages.

Changes to the Setup File

As with all versions of this script, the first thing you must do is to customize the Setup file. For the most part, the setup file for the HTML store will be like any other implementation. Eighty percent of the setup should remain constant through all versions. However, there will be some differences. In the following section, we will go through **web_store.setup.html** highlighting variables specific to the HTML store and explain their usage. Variables not mentioned should be considered globally important to all versions of the store and will be defined the same way regardless of which implementation you choose. Chapter 2 covers all these general variables.

Global File Location Variables

The first variable of interest in **web_store.setup.html** is **$sc_db_lib_path**. **$sc_db_lib_path** defines the location of **web_store_db_lib.pl** used to perform searches on the data file. Because we will be using HTML product pages, there will be no need to search the data file and no need for that variable. Searching is done with **web_store_html_search.pl**, which will search every HTML product page for customer-submitted keywords and create a dynamic list of hits. However, just because we will not be searching the data file does not mean that the data file is useless for the HTML store. In fact, many HTML stores will utilize the database back end to verify orders received from the product pages as discussed in Chapter 8. If you are using this double-check method, you will still want to define **$sc_data_file_path**. If you do not want to double-check orders against a datafile, you may disregard that variable completely.

Similarly, there is no neeed to be concerned with **$sc_options_directory_path**. This variable defines the locations of the options files used by the Database store to create option-related HTML. Since you will be including all option HTML in the actual product pages themselves, you will not have any need for separate Option files, the Options subdirectory, or the **$sc_options_directory_path** variable.

$sc_html_product_directory_path, however, is crucial. As mentioned earlier, this variable defines the location of the directory in which you put your product pages as well as your list of product pages.


NOTE

A *list of products* page is a page that contains hyperlinks to pages containing products called *product pages*. These are mainly used for store navigation and the categorization of similar types of products.

The application must know the location of this directory if it is to filter requested product pages.

When a customer clicks on a button or hyperlink for a product, the script should receive a **$page** variable as form data. This **$page** variable will correspond to a specific HTML document that the script should filter and display in the browser window.

For example, in **Letters.html**, which is our example of a list of products page located in the distribution Products subdirectory, you will find a link to the Vowels product page with the following syntax:

```
<A HREF =
"web_store.cgi?page=Vowels.html&cart_id=">Vowels</A>
```

Figure 3.2 shows **Letters.html** as it appears on the Web.

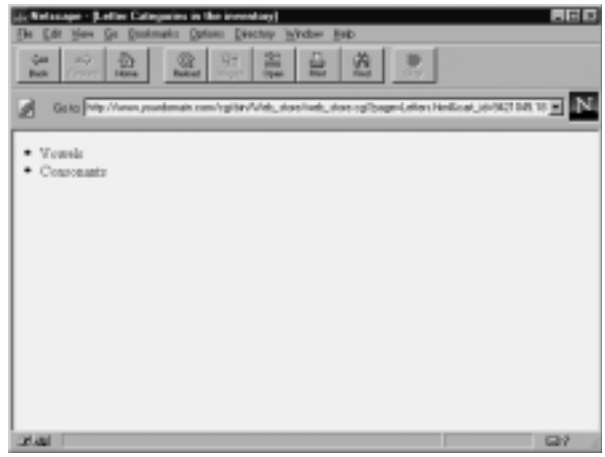

Figure 3.2 Letters.html.

Notice that in the above URL, we have passed the variable **page** with the value of **Vowels.html** to the script. We will discuss the **cart_id=** tag later. However, in order to actually find the page **Vowels.html**, the script must know the location of the file on your server. **$sc_html_product_directory_path** provides it with that information. Thus, if you set this variable equal to **/usr/local/httpd/cgi-bin/Web_store/Html/Products/**, the script will expect to find the file **Vowels.html** using the following location **/usr/local/httpd/cgi-bin/Web_store/Html/Products/Vowels.html**.

You may also define subdirectories in your hyperlink calls if you decide to partition your HTML product pages. Thus, the following hyperlink would access **e.html** in the **/usr/local/httpd/cgi-bin/Web_store/Html/Products/Letters/Vowels/Lower_case/** subdirectory:

```
<A HREF = "web_store.cgi?page=Letters/Vowels/Lower_case/
e.html&cart_id= ">Vowels</A>
```

Understanding the Tags for Maintaining State

Now that we have started discussing the concept of a list of products page and introduced the strange **cart_id=** tag, we should take a moment to explain state maintenance tags in more detail. The trick about list of product pages like **Letters.html** as well as product pages like **Vowels.html** for that matter, is that they are tagged so that the script can automatically insert state information when it filters them. pAs we said in the previous chapter, all HTML pages must be filtered through **web_store.cgi** if the customer's state information is to be maintained. For list of product pages, there are two bits of state information that must be passed: (1) the product page from the list that the customer wants to see, and (2) the location of the customer's unique shopping cart.

In fact, every internal-store hyperlink (most commonly used in list of product pages) must communicate that information. The problem is that when you write the list of product pages in HTML, there is no way for you to know in advance what the cart i.d. will be. In fact, many different customers, all with different cart ids will be navigating through the same page. There is no way to hardcode that information into your HTML. But you can provide a tag which the script will recognize and dynamically exchange for the current value it has for the customer's cart.

This task is handled in the **display_page** subroutine in **web_store.cgi**. The three magic lines are as follows:

```
s/cart_id=/cart_id=$cart_id/g;
s/%%cart_id%%/$cart_id/g;
s/%%page%%/$form_data{'page'}/g;
```

These three lines assure that every HTML page passed to **web_store.cgi** will be filtered for these two state variables. Let's consider the line in **Letters.html**.

```
<A HREF = "web_store.cgi?page=Vowels.html&cart_id=">Vowels</A>
```

In this line, we see that we left the URL encoded hyperlink undone. Specifically, we did not set a value for **cart_id**. We simply left it dangling. However, the script is prepared for that and is actually looking for the **cart_id** flag.

As the script filters the file, preparing it to send to the Web browser, it checks to see if the flag exists. If it does, the script substitutes the incomplete phrase **cart_id=** with **cart_id=[THE ACTUAL CART VALUE]**.

So what are the other two lines with the %% flags? Well these are used to filter hidden form tags instead of hyperlinks. After all, there are two ways to submit information to a script: by URL encoded hyperlink or by hitting a Submit button on a form. If a submit button is used, state information must be passed as hidden form fields. This is most common for product pages on which customers can hit the submit button to purchase an item. Thus, we need a tag which the script will use to filter hidden <INPUT> tag fields. A quick glance at the hidden fields in **Vowels.html** provides an example of how you should prepare the flag:

```
<INPUT TYPE = "hidden" NAME = "cart_id"
VALUE = "%%cart_id%%">
<INPUT TYPE = "hidden" NAME = "page"
 VALUE = "%%page%%">
```

These two lines will be "filtered" by **web_store.cgi** which will insert the actual value of **cart_id** and page.

In summary, it is essential that all product pages have the %% hidden form tag flags and that all hyperlinks have the **cart_id=** flags.

Bypassing Database Definition Variables by Encoding your Product Pages

Because we are not using a database, the entire group of database definition variables in the Setup file becomes unnecessary unless you are using the database check functions. This is not to say that we will not need to define a specific format for our products. In fact, you will need to follow a very specific format for defining the fields that make up this information in your HTML product pages. **Vowels.html** is an excellent example of a product page.

Each product page begins with a standard header. In the **Vowels.html** example, we use a simple borderless table to position our products. Of course, you can change this to reflect the specifics of your own site—that is the beauty of the HTML store.

Be absolutely sure that the form call in every page points to **web_store.cgi**. Every page that the customer sees must be processed by **web_store.cgi**

```
<HTML>
<HEAD>
<TITLE>Vowel Page</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<FORM METHOD = "post" ACTION = "html_web_store.cgi">
<H2><CENTER>Vowel Mart!</CENTER></H2>
<TABLE>

<TR>
<TH>Quantity</TH>
<TH></TH>
<TH>Description</TH>
</TR>

<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>
<TR>
```

The syntax defining a product appears next.

```
<TR>
<TD ALIGN = "center"><INPUT TYPE = "text"
NAME = "item-0010|Vowels|15.98|The letter A|~lt~IMG SRC =
~qq~/~erict/Scripts/Web_store/Html/Images/a.jpg~qq~ ALIGN =
~qq~left~qq~~gt~" SIZE = "3" MAXLENGTH = "4"></TD>

<TD ALIGN = "center"><IMG SRC =
"/~erict/Scripts/Web_store/Html/Images/a.jpg" ALIGN =
"left"></TD>
<TD>You got it,the world renowned letter "A" for the plow base price
of $15.98<BR><P>
```

You can customize this to display in any way you would like so long as you follow a few rules. Specifically, every purchasable item *must* have a text field entry box representing a quantity and the INPUT tag must follow the following basic format:

```
<INPUT TYPE = "text"
       NAME = "item-WWW|XXX|YYY|ZZZ"
       SIZE = "3" MAXLENGTH = "4">
```

where **WWW** is a **UNIQUE** product identification (like an ISBN number) and **XXX**, **YYY**, and **ZZZ** are bits of information that you want displayed in the customer's cart view. p<NOTE>Each item has a **TEXT** input field associated with it. This text field is three characters long and provides a maximum of four characters by default. Since this corresponds to the quantity that the customer might want to order, it is set to allow only orders less than 9999 quantity. You can change this if you want. pThe definition of this **NAME** value will correspond exactly to the logic of **@sc_db_index_for_defining_item_id**. In fact, that variable is used to dynamically generate the same code for the database store that you must personally do on HTML pages.

Defining an Item

The process of writing your own **NAME** value is not difficult, but it must be painstakingly accurate. First, and most importantly, notice that the **NAME** argument of the **INPUT** field is item-0010|Vowels|15.98|The letter A|~lt~IMG SRC=~qq~Html/Images/a.jpg~qq~ ALIGN=~qq~left~qq~~gt~".

This is where you will define what information gets added to the customer's shopping cart. There are three points to note about the **NAME** string. pFirst, the **NAME** string corresponds to the **%cart** just as **@sc_db_index_for_defining_item_id** did in the Setup file discussed in Chapter 2. You are simply hardcoding that array for each item.

Second, notice that the tag **item-** has been appended to the product i.d. We do this in order to differentiate items from other incoming form data when the customer submits an order. The **item-** tag will be stripped off by the script when it adds the row to the cart, but it *must* be there in your HTML code so that the script will know that this field represents an item.

Finally, notice the **<IMG SRC>** reference. It has several special tags which we use to represent characters that may cause trouble within your HTML code (specifically, characters used in the HTML code standard itself). There are three in particular. **~qq~** represents a double quote mark (“), **~gt~** represents a greater than symbol (>) and **~lt~** denotes a less than symbol (<). The script knows how to translate these when it uses them to display the customer's cart but you must encode them here so that they will not confuse the **<INPUT>** tag in which they are embedded.

Cart Definition Variables

Now that we have gone through all the trouble of hardcoding our product information into all our HTML pages, it is time to make it pay off. The Cart Definition variables will be used to index the product information coming in to the script in the form of text field **NAME** form data. Let's go through the cart definition for the Vowels page discussed previously.

Given the definition of the letter *A* for sale on the **Vowels.html** page, we can tell that the format of **%cart** must be

```
$cart{"quantity"}                    = 0;
$cart{"product_id"}          = 1;
$cart{"category"}             = 2;
$cart{"price"}                       = 3;
$cart{"description"}         = 4;
$cart{"image_location"}              = 5;
$cart{"options"}                     = 6;
$cart{"price_after_options"}         = 7;
```

As was discussed in Chapter 2, fields 1 through 5 are the same as defined in the **NAME** argument of the quantity text box which defines what gets passed to the cart for each item.

All other variables will depend upon utilizing these array elements.

Order Form Definition Variables

These variables will basically be the same for all versions of the Web store regardless of whether they take their information from product pages or a data file because they will all have the same order-processing interface. In terms of the HTML store though, there will be one important variable, **$sc_order_check_db**. If you set this variable to **yes**, the script will use the database routines to double-check that the customer has not attempted to fool around with the cart values by entering fake values on their own custom designed HTML pages or by creating their own URL encoded string. For example, what do you think would happen if some customer typed in the following URL string?

```
http://www.yourdomain.com/cgi-
bin/Web_store/web_store.cgi?page=Words.html&add_to_cart_
button=IHACKEDYOU&cart_id=8427734.9651&item-
1000|Words|14.98|Extropy=5
```

Well, to put it bluntly, they would have just received a discount. If you check the Meme product page, you will see that the Extropy Meme is 15.98, not 14.98. Unfortunately, if you are not careful, you may miss such a sneak attack and the hacker could get away with a discount. Thus, it is advisable for high-traffic stores, for you to create a database backend for your HTML product pages so that all values can be checked against a database that is safely on your server. When the customer attempts to order, if there are any discrepancies, the script will alert you and the customer. As a side note, this could also help in case you make typos in your product pages because the script will recognize accidental as well as intentional discrepancies.

This is not meant to frighten you away from using the HTML store. The versatility of the HTML interface gives you quite a bit of power with creative sites that the Database store cannot give. It is simply a warning for you to take care when processing orders and to suggest that you implement the database checking routines as soon as possible. The extra work will pay off in terms of a good night's sleep.

Store Option Variables

Of course the most basic variable definition for the HTML store is **$sc_use_html_product_pages**. The only way to use product pages is to set this variable to **yes**. If it is set to **no**, the script will look for a data file from which to dynamically generate product pages.

HTML Search Variables

One of the powerful tools of the HTML Web store is the built-in search engine. The search engine takes a list of whitespace separated keywords and searches every product page for pages that contain all of those words. The keywords are entered in a standard FORM text box defined with the following code:

```
<INPUT TYPE = "text" NAME = "keywords">
```

Unlike the Database-based search engine included with this script that can search by dates, keyword, or number, the HTML-based search can only search by keyword, since the nonstandardized HTML format does not allow the more complicated searching.

Figure 3.3 shows the HTML frontpage with the search input box included on the bottom.

Figure 3.3 Frontpage with Keyword Search box.

The customer can enter as many keywords as he or she wishes and the script will generate a list of pages that satisfied all the keywords entered.

The HTML keyword search uses the "and" methodology so that every keyword must be found on a page for that page to be displayed as a hit.

Figure 3.4 shows a dynamically generated "list of hits" page that includes hyperlinks to every page that satisfied the customer's search criteria.

Figure 3.4 List of Hits page.

$sc_root_web_path is the location of the root directory containing all the HTML product pages. In the distribution, this is "./Html/Products/".

@sc_unwanted_files is the list of all the file types which you do not want to be searched when submitting a search term for keyword match. This subject will be discussed more thoroughly in Chapter 13.

Error Message Variables and Miscellaneous Variables

These variables are global for all versions of the Web store and have already been discussed in sufficient detail in Chapter 2.

Understanding Options

Before you begin coding your product pages, there is one last thing to discuss: options. *Options* are unique characteristics that can be applied to a generic product. For example, a basic T-shirt might come in black, blue, or green or small, medium or large. We need a way to communicate to the script what an option is, which item it belongs to, what effect it will have on the base price of that item, and the value set by the customer.

The first part in this process is to make sure that we associate options with items for sale. We do this by using the NAME argument of the form tag which defines each option. Below is an example of using a select menu for options.

```
<P><B>Available Options<B><P>
Font: <SELECT NAME = "option|1|0001">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No
charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>
<P>
```

In this case, the name syntax breaks down as follows:

1. *Option flag* This flag tells the script that the incoming data is an option, not an item Thus the first field in a pipe-delimited option **NAME** value will *always* be **option**, just as item **NAME** tags **ALWAYS** begin with **item-**
2. *Unique sequence number of the option* Each item for sale may have several options associated with it. It is essential that they each gets its own number. If item #0001 had all its options called "option|0001," it would be impossible to parse them separately. So we will name them uniquely, as in "option|1|0001" for color, "option|2|000" for size, or "option|3|0001" for brand name.
3. *The i.d. of the item that the option is associated with* Notice that this i.d. is the same as what was used in the **NAME** argument for the quantity text box in a product field box. This is deliberate and essential. Options must be associated with the items they modify. Figure 3.5 shows how options might be integrated into a product page.

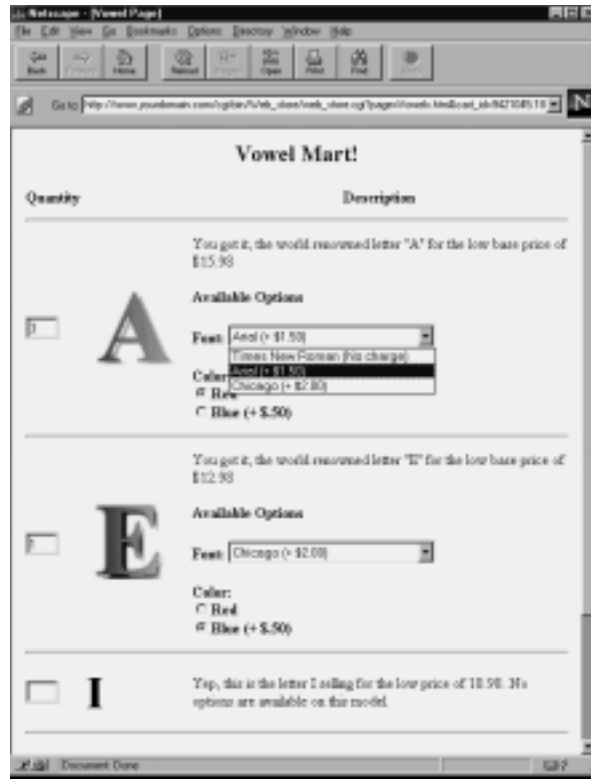

Figure 3.5 Using options.

Finally, notice that options also contain VALUES which are two field pipe delimited lists containing an option description and an option price. The option description will be used for display in the customer's cart and the price will be used to modify the base price of an item.

The following is an example of using a radio button to create an option. It uses the same naming conventions as the **<SELECT>** tag but included here for variety.

```
Color: <BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"p       VALUE = "Red|0.00"
CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"
       VALUE = "Blue|.50">Blue (+ .50)
```

In other words, product number 0001 has two possible options. Font type is option 1 and color is option 2. The customer may choose from three font types. If she chooses "Arial," $1.50 will be added to the base price of the item. She can also order "red" or "blue." If she chooses "red," nothing will be added to the base price of the item.

If this option modified the "letter A" discussed above, and a customer ordered three of them, all red with Arial font, the cart row would appear as follows:

```
3|0010|Vowels|15.98|The letter A|<IMG SRC =
"/~erict/Scripts/Web_store/Html/Images/a.jpg" ALIGN =p"left">|Arial
1.50, Red 0.00|17.48|358
```

Simple Item Definition

Options are not mandatory, of course. In fact, we have also provided an example of an item with no options in case you don't care about options and just want an example without them. This is obviously much simpler since all you need to worry about is the one NAME argument of each item for sale. In **Vowels.html** we have prepared the letter *I* as an example without options.

Finally, the entire code of both **Vowels.html** and **Letters.html** are included for a complete reference:

Vowels.html

```
<HTML>
<HEAD>
<TITLE>Vowels Page</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<FORM METHOD = "post" ACTION = "web_store.cgi">
<INPUT TYPE = "hidden" NAME = "cart_id"p       VALUE = "%%cart_id%%">
<INPUT TYPE = "hidden" NAME = "page"p       VALUE = "%%page%%">
```

```
<TABLE BORDER = "0">
<TR>
<TH>Quantity</TH>
<TH></TH>
<TH>Description</TH>
</TR>

<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>
<TR>p<TD ALIGN = "center"><INPUT TYPE = "text"
NAME = "item-0010|Vowels|15.98|The letter A|~lt~IMG SRC =
~qq~/~erict/Scripts/Web_store/Html/Images/a.jpg~qq~ ALIGN =
~qq~left~qq~~gt~"
SIZE = "3" MAXLENGTH = "4"></TD>

<TD ALIGN = "center"><IMG SRC =
"/~erict/Scripts/Web_store/Html/Images/a.jpg" ALIGN =
"left"></TD>

<TD>You got it,the world renowned letter "A" for the low
base price of$15.98<BR><P>
<B>Available Options<B>
<P>
Font:p<SELECT NAME = "option|1|0010">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>

<P>
Color:p<BR>
<INPUT TYPE = "radio" NAME = "option|2|0010"
       VALUE = "Red|0.00" CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0010"
       VALUE = "Blue|.50">Blue (+ $.50)
</TD>

</TR>
</TR>
<TR>p<TD COLSPAN = "3"><HR></TD>
</TR>
<TR>p<TD ALIGN = "center"><INPUT TYPE = "text"
NAME = "item-0011|Vowels|12.98|The letter E|~lt~IMG SRC =
~qq~/~erict/Scripts/Web_store/Html/Images/e.jpg~qq~ ALIGN =
~qq~left~qq~~gt~" SIZE = "3" MAXLENGTH = "4"></TD>
```

```
<TD ALIGN = "center"><IMG SRC =
"/~erict/Scripts/Web_store/Html/Images/e.jpg" ALIGN =
"left"></TD>

<TD>You got it, the world renowned letter "E" for the
low base price of $12.98<BR><P>
<B>Available Options<B>
<P>
Font:p<SELECT NAME = "option|1|0011">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>

<P>
Color:p<BR>
<INPUT TYPE = "radio" NAME = "option|2|0011"
       VALUE = "Red|0.00" CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0011"
       VALUE = "Blue|.50">Blue (+ $.50)
</TD>

</TR>
<TR>p<TD COLSPAN = "3"><HR></TD>
</TR>
<TR>p<TD ALIGN = "center"><INPUT TYPE = "text" NAME = "item-
0012|Vowels|18.98|The letter I|~lt~CENTER~gt~~lt~FONT SIZE =
~qq~+4~qq~~gt~I~lt~/FONT~gt~~lt~/CENTER~gt~"
SIZE = "3" MAXLENGTH = "4"></TD>

<TD ALIGN = "center"><CENTER><FONT SIZE =
"+4">I</FONT></CENTER></TD>

<TD>Yep, this is the letter I selling for the low price
 of 18.98. No options are available on this model.
<BR></TD>

</TR>
<TR>p<TD COLSPAN = "3"><HR></TD>
</TR>

</TABLE>
<P>
```

```
<CENTER>
<INPUT TYPE = "submit" NAME = "add_to_cart_button"
       VALUE = "Add Items to my Cart">
<INPUT TYPE = "submit" NAME = "modify_cart_button"p       VALUE =
"View/Modify Cart">
<INPUT TYPE = "submit"p       NAME = "return_to_frontpage_button"
       VALUE = "Return to Frontpage">
<INPUT TYPE = "submit" NAME = "order_form_button"p       VALUE =
"Checkout Stand">
</FORM>
</CENTER>
</BODY>
</HTML>
Letters.html
<HTML>
<HEAD>
<TITLE>Letter Categories in the inventory</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">

<LI><A HREF =
"web_store.cgi?page=Vowels.html&cart_id=">Vowels</A>
<LI><A HREF =
"web_store.cgi?page=Consonants.html&cart_id=">Consonants</A>

</BODY>
</HTML>
```

Summary

To utilize the HTML-based interface, you must satisfy several requirements. First, you must change the following variables in the setup file:

```
$sc_data_file_path
$sc_html_product_directory_path
$sc_order_check_db
$sc_use_html_product_pages
$sc_root_web_path
$sc_unwanted_files
```

Second, any list of product pages that you create for navigation must be hard-coded with the **page** and **cart_id=** flags for filtering.

Finally, you must create product pages with text field boxes with hard-coded **NAME** values according to the data you want transferred to the cart. Options must also be prepared where appropriate.