BAIKONUR
Internet/Intranet Suite
Application Developer's Guide

Introduction

BAIKONUR Intranet Suite includes the Web Application Server, the HTML Controls Library and additional tools. It enables the developer to create application software in Internet/Intranet technology. An Intranet application is an application that utilizes the technologies and software tools that had originally been designed to access data in Internet. The wide proliferation of Internet technologies offers solutions to a number of complicated problems that challenge the developers of almost any application software, be it a rudimentary program or an elaborate corporate information storage/retrieval system. Among the most important advantages that Intranet technology offers, the following clearly stand out:

a) It provides an easy and inexpensive way of gaining access to remote data. Internet is a global network, and therefore you don't have to use a long-distance or leased telephone communication line to access a database in Moscow from, say, London – an outlet to Internet is all you need to do that.

b) It allows the client to use virtually any operating system and any computer - any Web browser on any platform can be used as a client program. No additional software download is required.

c) It makes administration easier. Only one copy of Intranet application running on the Web is required.

d) It permits many users to simultaneously work with data on an SQL server even if the number of its user licenses is limited. An Intranet application can supply data for many users by utilizing only a single connection to a database. A country-wide ticket reservation and booking system is a vivid example of that.

e) It enables the developer to easily and quickly create HTML pages that can represent most up-to-date data, even in real time (for example, a HTML page may contain the latest stock exchange quotations).

Due to the use of Borland Delphi (or C++ Builder, or JBuilder), applications of this kind can be developed in a visual environment and in as short a time span as conventional application software.

To build software systems in Intranet technology, you need the following:

- A Web browser as a client application. There is now a horde of Web browsers on the market for practically any software and hardware platforms (such as Netscape, Mosaic, Lynx, Internet Explorer, Ariadna and so on). More than that, a Web browser can be built into a client application. Delphi , for one, incorporates a library of components allowing you to do just that. Senior versions of BAIKONUR (starting from BAIKONUR Enterprise) include components allowing the developer to create client applications that do not use the Hypertext Markup Language (HTML).

- A Web server that can transmit data from an Intranet application to the client and backward. We suggest that you use BAIKONUR if you want to be able to utilize the Baikonur HTML Controls library most comprehensively. BAIKONUR is a Web server that, apart from supporting typical Web-server functions, can also function as an application server. BAIKONUR can launch the necessary application on the client's request, and then supervise and manage the transmission of data to/from the client.
- An Intranet application software. To facilitate the building of such applications in Delphi or C++ Builder, we have incorporated a Baikonur HTML Controls Library into our BAIKONUR Intranet Suite.

The Intranet application development technology which BAIKONUR offers can be utilized both for developing closed corporate information systems and building an open Internet Web site or expanding its functionality. This technology and the latest Internet developments (Java, ActiveX, etc.) are not mutually exclusive, but rather supplement one another.

Apart from the development of conventional Internet/Intranet applications, BAIKONUR can also be employed to solve non-trivial (from the point of view of Internet) tasks, such as data replication, remote monitoring, organizing a print server, etc.

Chapter I

This chapter presents an overview of the general principles of developing applications for BAIKONUR, and some of issues related to the use of HTML and browsers.

What is BAIKONUR?

BAIKONUR Web Application Server, as its name implies, is a two-faceted software package - it is a usual Web server and an application server all in one.

The Web server is needed to perform the usual functions of sending HTML pages, images and other resources on a client's (browser's) request. In addition to that, the Web server supports CGI and ISAPI interface technologies. You can use BAIKONUR to build a conventional Web site. See "BAIKONUR. Getting Started" for details on how to install, start-up and administer a Web server.

The application server is needed to organize an interface between the user and the application launched on the server upon receipt of a corresponding request. This request is essentially the URL specifying the name of the program (called the Web application) to be launched. The remote client can use any Web browser that supports the HTML protocol (Netscape Navigator, Microsoft's Internet Explorer, etc.). The application server is responsible for launching the appropriate application on the client's request, passing data from the client to the application and backward, and closing the application.

To exchange data between the client and application, use is made of the HTTP protocol. Dialog screen forms are described with the help of the HTML (Hypertext Markup Language), which permits the use of interactive control elements, such as single-line and multiple-line text input/edit fields (Edit, Memo), button controls (Button), view lists and drop-down lists (ListBox, ComboBox), and others. Once it is launched in response to a client's request, the Web application builds an HTML script describing the user dialog, and passes it to the client. The user performs the necessary actions (views and modifies data in the input fields), and sends the modified data (generated by the browser after the user clicks a submit-type button) to the application. The application parses and saves the incoming data, builds a new HTML script and sends it back to the client, and so on. The *HTMLPage* and *HTMLControl* components from Baikonur HTML Controls Library are responsible for such functions as parsing of the incoming data, generation of the HTML script based on the form's current status, and transfer of data between the application and the server. All this is done automatically, so that the application developer doesn't normally have to interfere into this process.

The client-application interface scheme outlined above is very reminiscent of the CGI or ISAPI, but with one important exception. The Web application does not stop functioning after data is sent to the client. Instead, it retains its status in expectation of a next user-generated request. In fact, the user works on a remote terminal and application is located on the server. The user can have several applications launched on the server and work with all of them simultaneously.

There exists a somewhat similar interface standard (known as FastCGI), which services all incoming requests within the framework of a single process, but does not terminate it after the user breaks the connection. BAIKONUR server can open an arbitrary number of processes and direct information flows from the client to the appropriate process or thread. In this sense, BAIKONUR server solves the most general (and therefore most complicated) remote user-to-application interface problem more efficiently than most existing approaches.

Building of Web applications

For developing BAIKONUR applications in Delphi (or C++ Builder), use is made of Baikonur HTML Controls Library. After setup, the components of that library are located on the "HTML", "HTML DB" and "HTML Add" pages of the Components Palette. The process of connecting the components library is described in the "BAIKONUR. Getting Started" manual. A detailed description of these components can be found in Annex A.

The following Section gives an overview of the basic principles of building Web applications. Specific solutions are given in the "Examples" section.

Types of Web applications

There are three types of Web applications:

- A single-user Web application, whereby a new instance of the application is launched for every user.
- A multiuser Web application, whereby only one instance of the application is launched, but its users do not interact with one another in any way.
- A shared Web application (a variation of the multiuser application), whereby all users can work together in one and the same dialog.

The simplest type of an Web application for a developer is the single-user one, because the technology employed to create it is basically no different from that of creating a conventional application program. For example, it can be a program for viewing and editing data, a program for gaining access to a private bank information, etc. From the point of view of the developer, everything looks as if all the users were working concurrently on a single computer, each with his/her own instance of the program. However, the simplicity of such an approach is fraught with the problem of computer system resources (since the number of program instances that can be launched simultaneously is limited). Therefore we have provided a capability of setting the maximum number of simultaneously launched programs in the BAIKONUR application server's administration utility.

The problem of limited system resources can, to a certain extent, be solved by organizing a multiuser-type Web application (although in that case the programmer must himself ensure that dialog forms would be dynamically built for each user during his/her work session with a program). The maximum number of users of a multiuser Web application increases 10- to 20-fold. Some examples of building multiuser and shared applications are given further in the text.

General Tips of Building an Web application

1. The application's main form must contain a *HTMLControl* component responsible for the interface between the application and the BAIKONUR server.
2. On every form of the application there must be a *THTMLPage* component responsible for generation of the HTML script describing that form.
3. You should use Baikonur HTML Controls library components as the objects for building the user dialog.
4. You can use standard Delphi (or C++ Builder) components ("Data access" page of the Components Palette) to provide access to data.
5. You can use any Delphi components in the program, but they will not be visible in a browser. It therefore makes sense to employ only invisible components (such as *TTimer* or *TIBEventAlerter*). Besides, it is good practice to employ *TPanel* when designing a form.
6. An application may contain more than one form, but it cannot be an MDI application. The forms should not be modal (i.e., you can't use *ShowModal*). It is desirable that once a form is closed, the focus is explicitly returned to the calling form, for example:

```
procedure TForm2.FormClose(Sender: TObject; var Action:
TCloseAction);

begin

    Form1.SetFocus;

end;
```

Form Design Tips

Since the onscreen appearance and placement of objects on an HTML page cannot be arbitrary (they are determined by the Web browser), a Web-browser page can differ substantially from the corresponding Delphi form.

The basic algorithm of designing an HTML page is as follows:

- If the top edge of one object is located above the bottom edge of another object on the form, the two objects will be located on one and the same line on the

HTML page. In this version of HTML Components Library, the object located higher on the form will appear left-most on the HTML page.

- The vertical spacing between objects is calculated based on the size of the form's current font, and equals an integer number of lines.
- The horizontal spacing between objects equals an integer number of character positions and is determined by the control's *Distance* property.

Considering that an HTML page is essentially a set of lines, we recommend that the following technique be used to create an application interface. First, a panel (*TPanel*) is placed on the form and top alignment is specified for it (*Align*=alTop). Then, the objects that must be located on one and the same line on the HTML page are placed on the panel and are left-aligned (*Align*=alLeft). If necessary, the *HTMLRuler* can be placed between the panels.

In this version of BAIKONUR, a design-time form preview capability is not provided. However, you can still obtain the HTML script for a form at design time by double-clicking on the *HTMLPage* object, saving the script to a file and then viewing it from a browser.

Controlling the Application's Behavior

The behavior of an Web application is controlled by the *HTMLControl* component, which must be located on the application's main form. By appropriately setting the properties of that component, you can specify whether the Web application is a single-user or a multiuser one, set its timeout value, etc.

- The *FinalURL* property determines the HTML page that will be sent to the user when the given application is closed. The *HideApp* property determines whether the application will appear in the Task Bar at start up. If the application and all its screen forms are "hidden" (using the *HideForm* property of the *HTMLPage* component), the user will not be able to switch from it elsewhere by using the <Alt>+<Tab> accelerator key. Besides, the application will not appear on the applications list in the Task Manager (although it will show in the list of processes). If BAIKONUR Web Server works as a Windows NT service, the application will not appear on the screen in any case. It will only be visible onscreen when working under Windows 95 or when BAIKONUR is started up in the debug mode.
- The *MultiUser* property indicates whether the service is of the multiuser type. If this property is set to False, a single-user application is implied (whereby a separate instance of the program will be launched for every user). If the same property is set to True, the application will be a multiuser one, with only a single instance of the program launched for all users. The way the interface between such a program and its users can be organized will be discussed further in the text. The setting of the *MultiUser* property has a meaning only at program startup. Changing this property at run time will have no effect on program execution.
- The *TimeOut* property determines how long the program is allowed to idle (in seconds) before it will be closed. If *TimeOut*=-1, the application will not be closed. If the *TimeOut* property is set to 0 (zero), the application will start up, post one screen form to the user, and terminate. If the case of a multiuser application, a timeout event for one of the users will result in that the latter will be merely deleted from the list of users working with this specific application. If it is the last user on the active users list, the application will terminate.

Controlling a Form's Behavior

The onscreen appearance and behavior of a form is determined by the *HTMLPage* component.

- You can specify a background image (in GIF or JPEG format) in the *BackImage* property for the page. If it is not specified or an incorrect file name is specified, the page background color will default to the *clrBackground* property.
- The *CheckFrame* property indicates whether the form's actuality is to be checked. By default, the Web application assigns a unique number to each page it sends out. Each time a next user-originated request is received, a check is made of whether it is associated with the current form. If not, an error message is posted. An error situation may occur if the user clicks the "Back" button in his/her browser to go back to the previous page or a page belonging to another application, and then accidentally clicks a Submit-type button on that "obsolete" form, thereby originating a wrong (in the context of the current program) request. Normally, the default value of this property needs not be changed (see "Browser's "Back" Button and Page Caching" in Chapter II).
- The *HideForm* property determines whether the screen form will be hidden.
- The *Multipart* property determines the document's type. The page being sent out will be treated as a type '*text/html*' document if *Multipart*=False, and as a type '*multipart/mixed*' document if *Multipart*=True. '*Multipart/mixed*' means that the application can send out a page to a user without waiting for that user's request. If necessary, you can use the *SendMultiPage* procedure of the *HTMLControl* component to have a new page sent out. Such a page will be received by all the users who are currently working with it in a browser and did not break the connection.

IMPORTANT. At his time, documents of the "multipart/mixed" type are supported only by Netscape Navigator 2.0 and above.

- You can use the *Title* property to specify the text to be displayed in the browser's caption.

MultiUser Web applications

By default, a Web application will be a single-user one, because the *MultiUser* property of the *HTMLControl* component defaults to False. In a single-user service, a separate instance of the program is launched for every user.

A Web application will be a multiuser one if you set the *MultiUser* property of the *HTMLControl* component to True at design time. All users of a multiuser Web application will be working with a single instance of the program. There are two general kinds of multiuser services, whereby:

- all users are working with their own sets of screen forms and are independent from one another.
- all users are working in one common dialog and can interact with one another.

Let us consider the first method of building a multiuser service. Apparently, this method calls for the use of dynamically created forms.

The program's main screen form will, of course, be created automatically, and it is

this form that all the users accessing your application will normally start working with. You must therefore take this into account when designing your main form. As far as possible, you must not place any input fields on the main form or, if you do so, make sure that they are returned to their original state once an HTML script is sent to the user. For example, the main screen form may contain only a menu with type *HTMLButton* or *HTMLImageButton* buttons. If the main form does include fields for the input of some information (for example, information about the user), it is good practice to ensure that these are returned to their initial state after the received information has been processed and the appropriate HTML script has been generated. You can use the *OnDataSend* event of the *HTMLPage* component to achieve this.
Thus, the principal requirement that the main form of a multiuser-service should meet consists in that all its input fields (if any) must return to their initial state after the appropriate data is sent to the user.
Secondary screen forms of the program must usually be created dynamically (a secondary screen form may be created automatically as well, as long as the above requirement to the main form is met). For example, if clicking on a button on the form must take the user to another form, the code could be as follows:

```
procedure TForm1.HTMLButton1Click(Sender: TObject);

begin

   Form2:=TForm2.Create(Self);

   Form2.Show;

end;
```

Note that you must use the *Show* method rather than the *ShowModal* method.
Once a secondary form is closed, it must be destroyed. It is also desirable that the input focus be set to the calling form immediately after that:

```
procedure TForm2.FormClose(Sender: TObject; var Action:
TCloseAction);

begin

  Action:=caFree;

  if Owner is Tform then

    (Owner as TForm).SetFocus;

end;
```

Let us briefly discuss the second method of building a multiuser service, whereby users can concurrently work with one and the same form. If program logic requires that all users currently working with a given form must see the changes made in the program by one of the users, it would be good practice to employ the "*multipart/mixed*" document type (see the *Multipart* property of the *HTMLPage* component). Multiuser services of this type are obviously still fairly rare. They are mostly employed for performing such tasks as organizing server chats, viewing stock exchange quotation, viewing database HTML pages, and the like.

User's Interface Limitations

To the programmer, building an Web application is no different from creating a conventional application in Delphi, although there do exist a few limitations. These limitations are determined by certain specific features of the HTTP protocol and the HTML standard.

Events

The main limitation is that the browser generates no events other than a Submit-type button click event. Consequently, all changes made by the client in the form's control elements are passed to the application simultaneously when a Submit-type button is clicked. Besides, the order in which objects are updated in the application can be arbitrary. The only sure thing is that the button will not be clicked until after all other objects are updated. You must therefore create such an application whose status would not depend on the order in which objects on the form are edited.

In the browser, the user interface turns out to be poorer in its functional capabilities compared to the commonly familiar interface of the programs written in Delphi. Thus, most of the existing browsers do not support "drag-and-drop" capability, cannot respond to mouse- or keyboard-generated events, etc.

You can enhance your interface somewhat through the use of Java scripts, Java applets or ActiveX objects. As far as Java scripts are concerned, they are supported by this version of the library. The other extensions can, in principle, be used now as well, but subsequent versions of the library will have special controls built in to support these extensions.

Forms

You must also take into account the fact that only a single form can be displayed in a browser at a time. Your application can be a multi-window one, but in this version of the library you cannot use modal screen forms (*ShowModal*). And, of course, you must preclude the use of standard dialogs (such as *MessageDlg* or *ShowMessage*) and objects from the «Dialogs» page of Delphi Components Palette, because no HTML scripts are generated for them. Otherwise, the client will have no way of knowing that a screen form of this type has been opened in the application, and will not be able to close it.

The placement of items on the form cannot be arbitrary either, although it is possible to build fairly diverse pages by using the HTML table tools (*HTMLTable* component, the <TABLE> tag).

Components

There are substantial differences between HTML objects and Delphi's standard controls (in the currently available browser versions):

- text font control capabilities are less flexible;
- conponents have no mouse- and keyboard-driven events;
- there are no customary drop-down or pop-up menus;
- there are no owner-draw components.

The peculiarities of the user-to-application interface via a browser have a bearing on the way data-aware components are functioning. In the current version of the library, data is updated as follows. The user may edit the contents of the input field (*HTMLDBEdit*, *HTMLDBMemo*, *HTMLDBCheckBox*) and click a submit-type button. The modified data will be sent to the application and saved in the database if:

- the data set is editable (i.e., the *ReadOnly* property for *TTable* is set to False or the *RequestLive* property for *TQuery* is set to True);
- the field is allowed to be edited (i.e., the *ReadOnly* property of the input field associated with the given element is set to False);
- the input data matches the field type (otherwise, an error message will be generated);
- the new data does differ from the old one;

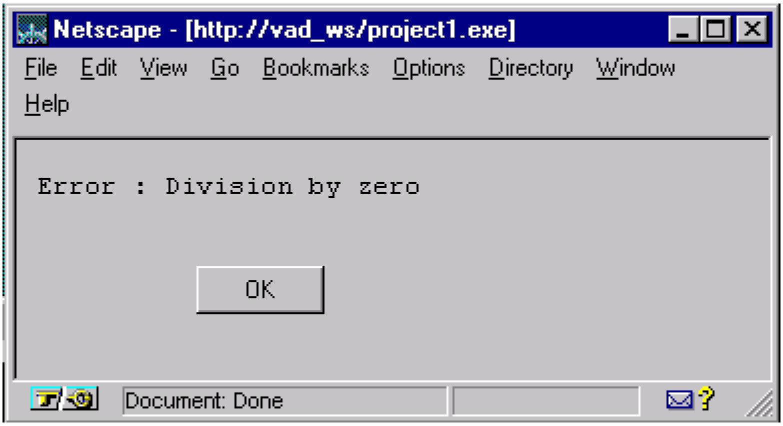
If the data set is in the data browse mode (*dsBrowse*) at the instant of arrival of fresh data, it will be changed to the edit mode (*dsEdit*).

HTML tools are inadequate to organize verification of the input data on the client's side. The user may therefore input, for example, an incorrect date. An attempt to save such a date will cause an exception, and a corresponding message will be posted to the client. Verification of user-input data can be organized by utilizing JavaScript capabilities.

In most cases, the *HTMLNavigator* component is employed to navigate through tables, the process being as follows. The user modifies data in the input fields and clicks the "Refresh" button or "Next" button. The forms' description is passed to the application and a check is made of whether or not the contents of the input fields have changed. If they have, the data is saved in the table.

Handling of Exceptions

All unhandled exceptions occurring between the receipt of a user request and the sending of a response are intercepted and processed by the *HTMLControl* component. By default, an HTML script with an error message is generated. The default error message appears as shown in Figure 1.


Netscape - [http://vad_ws/project1.exe]
File Edit View Go Bookmarks Options Directory Window
Help
OK
Document: Done

```
Error : Division by zero
```

Figure 1. Error Message

As an application developer, you can interfere into the exception handling process in two ways: (a) with the help of a try ... except construct, or (b) by creating an *OnException* event handler of the *HTMLControl* component. With such an event handler, you can send an HTML script with a custom error message to the user, or perform some actions before simply returning the current form's script.

To have a custom error message sent as an HTML script, you can use the *SendErrorScript* procedure of the *HTMLControl* component, as shown in the following example:

```
procedure TForm1.HTMLControl1Exception(Sender: TObject;
E: Exception);

var

  s : string;

begin

  if E is EDivByZero then

    s:='Division by zero.'

  else

    s:=E.Message;

  HTMLControl1.SendErrorScript(s);

end;
```

To perform certain actions and then send out the current form's script, you can write something like this:

```
procedure TForm1.HTMLControl1Exception(Sender: TObject;

  E: Exception);

var

  s: string;

begin

  StatusLabel.Caption:=E.Message;

  s:=HTMLPage1.HTMLPageScript;

  HTMLControl1.SendResource(@s[1], Length(s), "text/
html");

end;
```

Debugging of Web applications

This version of the library does not support Web application run-time debugging capability from Delphi's integrated debugging environment (although this capability will undoubtedly be built into subsequent versions). You can, however, debug applications executed under the control of BAIKONUR with the aid of Borland's Turbo Debugger. To do so:

- Check the "Include TDW debug info" option on the Linker page in the project's settings.
- Compile your application.
- Run the application by issuing a request from the browser.
- Start the Turbo Debugger.
- Attach to the running program by checking the TDW option in the File|Attach menu.

Application debugging can be performed in Windows NT (when BAIKONUR operates in the debug mode) or Windows 95 environment.

While debugging an application, it is good practice to view the data received from and sent to the client. The received data (which also includes the HTTP header) is accessible in the *OnReceive* event handler of the *HTMLControl* component. The sent-out data is accessible in the *OnSend* event handler of the *HTMLControl* component.

Browser-Specific Features

Different HTML browsers display one and the same differently and support HTML 2.x and 3.x standards in different ways. Some browsers have their own HTML extensions. Therefore we recommend that you check your applications with various browser versions.

This version of Baikonur HTML Controls Library supports the use of '*multipart/mixed*' type documents. Documents of this type are currently supported only by Netscape 2.0 and above.

Delphi HTML Controls library may not contain some element which a certain browser does support. For instance, there is no <marquee> element supported by Microsoft's Internet Explorer in the library. In order to place such a tag on a form, you must use the *HTMLTag* component as follows:

Place the *HTMLTag* where you need it on the form.

Specify '<marquee> Some scrolling text </marquee>' in the *Caption* property.

In MS Internet Explorer, this element will be displayed as scrolling text.

Chapter II

In this Chapter we analyze a number of examples illustrating the techniques employed to develop applications and control their behavior at run time.

Running and Closing of Application, Passing of Parameters

Running of Application by the User

The user can run an application by specifying its URL in the browser, for example as: *http://web_serv/demo.exe*. Obviously, this is not very convenient for the end user. The best way is to place a reference to that application on one of the static HTML pages. The application would then be launched when the user clicks on that reference with the mouse.

One and the same application can be called for execution with the use of either the application's name or its alias. For example, if the alias for the DEMO.EXE program is specified as *baik_demo.html=c:\baikonur\home\demo.exe* in the server settings, an alternate way of running it would be to specify its URL as *http://web_serv/baik_demo.html*.

Passing of Parameters to the Application

You can pass certain parameters to an application at start-up time as well as at run-time. Since an application can be a muiltiuser one, the passing of parameters is performed in a somewhat unusual way. To have a program called with parameters, you must specify them in that program's URL. For example, you can specify http://web_serv/demo.exe?params=/p+/c+qqq+/D%3a90 (which will be equivalent to launching the *demo.exe* program from the command line by specifying *demo.exe /p /c qqq /D:90*). That is, you append the application's URL with the question mark "?" and the "params=" keyword followed by the parameters proper. Note that all space characters should be replaced with the "plus" character ("+"), and any other "dangerous" characters– with their hexadecimal equivalents. For example, the colon character (":") must be replaced with %3a.

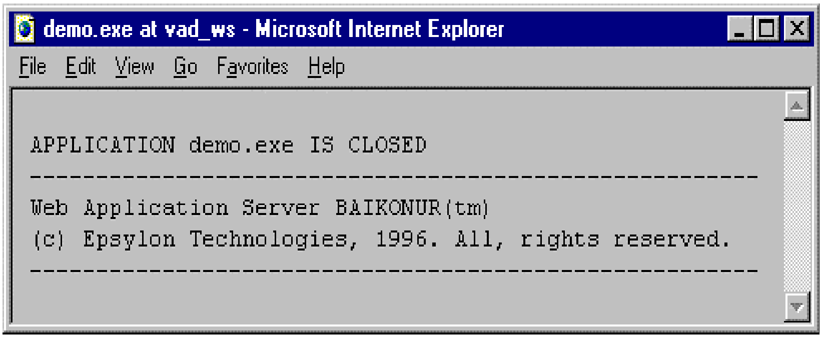
It is important to realize that parameters in an application are accessed via the *Params* property set for a given user, and NOT via the *ParamCount* and *ParamStr* variables. For example, if you specify *HTMLLabel1.Caption:=HTMLControl1.CurrentUser.Params* somewhere in the application, then *HTMLLabel1* shall point to the string "/p /c qqq /D:90".
You can specify an individual set of parameters for every user.
IMPORTANT. Parameters for an application can be specified not only at program start-up time, but at any time during program execution as well. In the latter case, the value of the *Params* property for the user will be changed to match the newly specifuid parameters.

Closing Application

You can close a program from the browser by specifying ".!" after the ".exe" extension in that program's URL. For example, the URL specified as *http://web_serv/demo.exe.!* will close the *demo.exe* application. If the application has "hung up" and does not respond to any requests, you can still terminate it by issuing an URL like *http://web_serv/demo.exe.!?terminate* - this will have the same effect as removing the task via the Task Manager by clicking the "End Process" button.
You can also place a reference with a similar URL (http://web_serv/demo.exe.!) on the application's form; clicking on that reference will likewise terminate the task.
Besides, an application can be closed on the click of a button if you specify the following in that button's *OnClick* event handler:

```
procedure TForm1.HTMLButton1Click(Sender: TObject);

begin

  HTMLControl1.UserClose;

end;
```

When an application is closed, a screen form with a message informing the user that the application is closed (default final page) will be posted to the user:


demo.exe at vad_ws - Microsoft Internet Explorer
File Edit View Go Favorites Help

```
APPLICATION demo.exe IS CLOSED
-------------------------------------------------------
Web Application Server BAIKONUR(tm)
(c) Epsylon Technologies, 1996. All, rights reserved.
-------------------------------------------------------
```

Figure 2. Default Final Page

It is often desirable to change the application’s default termination behavior and display some static HTML page on the screen instead. If some URL is specified in the *FinalURL* property of the *HTMLControl* component, then a corresponding page will be show in a browser after closing application.

Termination of Task on TimeOut

The *HTMLControl* component includes a *TimeOut* property. Setting this property to a positive value greater than zero will lead to automatic termination of the corresponding application for the “idling” user. *TimeOut* is essentially the period of time (in seconds) elapsing after the most recent instant of user access to the application. Once the *TimeOut* period expires the application automatically deletes the user information from memory (although this does not result in the release of system resources captured by the user, if any). If the user is the last one to access it, the application will be closed. If subsequently the user issues a request to the application closed on timeout (for example, by clicking a button on the form of the application that has already been closed), he/she will receive the following error message:

Clicking the “OK” button will take the user to the application’s main form, and he/she will have to start work

from the very beginning. You can individually set timeout value for every user in application by using *TimeOut* property of TuserInfo class.

We shall discuss how an application is closed on timeout and how system resources are released in greater detail later in the text, while describing a sample multiuser SQL server application.

Deleting the User From Application

Apart from the normal program termination technique described above, there is yet another method of deleting the user from the application (i.e., closing the application for the user). The conditions under which this takes place are determined by the program itself. Thus, you can have a user “disconnected” from the application if he/she violates certain pre-defined rules (for example, exceeds the allowed time limit of using the program). For deleting a user from the application, use is made of the *DeleteUserByID* method of the *HTMLControl* component . For the user, this will appear as a timeout. You can call this method at any time except while the incoming request is being processed (most frequently, on timer). In the latter case, use should be made of the *UserClose* method.

Like the *UserClose* method, the *DeleteUserByID* method merely deletes information on a specific user from memory, but does not release the resources captured by that user (the developer himself must see to it that the resources are freed).

Here is an example of the “delete user by ID” method:

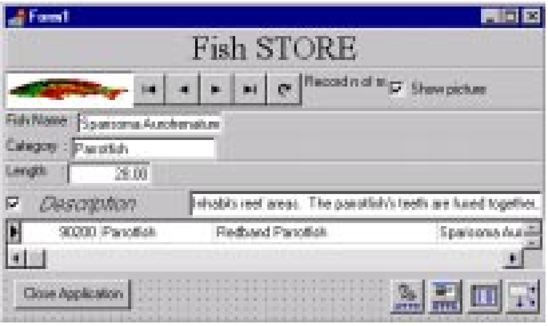
```
procedure TForm1.Timer1Timer(Sender: TObject);

var

  i: Integer;

begin

  with HTMLControl1 do

    for i:=0 to UserCount-1 do

      if TimeLimitExceed(Users[i]) then begin

        FreeResource(Users[i]);

        DeleteUserByID(Users[i].ID);

      end;

end;
```

Designing a User Interface

Placement of Components on a Form

The way components are placed on a form at design time is conditioned by the peculiarities of the Hypertext Markup Language. Since an HTML page can be thought of as a set of lines in which individual items follow one another, these lines can be “emulated” at design time by means of conventional *Tpanel* panels. The panels are placed on the form and are top-aligned (*Align*=alTop). The components are placed in a panel and are left-aligned (*Align*=alLeft). This guarantees that in a browser these components will be placed exactly as they had been arranged at design time. The horizontal distance, or spacing, between individual items (in character positions) is determined by the *Distance* property of each visible HTML component.

An example of how a form can be built following this algorithm can be the DBTOTAL program - a form designed to represent the BIO_GIF.DB table. The form's view at design time is shown in Figure 3.



Fish Name : Sparisoma Aurofrenatum
Category : Parrotfish
Length : 28.00
☑ Description Inhabits reef areas. The parrotfish's teeth are fused together.
90030 Parrotfish Redband Parrotfish Sparisoma Au

Figure 3. View of Form at Design Time

The same form will appear in a browser as follows:

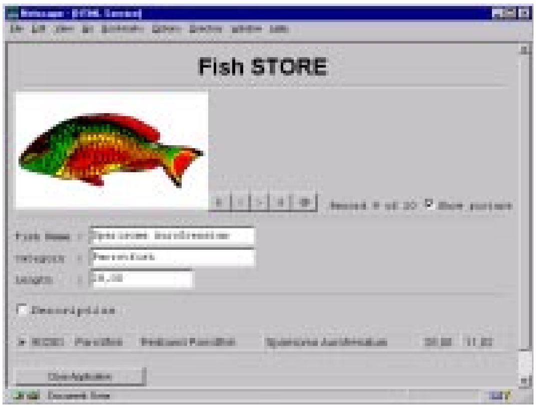


Figure 4. View of Form in a Browser

There are, however, certain alignment peculiarities in situations when an image must be placed on a form. Graphic objects of the HTML components library have an *AlignImage* property, which determines the alignment of the image on the page relative to its other elements. In the above example, that property for the HTMLDBGIF component was set to aiDefault. If you set this property to aiRight, the page will appear as follows:

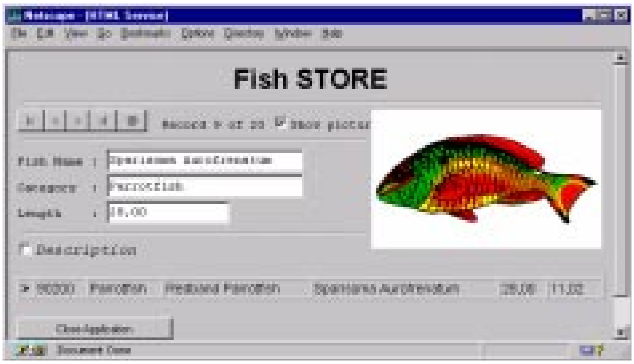


Figure 5. Image Alignment

Controlling the Appearance of Text on a Page

The way text will appear on a page is to a great extent dependent on the browser. Although the HTML includes a large number of tags designed to control text properties, not all of them are supported by different browsers. Besides, the name of the font and its base size are normally conditioned by the browser's current settings. You can specify a different font name on an HTML page when FontFace property of HTMLPage component is True. But it leads to increasing of HTML script size.

When working with Baikonur HTML Controls library's text components, you can specify:

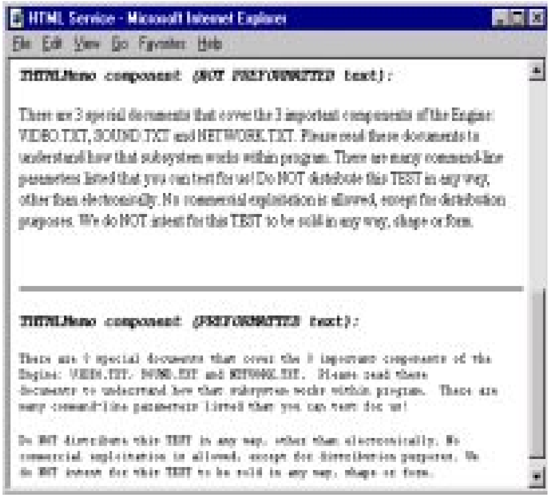
- font type (proportional or monospaced);
- font face (when *FontFace* property of *HTMLPage* is True)
- font color;
- font size;
- font style (bold, italic, underlined, strike-out).

The text components include *HTMLLabel* and *HTMLDBText*, *HTMLMemo* and *HTMLDBMemo* (when *Style*=msText), *HTMLCheckBox* and *HTMLDBCheckBox*, *HTMLRadio*, *HTMLList, etc*.

The font of a component is assumed to be the base font if it matches the form's font. If that font's properties differ in any way, then a corresponding tag must be placed on the HTML page.

The font type is determined by the *Preformat* property of the corresponding components. If you set this property to False, the browser will display the text in proportional font and "autoformat" it (i.e., delete all "unnecessary" white spaces and line breaks and adjust the text so that it would fit the width of the browser's screen page). If you set the *Preformat* property to True, the browser will display the text in monospace (fixed-width) font, and no "autoformatting" will take place.

Figure 6 shows how preformatted and non-preformatted texts will appear on a form (the TOTAL program):


HTML Service - Microsoft Internet Explorer
File Edit View Go Favorites Help

THTMLMemo component (NOT PREFORMATTED text):

There are 3 special documents that cover the 3 important components of the Engine: VIDEO.TXT, SOUND.TXT and NETWORK.TXT. Please read these documents to understand how that subsystem works within program. There are many command-line parameters listed that you can test for us! Do NOT distribute this TEST in any way, other than electronically. No commercial exploitation is allowed, except for distribution purposes. We do NOT intent for this TEST to be sold in any way, shape or form.

THTMLMemo component (PREFORMATTED text):

```
There are 3 special documents that cover the 3 important components of the
Engine: VIDEO.TXT, SOUND.TXT and NETWORK.TXT.  Please read these
documents to understand how that subsystem works within program.  There are
many command-line parameters listed that you can test for us!

Do NOT distribute this TEST in any way, other than electronically. No
commercial exploitation is allowed, except for distribution purposes. We
do NOT intent for this TEST to be sold in any way, shape or form.
```

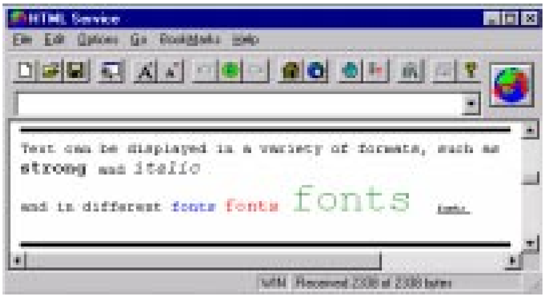
Figure 6. Text Formatting Examples

The font color is determined by the components' *Font.Color* property. Note that some browsers have no color font support capability.

The font size is determined by the components' *Font.Size* property. However, the HTML supports only 7 font sizes. Therefore the size of the font of a component on the page may differ somewhat from its size at design time.

The font style is determined by the components' *Font.Style* property. Again, not all browsers support all the font styles that can be specified with HTML.

Figure7 gives an example of different font styles, sizes and colors:



Text can be displayed in a variety of formats, such as **strong** and *italic*

and in different fonts fonts fonts fonts

Figure 7. Examples of Font Styles, Sizes and Colors

Apart from the text formatting capabilities implemented in the library, the HTML also includes a multitude of other tags affecting the appearance of text. They can be placed on a page by using the *HTMLTag* component. For example, if you assign a string like

<SMALL>This text is smaller in size.</SMALL>

to the *Caption* property of the *HTMLTag* component, the text will be output in a browser one size smaller than the base font.

Use of the *THTMLTable* Component

As is well known, a HTML document is little more than a set of lines that may include text, pictures, input fields, buttons, etc. Therefore the onscreen appearance of the user dialog turns out to be rather primitive. You cannot, for example, have several vertically stacked input/edit fields placed to the right of a memo field.

To obtain a more complex and attractive interface, use can be made of the *THTMLTable* component. This component is displayed in a browser in the form of a table (with or without borders) in which you can place text or any objects (such as images, input fields, button controls, etc.). Without the use of a table, for example, it is impossible to build a page on which several radio buttons are located to the right of a Memo object or a ListBox.

At design time, *HTMLTable* appears as a panel partitioned into several cells. The number of rows and columns in the table is determined by its *Rows* and *Cols* properties, respectively. The object placed into a table is assumed to be located in the cell which its upper left-hand corner is located in. You can specify alignment type for a component in the cell (for example, *Align*=alClient), in which case the component will occupy the entire cell. You can place several objects and even another *HTMLTable* component into one and the same cell. To simplify the placement of several components in a table cell, you can first place a usual panel into the cell and only then place the necessary components there. Components can be moved from one table cell to another.

The way a table will appear in a browser is determined by the *Border* and *WidthOnPage* parameters. The *Border* parameter sets the width of the table's border. If *Border*=0, your table will have no border around it. The *WidthOnPage* parameter determines the width of the table on the browser page (in percent). If *WidthOnPage* = 0, the browser will "autoadjust" the table to its screen page width.

If you use tables, bear in mind that pages containing tables take longer to be displayed in browsers.

Here is an example of an HTML table at design time:

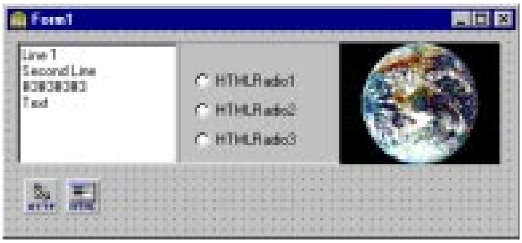
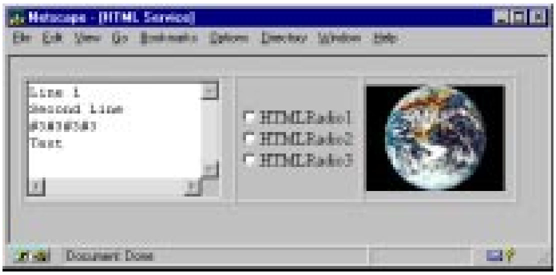


Figure 8. Table at Design Time

The same table will appear in a browser as follows:

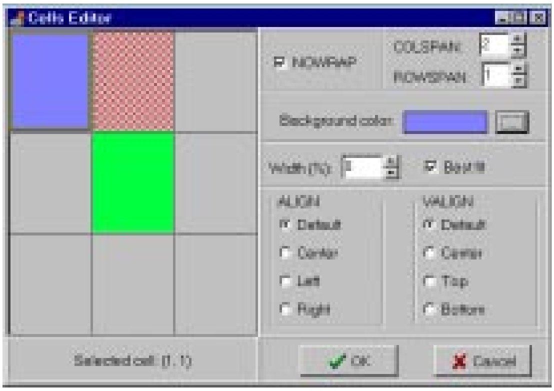

Netscape - [HTML Service]
File Edit View Go Bookmarks Options Directory Window Help
Document: Done

Line 1
Second Line
#3#3#3#3
Text

○ HTMLRadio1
○ HTMLRadio2
○ HTMLRadio3

Figure 9. Table in Browser

At design time, you can individually “tune up” every table cell by specifying its background color, alignment and other parameters. To invoke the cells editor, double-click on HTMLTable or invoke the *Cells* property editing dialog in the Objects Inspector. An example of such a dialog is illustrated by the following picture:



☑ NOWRAP
COLSPAN: 2
ROWSPAN: 1
Background color:
Width (%):
☑ Best fit
ALIGN
● Default
○ Center
○ Left
○ Right
VALIGN
● Default
○ Center
○ Top
○ Bottom
Selected cell: (1, 1)

Figure 10. Editing of HTML Table Cells

For each cell you can specify whether the text wrapping feature will be enabled (NOWRAP), how many columns and rows a given cell will span (COLSPAN and ROWSPAN), what background color the cell will have and how wide it will be (in percent of the table width), and how it will be aligned horizontally and vertically (Align and Valign, respectively).

Use of *THTMLDynamicTable* Component

Using a dynamic HTML table, the developer can place on the form a table the appearance of which will vary dynamically at run time. Thus, you can add new rows and columns to a dynamic HTML table, change color of its cells, etc. The version of the library under discussion supports only the placement of text in dynamic table cells. If, however, you add the text of an HTML tag into a cell, that object will be visible there in a browser. Unlike a static HTML table, a dynamic table can include up to 32767 columns and as many rows. It should be borne in mind, however, that when the table contains a large number of cells, program execution speed slows down considerably (both at design time and at run time).

At design time, the text in and the format of dynamic table cells are adjusted with the help of the component's cells editor (which can be invoked by double-clicking the cell) or by accessing the *DynCells* property of the Objects Inspector. The dynamic table cell editing dialog is illustrated below:

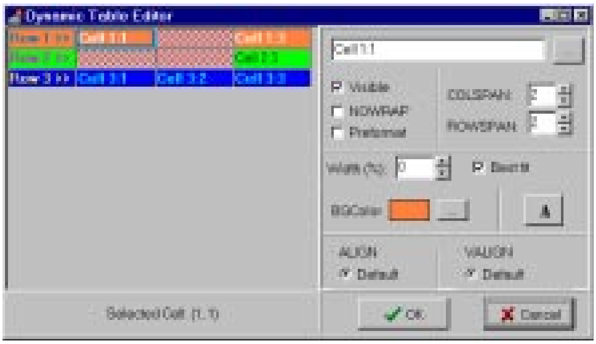


Figure 11. Dynamic Table Editor Session for Adjusting Cell Parameters

You can specify the text to be displayed in a specific cell by entering it in the input field or by invoking the text input dialog (the "..." button). You can specify the properties of individual cells or the properties of an entire row of cells. For each cell you can specify whether it will be visible (Visible), whether the text wrapping feature will be enabled (NOWRAP), whether the text will be preformatted (Preformat), how many adjacent columns or rows the cell will span (COLSPAN and ROWSPAN), how wide the cell will be (in percent of the table width), what background color it will have (BGColor), what font will be used to display text in the cell (button A), and how the cell will be aligned horizontally and vertically (Align and Valign, respectively).

The dynamic table will appear in a browser as shown below:

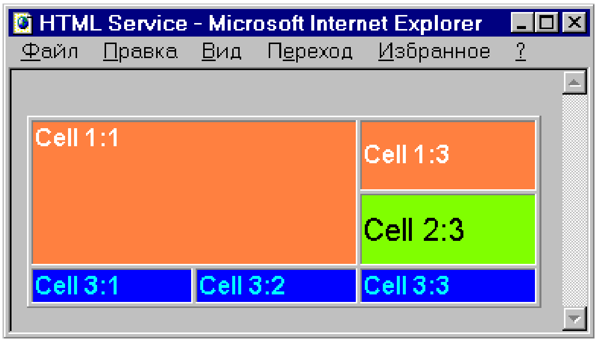


Figure 12. Dynamic HTML Table in Browser

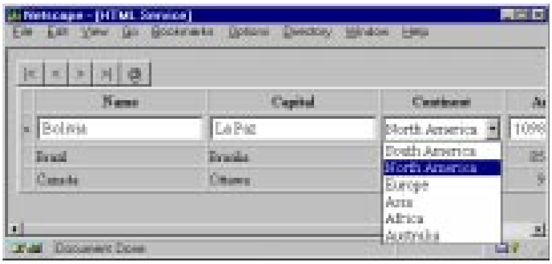
At run time, you can control the properties of individual cells, columns and rows of your dynamic HTML table by accessing its *Cell*, *Col* and *Row* properties, respectively. See the component's description to learn more about these properties.

If you wish to minimize the size of the HTML script sent from the application to the client, you should set the component's *CoolTable* property to False. You will still be able to control the cell's contents, its visibility, and COLSPAN and ROWSPAN parameters. However, such parameters as background color, font color and type and alignment will then be set to their default values.

Use of *THTMLDBGrid* Component

The *THTMLDBGrid* component is a direct descendant of *TCustomDBGrid*, but the way it is presented on an HTML page is somewhat different.

You can navigate between records within the grid only with the help of the *THTMLNavigator*. If data is marked as editable, the browser will display input fields in the current record. If the field contains a list (i.e., *Pick List* was specified for the field at design time), the browser will display a combobox with a list in that field, as shown below:



Name	Capital	Continent
Bolivia	La Paz	North America
Brazil	Brasilia	
Canada	Ottawa	

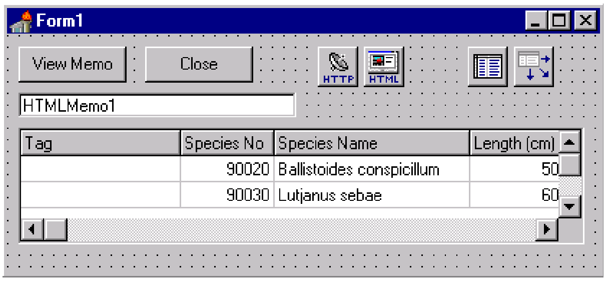
Figure 13. View of *HTML DBGrid* in a Browser

There is also a grid operating mode that differs somewhat from what most programmers are accustomed to. This mode takes effect when the *ShowAll* property is set to True. In that case the browser will display all records from the source data set, but you will not be able to navigate through the table or edit its data. Although browsing through all the table records is often all that is needed, sometimes you would want to be able to select one of them and perform some action with it.

Let us consider a sample program (see the DBTOTAL program). Here, we want all records from the BIOLIFE.DB table to be visible onscreen, and a radio button displayed against each record. When the user checks one of these buttons and clicks the Submit button, the program should return a description from the corresponding Memo field.

To have a radio button placed into the grid, you can use the following technique. Add a computable string-type field to the table, and assign to it the string containing the corresponding HTML tag in the *OnCalcFields* event handler for *Ttable*. In the browser, this field will appear as a radio button. After the application receives a request from the browser (*OnUpdate* event for *HTMLPage*), you will have to check which of the buttons was selected and update the memo field contents accordingly.

Place the *THTMLControl*, *THTMLPage*, *TTable*, *TDataSource*, *THTMLDBGrid*, *THTMLMemo* components and two *THTMLButton* buttons on the form. At design time, your project's form will appear approximately as shown in Figure 14:



Tag	Species No	Species Name	Length (cm)
	90020	Ballistoides conspicillum	50
	90030	Lutjanus sebae	60

Figure 14. Project Form in the Designer

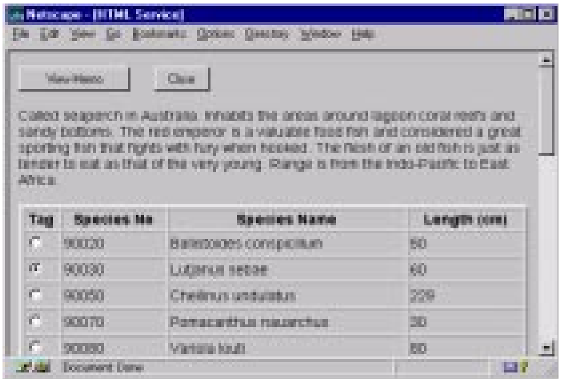
1. Tune *TTable* to the BIOLIFE.DB table from the DBDEMOS alias, and use the Fields Editor to add the *SpeciesNo*, *SpeciesName*, and *Length(cm)* fields to it. Create a computable string-type field 100 characters long and name it *Tag*.
2. Set the grid's *ShowAll* property to True and the *dgIndicator* option to False.
3. Write the *OnCalcFields* event handler for *TTable*:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
var
  s: string;
begin
  {generate radio button's tag}
  s:='<input type=radio name=»R1" value=' +
     Table1SpeciesNo.AsString;
  {is radio button associated with the viewed records
checked?}
  if Key=Table1SpeciesNo.AsString then s:=s+' checked';
  s:=s+'>';
  Table1Tag.AsString:=s;
end;
```

1. Write the *OnUpdate* event handler for *HTMLPage*:

```
procedure TForm1.HTMLPage1Update(Sender: TObject;
  ValueList: TStringList);
begin
  {what button is selected?}
  Key:=ValueList.Values[“R1”];
  {if selected, read memo field}
  if Key<>’’ then begin
    Table1.FindKey([StrToInt(Key)]);
    HTMLMemo1.Text:=Table1Notes.AsString;
    HTMLMemo1.Visible:=True;
    {save selected record}
    Key:=Table1SpeciesNo.AsString;
  end;
end;
```

The *Key* variable should be declared as global.
Your project is now ready. In a browser, it will appear as follows:


Netscape - [HTML Service]
File Edit View Go Bookmarks Options Directory Window Help
View Memo
Close
Document: Done

Called seaperch in Australia. Inhabits the areas around lagoon coral reefs and sandy bottoms. The red emperor is a valuable food fish and considered a great sporting fish that fights with fury when hooked. The flesh of an old fish is just as tender to eat as that of the very young. Range is from the Indo-Pacific to East Africa.

Tag	Species No	Species Name	Length (cm)
○	90020	Ballistoides conspicillum	50
●	90030	Lutjanus sebae	60
○	90050	Cheilinus undulatus	229
○	90070	Pomacanthus nauarchus	30
○	90080	Variola louti	80

Figure 15. View of Project in a Browser

Examples of Multiuser Applications

Obtaining a multiuser application is seemingly straightforward – all you have to do is set the *MultiUser* property of the *HTMLControl* component to True. Once you do that, all user requests will be addressed to one and the same instance of the application. However, the developer in this case is sure to bug himself with questions like “how do I know who the request has originated from?”; “how do I ensure that the users would not “interfere” with one another while working with one and the same form?”; “how do I determine when a user disconnects from the program?”, etc. Let us see how these problems can be solved.

Connection to SQL server

An Intranet application developer often encounters the situation whereby several users with their unique names and passwords must be able to access and use data of an SQL server concurrently but independently from one another. As was already said earlier, the obvious solution in such a case is to give every user access to his/her own instance of a dynamically generated form. Let us see how this solution can be implemented, using the InterBase table as an example (see the IB_CONN example).

Obviously, a user wishing to gain access to the InterBase database must first specify his/her name and password. Therefore the application’s form #1 (main form), which is static and will be common for all users of the application, must prompt the user to input his/her login name and password. When the user does that and clicks the “OK” button, a second form (form #2) is dynamically generated, a connection with the InterBase (the *DataBase* component) is established and the table is opened, enabling the user to proceed with his/her work. Each user gets his/her own instance of the second form to work with, and does not interfere with the work of other users. If no connection takes place, the form #2 is deleted and the user receives an error message. Clicking the “OK” button in the error message returns the user to the main screen form.

So, our project’s form #1 will appear as follows at design time:

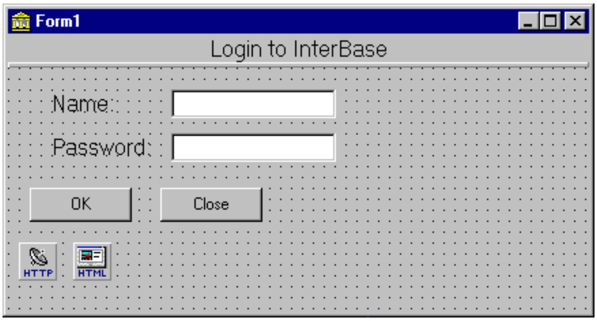


Figure 16. Form #1 at Design Time

Form #2 at design time will look like this:

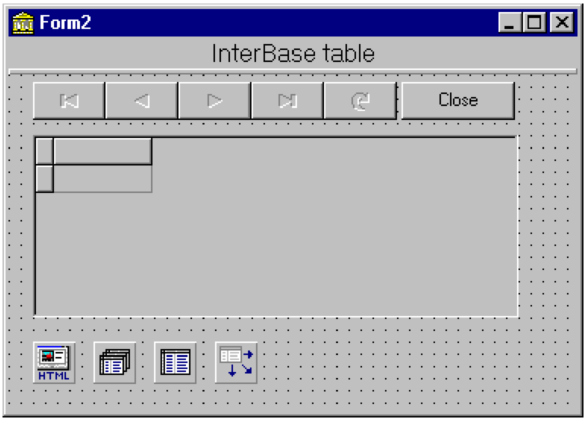


Figure 17. Form #2 at Design Time

The “OK” button click event handler for Form #1:

```
procedure TForm1.HTMLButton1Click(Sender: TObject);
begin
  {create a new instance of Form 2}
  Form2:=TForm2.Create(Self);
  try
    with Form2.DataBase1 do begin
      {database name should be unique!!!!}
    DataBaseName:=’Temp’+IntToStr(HTMLControl1.CurrentUser.ID);
      {request user name and password}
      Params.Values[“USER NAME”]:=HTMLEdit1.Text;
      Params.Values[“PASSWORD”]:=HTMLEdit2.Text;
    end;
    {clear input fields - Form 1 is to be shared by all}
    HTMLEdit1.Text:=’’;
```

```
      HTMLEdit2.Text:='';

      {open table and display Form 2}

     Form2.Table1.DataBaseName:=Form2.DataBase1.DataBaseName;

      Form2.Table1.Open;

      Form2.Show;

    except

      {connect error - delete form}

      Form2.Free;

      raise Exception.Create("Incorrect username or
  password.");

    end;

  end;
```

Hence, an instance of Form #2 is automatically created for every user wishing to access the database, provided that he or she enters a correct username and a valid password. Now, we have to insure that this instance is deleted after the user leaves the application. Judging by the application's structure, this can be done in the *OnUserGone* event handler of the *HTMLControl* component:

```
  procedure TForm1.HTMLControl1UserGone(Sender: TObject;
  UserID: Integer);

  begin

    {which form is user in?}

    {if in Form 1 - Form 2 is not created yet}

    {if in TForm 2 - it should be destroyed}

    if HTMLControl1.CurrentUser.ActiveForm is TForm2 then

      HTMLControl1.CurrentUser.ActiveForm.Free;

  end;
```

Organizing a User-to-User Interface in a Program

It is sometimes necessary to organize an interface between the users within the framework of one and the same application. Let us see how this can be done, using the CHAT (multiuser chat server) program as an example (see also the NN_DEMO program).

Form #1 in this example prompts the user for his/her name. Generally, it is possible to identify the user based on the *CurrentUser* or *Users* property of the *HTMLControl* component, but BAIKONUR server can also work in a mode whereby no username or password are polled for (see "User Authorization" Section).

So, the main form of our application will query the user for his/her name:

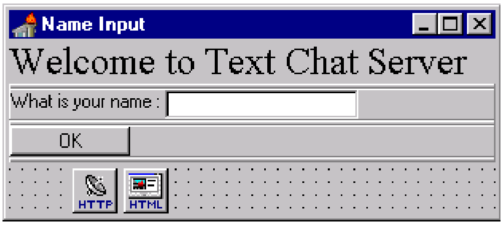


Figure 18. Requesting the User Name

The form's module will contain the following:

```
...
{user data record}
type
  UserData = record
    ID : DWord;
    Name : string;
  end;

{let no more that 10 users can be working concurrently}
const
  MaxUsers = 10;

var
```

```
  Ud : Array[1..MaxUsers] of UserData;
  Ui : Byte;

implementation
...
{initial settings}
procedure TForm3.FormCreate(Sender: TObject);
begin
  FillChar(Ud, SizeOf(Ud), #0);
  Ui:=0;
end;

{find free cell in usernames array}
function GetFreeCell: Byte;
var
  i: Byte;
begin
  Result:=0;
  for i:=1 to MaxUsers do
    if Ud[i].ID=0 then begin
      Result:=i;
      Break;
    end;
end;
```

When the user clicks the “OK” button, his/her name is entered into the array of usernames:

```
procedure TForm3.HTMLButton1Click(Sender: TObject);
begin
  Ui:=GetFreeCell;
  {if there is free space in array}
  if Ui<>0 then begin
    {fill in user record}
    Ud[Ui].ID:=HTMLControl1.CurrentUser.ID;
    Ud[Ui].Name:=HTMLEdit1.Text;
    HTMLEdit1.Text:='';
    {go to Form 2}
    Form1.Show;
  end
  {if the user is one too many}
  else
    raise Exception.Create("Too many users");
end;
```

An attempt of the 11th user to connect to the program should be “parried”:

```
procedure TForm3.HTMLControl1NewUser(Sender: TObject;
  UserID: Integer);
begin
  if GetFreeCell=0 then
    HTMLControl1.UserClose;
end;
```

Form #2 of the program is where the user-to-user chat will actually be displayed. In our case, this form should be of the ‘*multipart/mixed*’ type. To achieve this, we must set the *Multipart* property of the *HTMLPage* component to True. Besides, we must ensure that one and the same instance of the form is employed for all the users. Our form #2 will look in the designer as follows:

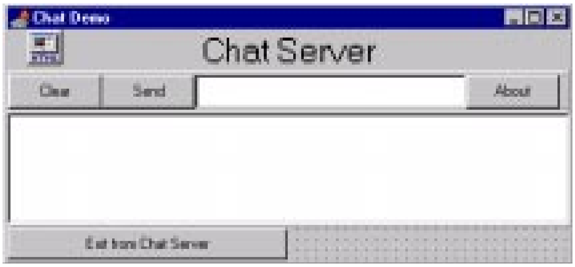


Figure 19. Chat Demo Form in Designer

When the user types in a message in *HTMLEdit* and clicks the “Send” button, the message must be moved into the *HTMLMemo* field, and the updated form should be sent to all the users who are currently viewing it:

```
procedure TForm1.SendClick(Sender: TObject);

var

   UName: string;

begin

  UName:= UserByID(Form3.HTMLControl1.CurrentUser.ID);

  HTMLMemo1.Lines.Add(UName +’:’+HTMLEdit1.Text);

  HTMLEdit1.Text:=’’;

  Form3.HTMLControl1.SendMultiPage(HTMLPage1, False);

end;



{UserByID returns username by his/her ID}

function UserByID(ID: DWord): string;

var

  i : Byte;
```

```
begin
  Result:=’Unknown’;
  for i:=1 to MaxUsers do
    if Ud[i].ID=ID then begin
      Result:=Ud[i].Name;
      if Result=’’ then
        Result:=’No “+IntToStr(Ud[i].ID);
      Break;
    end;
end;
```

Clicking the “Clear” button should clear the Memo field:

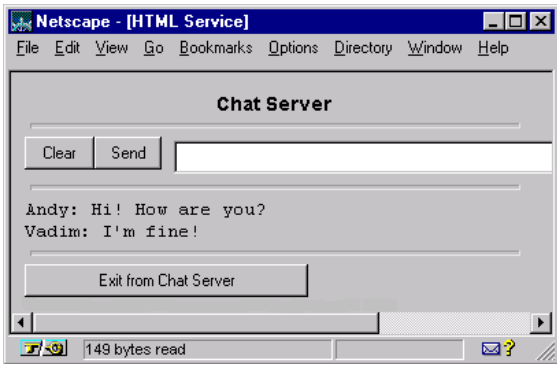
```
procedure TForm1.ClearClick(Sender: TObject);
begin
  HTMLMemo1.Lines.Clear;
  Form3.HTMLControl1.SendMultiPage(HTMLPage1, False);
end;
```

When one of the users exits from the chat server program, his/her name should be purged from the users list:

```
procedure TForm1.HTMLButton4Click(Sender: TObject);
var
  i: Byte;
begin
  for i:=1 to MaxUsers do
    if Ud[i].ID=Form3.HTMLControl1.CurrentUser.ID then
    begin
```

```
      Ud[i].ID:=0;

      Form3.HTMLControl1.UserClose;

      Break;

    end;

end;
```

IMPORTANT. Since the above program relies on the use of the ‘multipart/mixed’ document type, it will function only in Netscape Navigator 2.0 and above.
At program run time, form #2 will appear as shown in Figure 20.


Netscape - [HTML Service]
File Edit View Go Bookmarks Options Directory Window Help
Clear
Send
Exit from Chat Server
149 bytes read

Chat Server

```
Andy: Hi! How are you?
Vadim: I'm fine!
```

Figure 20. Chat Form in a Browser

Special Types of Forms

Creating ‘*Multipart/Mixed*’ Type Documents

This section demonstrates the use of ‘*multipart/mixed*’ type documents (documents of this type are presently supported only by Netscape 2.0 and above). An application using this document type can update the contents of a page in the client’s browser without waiting for a user request. If we want a form in our application to conform to a ‘*multipart/mixed*’ page type, we must set the *Multipart* property of the *HTMLPage* component to True. When the browser receives such a form, the connect will not be broken and the application will be able to update the form and send it to the client when appropriate. This is done through the use of the *SendMultiPage* method of the *HTMLControl* component.
In the following example (see the NN_DEMO program), a new page is sent on a timer event:

```
procedure TForm1.Timer1Timer(Sender: TObject);

begin

  {update memo form's contents}

  UpdateMemoLines;

  {send updated page}

  HTMLControl1.SendMultiPage(HTMLPage1, False);

end;
```

The second parameter in the *SendMultiPage* method denotes whether the page being sent is the last one. If so, the connect will be terminated.

Whenever this method is called for execution, the page will be sent to all the clients who are currently viewing it and did not break the connect.

You can have a page sent out not only on a timer event, but also when a certain button in the application is clicked or when the database is updated.

Use of JavaScript in Application Development

This version of the library has JavaScript support built into it. JavaScript makes it possible to noticeably extend the capabilities of the user interface in a browser. However, not all browsers support JavaScript technology. Our library includes the *TJavaScript* component, and it is there that you can place your JavaScript program code. Besides, HTML components have some properties which you can use to link certain browser events to certain JavaScript functions. These properties begin with the letters *JS* followed by the event name. The *THTMLCheckBox* component, for example, has the *JSOnClick* property, with the help of which the necessary JavaScript function can be invoked when the user clicks on that component (for instance, to have the form sent to the server, like in response to a submit-type button click event). You can specify either the function described in the *TJavaScript* component or a set of function or method calls to JavaScript objects separated by a semicolon (';') in the properties of JavaScript-related components.

In the following example (see the JAVASCR program), JavaScript is employed to have a page sent to the server when the CheckBox object is clicked and when a ComboBox item is checked. To achieve this, place the components on the project's form as shown in Figure 21.

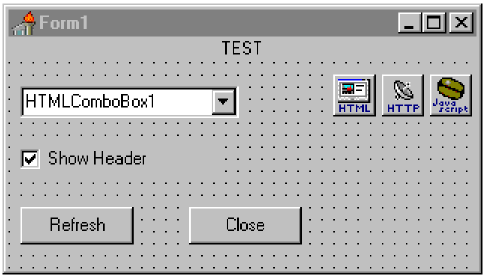


Figure 21. Form 1 in Designer

In this example, JavaScript is employed in a dual capacity, because: (a) *this.form.submit()* is explicitly specified in the *JSOnClick* property of the *ComboBox* component (case sensitive!), and (b) the name of the *DoSubmit()* function whose text is enclosed in the *TJavaScript* component is specified in the *JSOnClick* property of the *CheckBox* component.

Here is an implementation of the *DoSubmit()* function in the *JavaScript* component:

```
function DoSubmit() {

  document.Form1.submit()

}
```

The *CheckBox* component determines whether the *HTMLHeader* header will be visible in the form:

```
procedure TForm1.HTMLCheckBox1Click(Sender: TObject);

begin

  HTMLHeader1.Visible:=HTMLCheckBox1.Checked;

end;
```

The *ComboBox* component determines the contents of the *HTMLHeader* header in the form:

```
procedure TForm1.HTMLComboBox1Change(Sender: TObject);

begin

  HTMLHeader1.Caption:=HTMLComboBox1.Text;
```

```
end;
```

If you now run a program like this from a browser that supports JavaScript (Internet Explorer 3.0 or Netscape 2.0 and above), the form will change whenever the user selects one of the ComboBox items or clicks the CheckBox.

The *TJavaScript* component is employed in situations when large amounts of JavaScript text need be placed on a page. It also allows you to indicate the name of the function specified in that text in the components' properties.

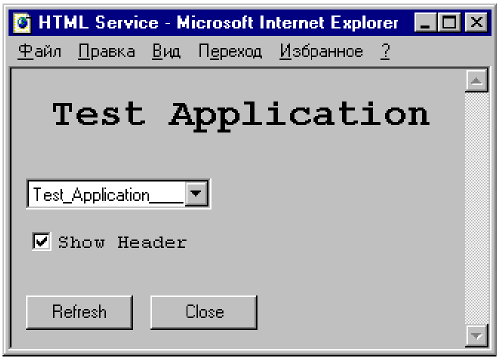

HTML Service - Microsoft Internet Explorer
Файл Правка Вид Переход Избранное ?
Test Application
Test_Application
☑ Show Header
Refresh
Close

Figure 22. View of Sample JavaScript in Browser

Updating of Form's Data

The following example illustrates the use of the *OnUpdate* event of the *HTMLPage* component. This event occurs every time a request describing a form is received from the browser (such a request is sent by the browser whenever the user clicks a Submit-type button). The event is invoked before the contents of the form's controls are updated, and can be used for preliminary verification or conversion of the user-input data.

Suppose we have a form that contains a case-sensitive *HTMLEdit* (or *HTMLDBEdit*) component, requiring that the text be input in uppercase. Obviously, we cannot MAKE the user do so by means of HTML tools alone. What we can do, however, is to convert the user-input text to uppercase before updating the corresponding data (the same goal can be achieved in another way as well, but we just give you an example here). The conversion is performed in the *OnUpdate* event of the HTMLPage component:

```
procedure TForm1.HTMLPage1Update(Sender: TObject;

  ValueList: TStringList);

begin

  ValueList.Values[HTMLEdit1.Name]:=
      UpperCase(ValueList.Values[HTMLEdit1.Name]);

  end;
```

This event can also be used for changing the text's code page (for instance, from KOI8-R to WIN1251).

Special Techniques

User Authentication

If the user authentication option has been enabled in the server's initial settings, the browser will query the user for his/her name and password when he/she first tries to access the BAIKONUR server, as shown in Figure 23:

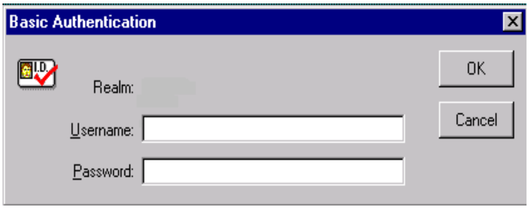


Figure 23. User Authentication Dialog

The user may enter any name and password and click "OK". BAIKONUR users are identified by their names and IP addresses. If user authentication is disabled (see "BAIKONUR Server Administrator's Guide"), the users will be distinguished only by their IP addresses. In the latter case, users with identical IP addresses may interfere with one another when working with applications. A situation like this may, for instance, occur when the users work via Proxy servers.

Verification of the username and password can be done in-program. The developer has all the necessary information on new users to decide how the program will subsequently work with a given user.

Let us consider an example whereby the program performs username and password verification when a new user tries to connect to it, and terminates if the newcomer is not a registered user.

The name, password and IP address of the current user (i.e., the user currently attempting to enter the program) are accessible in the *CurrentUser* property of the *HTMLControl* component. The list of the program's registered users is accessible in the *Users* property.

It is most appropriate to perform verification of the user-supplied password at the time a new user is attempting to log in to the program, i.e. in the *OnNewUser* event handler of the *HTMLControl* component:

```
procedure TForm1.HTMLControl1NewUser(Sender: TObject;

  UserID: Integer);
```

```
begin
  with HTMLControl1.CurrentUser do begin
    {are parameters valid?}
    if NotAllowed(Name, Password, IPAddr) then
      {if no, close user}
      HTMLControl1.UserClose;
  end;
end;
```

The *NotAllowed* function in this case should return True if the user attempts to log in with an invalid username/password combination or from an illegal IP address. In our case, the user will receive the program's final screen, although we could also arrange for a special custom-message screen form to be posted to such a user instead of executing the *UserClose* event.

IMPORTANT. Even if the user authentication option is not enabled in the server's settings, you can still make the application tell the browser that it must query the user for his/her name and password. To do so, use the OnReceive event and the SendResponse method of the HTMLControl component:

```
procedure TForm1.HTMLControl1Receive(Sender: TObject;
var Form: TForm;

  UserID: Integer; var Data: string; var Action:
TReceiveAction);

begin
  {if username and password are not supplied, then}
  if (HTMLControl1.CurrentUser.Name='')and
     (HTMLControl1.CurrentUser.Password='') then begin
    {cancel further processing of request}
    Action:=raCancel;
    {return 'unauthorized access' resonse}
   HTMLControl1.SendResponse(HTMLControl1.CurrentUser.ID,
      "401 Unauthorized",'WWW-Authenticate:
Basic','','');

    HTMLControl1.DeleteUserByID(UserID);

  end;

end;
```

As a result of execution of the above code the browser will query the user for his/her name and password, and then send out a fresh request.

Requesting a Resource

The following example demonstrates the use of the *OnResource* event of the *HTMLPage* component. This event occurs every time the browser requests the program for a resource the reference to which is placed on the form (for example, as *demo.exe?ImageRes1*).

A reference like this will appear on the form if the *THTMLImageRes*, *HTMLImageButton* or *HTMLDBGIF* component is used. The browser will then be able to request this resource automatically, and the program will send in the requested image. In this case, the developer does not need to interfere with the image send/receive mechanism. However, you can utilize that mechanism for your specific needs. You can, for example, place the *HTMLLabel* component on the form and specify the URL as *demo.exe?Some_Resource_ID.* Once the user clicks on such a reference in the browser, the application will receive a resource request, which will invoke the *OnResource* event handler of the *HTMLPage* component. Since the application doesn't "know" what resource the browser actually needs, the developer must in this case write such an event handler that would generate and send the requisite resource (which can be of any type – a GIF or JPEG image, plain text, an HTML page or a file of any type). If the requested resource is not sent, the browser will remain in the wait mode.

The LINKFORM example shows how the user can go to another form in the application by clicking on a resource reference rather than on a button. To achieve this, we must place the *HTMLLabel* component on form #1 and specify the URL as *linkform.exe?GotoForm2*. Then, we must write the *OnResource* event handler of the *HTMLPage1* component on form #1:

```
procedure TForm1.HTMLPage1Resource(Sender: TObject;
ResName: string;

  UserID: Integer; var ResourceWasSent: Boolean);

var

  s: string;
```

```
begin

  {what resource is requested?}

  if ResName='GotoForm2' then begin

    {set flag if response was sent}

    ResourceWasSent:=True;

    {get HTML image of Form 2}

    s:=Form2.HTMLPage1.HTMLPageScript;

    {send HTML script to user}

    HTMLControl1.SendResource(UserID, @s[1], Length(s),
"text/html");

    {'take' user to Form 2}

    HTMLControl1.CurrentUser.ActiveForm:=Form2;

  end;

end;
```

You can employ this method for solving such tasks as sending the selected file to the user, navigating through the HTML pages of a database, etc.

Browser's "Back" Button and Page Caching

Most browsers save the HTML pages which the user is browsing through in a special RAM area or on the hard disk (in cache memory). Such browsers allow the user to go back to previous pages simply by clicking the "Back" button. The necessary HTML page is then retrieved from the cache memory rather than requested from the Web server anew. This capability turns out to be very useful when static pages are involved, but can well lead to incorrect results if the user works with an application. Suppose that the user works with a program that allows him/her to navigate through the records of a database table and modify (edit) its fields. If he/she clicks the "Back" button and goes back to one of the previously viewed pages, the program's actual status (i.e., the table's current record) may no longer correspond to what the user sees on the screen. If the user now edits the record and saves it, the new data might not be saved in the record it was expected to.

If the application consists of more than one form and the user clicks the "Back" button to go back to one of the previous forms, he/she may cause its data to be sent to the actual form, which likewise may result in an incorrect execution of the program.

Several solutions can be suggested to guarantee correct execution of the program which the user works with from a browser.

The first solution (implemented in the BAIKONUR components library) consists in verifying whether the page sent from the browser to the application is actual. Every time a page describing a certain form of the application is sent to the user, it is automatically assigned a unique number in a hidden field (the *hidden* tag). Once the user fills in the page input fields and sends it back to the application (for example, by clicking a Submit-type button on the form), the program verifies that unique number. If it does not match the number of the most recently sent page, the user receives a message stating that the page just sent is not actual (<Not actual form or timeout. Don’t use “Back” button>). After the user clicks the button on the error-message form, he/she receives the actual form the application. This is the default behavior of the application. However, you can change it if you set the *CheckFrame* property of the *HTMLPage* component to False. With this second solution, no page actuality verification will be performed, and the developer will himself have see to it that the above-described situation would not lead to incorrect program execution. We shall discuss how this solution can be implemented later in the text.

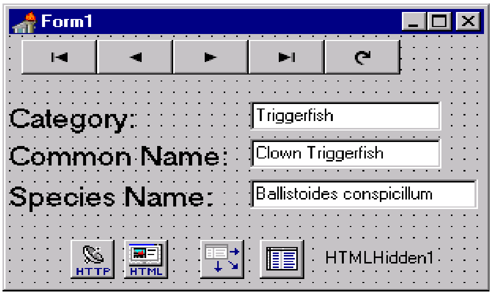
The third solution is to prohibit the browser from caching the application’s pages. This can be achieved by adding a “Cache-Control: no-cache” string to the *HTTPAdd* property of the *HTMLControl* component. In response to a click on its “Back” button the browser will then display a message stating that no data is available in the cache memory and the user will have to access the application again for the current form’s image (by clicking the “Reload” button). However, such a solution is not always convenient.

Let us therefore go back and see how the second solution (allowing the user to click the browser’s “Back” button without causing incorrect program execution) can be implemented.

First, we must disable the form actuality verification feature by setting the *CheckFrame* property of the *HTMLPage* component to False.

Second, we must keep a track of what kind of form has been sent in and act accordingly. Indeed, there is a multitude of forms which require no additional actions to be taken to ensure correct program execution without actuality verification. Forms containing nothing but text and buttons or forms serving only for adding a record to a database table (registration forms) are the obvious examples.

The most typical case requiring the developer to care about correct program execution is an application that allows the user to view and modify (edit) certain table records. In the way of an example, let us try and develop a single-form application designed to display and edit records of the BIOLIFE table (see the BACKBTN example). To do so, we must place on the new project’s form the *THTMLControl*, *THTMLPage*, *TTable*, *TDataSource*, *THTMLNavigator* and *THTMLHidden* components and the *THTMLDBEdit* and *THTMLLabel* components (three of each), and tune these components to the BIOLIFE table from the DBDEMOS database. At design time, the project will appear as follows:



Category: Triggerfish
Common Name: Clown Triggerfish
Species Name: Ballistoides conspicillum

Figure 24. Project at Design Time

Next, we must set the *CheckFrame* property of the *HTMLPage* component to False. For the program to execute correctly, we must ensure that no record other than the one sent in by the user will be modified. To achieve this, we must not rely on the current record, but rather identify the requisite record by its primary key (which will be stored on the form in the *Hidden* field).

Whenever the form is sent out, we should save the value of its primary key in the *HTMLHidden1* component; when data is received from the user, we must go to the requisite record based on its primary key.

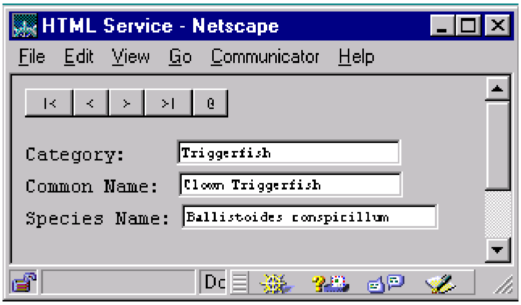
The key value should be saved in the *OnScript* event handler of the *HTMLPage* component:

```
procedure TForm1.HTMLPage1Script(Sender: TObject);

  begin

    HTMLHidden1.Caption:=Table1.FieldByName(“Species
  No”).AsString;

  end;
```

We can use the *OnUpdate* event handler of the *HTMLPage* component to actually go to the requisite record:

```
  procedure TForm1.HTMLPage1Update(Sender: TObject;
  ValueList: TStringList);

  begin

    Table1.Locate(“Species No”,
  ValueList.Values[“HTMLHidden1”], []);

  end;
```

We can now compile our program and call it for execution from a browser, where it will appear like this:



Category: Triggerfish
Common Name: Clown Triggerfish
Species Name: Ballistoides conspicillum

Figure 25. Form’s View in Browser

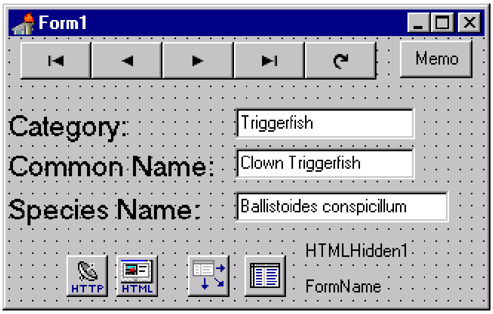
While working in this program, the user may browse to the middle of the table with the aid of the navigator buttons, click the “Back” button to go several records back, edit a record field and then click the navigator’s corresponding button to have the data saved. The program is guaranteed to save the data for the very record that the user sees in his/her browser.

Of course, the developer of a real program must take other factors into account as well (such as, for example, the possibility of another user deleting a certain record).

A multi-form application is a somewhat more complicated case. If the application logic is such that the user is allowed to navigate to one of the available forms, input certain data there and press a “save” button, then he/she must be able to do so while working in a browser as well. However, the developer should in that case keep track of what form the incoming data relates to, and post an error message to the user if an exceptional situation occurs (for instance, if data for an already deleted form is received).

To illustrate the above-said, let us make our previous sample program a bit more complicated by adding a second form to it. Suppose we want this second form to contain a memo field for the database records and be invoked by the click of a button on the first form.

To achieve this, we must add the *THTMLButton* and *THTMLHidden* components to the project’s first form, specify “Memo” in the button’s *Caption* property, and assign “FormName” value to the *Name* property of the *HTMLHidden* component. At design time, the first form of our project will appear as follows:



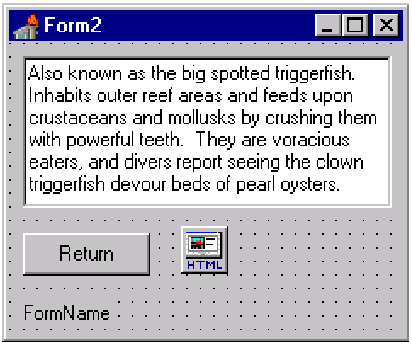
Category: Triggerfish
Common Name: Clown Triggerfish
Species Name: Ballistoides conspicillum

Figure 26. Modified Form 1

Clicking the “Memo” button will invoke the second form:

```
procedure TForm1.HTMLButton1Click(Sender: TObject);

begin

  Form2.Show;

end;
```

On the second form we shall place the *THTMLPage*, *THTMLDBMemo*, *THTMLButton* and *THTMLHidden* components:



Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with powerful teeth. They are voracious eaters, and divers report seeing the clown triggerfish devour beds of pearl oysters.

Figure 27. Form 2 of the Project

We must also set the *CheckFrame* property of the *HTMLPage1* component to False, change the name of the *HTMLHidden* component to FormName, tune *HTMLDBMemo1* to the table’s NOTES field, and write a button click (“go back to first form”) event handler:

```
procedure TForm2.HTMLButton1Click(Sender: TObject);

begin

  Close;

  Form1.SetFocus;

end;
```

Next, we must implement the basic idea of our sample program, i.e. verify the name of the form for which data has been received from the browser, and take the current user to that form.

At this point, it would be appropriate to recess briefly and recall how a multiuser application “learns” which form a given user is currently working with. When a new user gets connected to the program a record describing that user's parameters is created (whether it is a single-user or a multi-user application, the mechanism remains basically the same). Access to these parameters can be gained via the *CurrentUser* or *Users* properties of the *HTMLControl* component. The parameters include a pointer to the instance of the current form opened for the user (ActiveForm). It is this very parameter that we can employ to keep track of whether the current form for the user is changed. Besides, the current form for the given user is accessible in the *HTMLControl* component's event handlers.

Returning to our example, we must see to it that information on the form's name is present in the page sent to the client (browser). To achieve this, we use the *THTMLHidden* component named FormName on both of our forms. At first glance, we could make the name assignment (i.e., assign the form's name to the *Caption* property) at design time. However, if an instance of the form is created at run time, the name of that form will be determined dynamically, and might not coincide with what had been specified at design time. Therefore in our case the form name assignment will take place in the *OnScript* event handler of each form's *HTMLPage* components. For the first form:

```
procedure TForm1.HTMLPage1Script(Sender: TObject);

begin

  HTMLHidden1.Caption:=Table1.FieldByName("Species
No").AsString;

  FormName.Caption:=Name;  // a new line

end;
```

For the second form:

```
procedure TForm2.HTMLPage1Script(Sender: TObject);

begin

  FormName.Caption:=Name;

end;
```

At run time, the program will send forms along with information on their names to the browser, and receive data from the browser along with the name of the destination form for that data. Specifically, the data will contain a *FormName*=Form1 or *FormName*=Form2 substring.

We should use the *OnReceive* event handler of the *HTMLControl* component for verifying the form's name and making a transition (and re-directing the data) to another form in:

```
procedure TForm1.HTMLControl1Receive(Sender: TObject;
var Form: TForm;
  UserID: Integer; var Data: string; var Action:
TReceiveAction);
var
  s: string;
  i: Integer;
begin
  {does form has a name?}
  if Pos(“FormName=”,Data)<>0 then begin
    {get form’s name}
    s:=’’;
    i:=Pos(“FormName=”,Data)+9;
    while (i<=Length(Data))and(Data[i]<>’&’)and
          (Data[i]<>#13)and(data[i]<>#10) do begin
      s:=s+Data[i];
      Inc(i);
    end;
    {find form by its name}
    if Form.Name<>s then
      for i:=0 to Screen.FormCount-1 do
        if Screen.Forms[i].Name=s then begin
          Form:=Screen.Forms[i];
          Break;
```

```
            end;

      end;

    end;
```

The parameters passed to the event handler include the *Form* parameter, which identifies the active form for the user. We can change this parameter, thereby causing the data to be re-directed to another form. The string-type *Data* parameter contains a description of the form (with filled-in fields) sent by the browser. We parse this string to extract the form's name and, if it fails to match the name of the previous form for the given user, search for the necessary form among all other forms available in the application (for which the *Screen* object is used).

A more detailed description of the *OnReceive* event handler of the *HTMLControl* component can be found in Annex A.

Anyone working with the above-described sample application from a browser can use its "Back" and "Forward" buttons without any hesitation. Even if the application has been closed or unloaded on timeout, it will still execute correctly and will produce no error messages.

An application may contain both the forms that do require verification of the sent-in data actuality, and the forms which do not require it.

Caching of Images

While processing HTML pages, browsers are also caching the images (GIF and JPEG) which they encounter in these pages. Hence when a user browses through different pages that refer to one and the same image, he/she actually sees the image already stored in the browser's cache memory. In situations involving static images such a feature proves useful, and this has to be taken into account when developing BAIKONUR applications.

The BAIKONUR HTML Controls library currently includes four components that operate with images: *HTMLImageButton*, *HTMLImageRef*, *HTMLImageRes* and *HTMLDBGIF*.

As far as the *HTMLImageRef* object is concerned, its logic is fairly straightforward - it merely places a reference to a static image (a file stored on the disk) on the HTML page.

Other objects place a reference like <img src="project1.exe?HTMLImageRes1"> to a program's resource on the HTML page. When it first accesses the application, the browser will query the latter for that image, and will retrieve it from the cache memory thereafter. Even if the image in the program changes, the browser will have no way of "knowing" it, and will still show the old image on a new page. For the browser to display a new image on the page, the reference should be changed accordingly. For some library components, this is done automatically.

An image reference is generated by the HTML Controls library according to the following rule. The reference opens with the program's name (for example, *project1.exe*). Next goes the name of the component (HTMLImageRes1), followed (for the *HTMLImageRes* and *HTMLDBGIF* components) by a (optionally variable) character string.

Let us see how the image for various components can be updated.

In the case of the *HTMLImageRef* component, we must change the reference to the image file if we want to update the onscreen image (i.e., we must place the new image into a new file and assign the name of that new file to the component's *FileName* property).

In the case of *HTMLImageRes* and *HTMLImageButton* components, we must change the *ID* property; as a result, the reference on the page will change, making the browser request a new picture.

The *HTMLDBGIF* component is a somewhat special case, for a set of *ID* and *IDType* properties can be specified for it. By assigning different values to the *IDType* property, we can make the browser request a picture every time, or every time a jump to a new record is made, or just when the *ID* property changes its value.

If you look attentively at the rule according to which a resource reference is generated, you will see that the form's name is not part of that reference. One implication of this is that two different components (for example, *HTMLImageRes* with different images) with identical references might turn out to be located on different forms of the application. In a situation like that the browser will display one and the same image (namely, the one received first) on those pages. To preclude such a situation, we must assign different names to the image-displaying components, or else provide unique values for their *ID* properties. Conversely, if components designed to display one and the same image are to be present on different forms of the application, we must see to it that these components have matching names - this will help the browser work somewhat faster.

HTMLTag Example

Using the *HTMLTag* component, you can place on a form the text of those HTML tags that are not represented in the HTML components library. In our HTMLTAG example, we employ this technique to place the "marquee" (scrolling text) tag on the form – a feature supported only by Microsoft's Internet Explorer.

At design time, our program's main form will appear as follows:

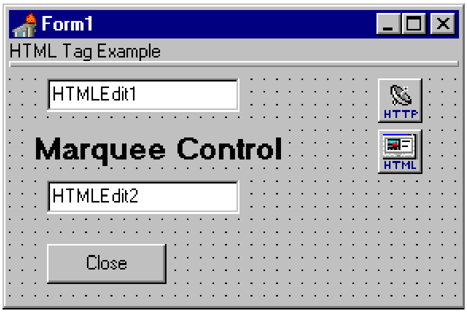
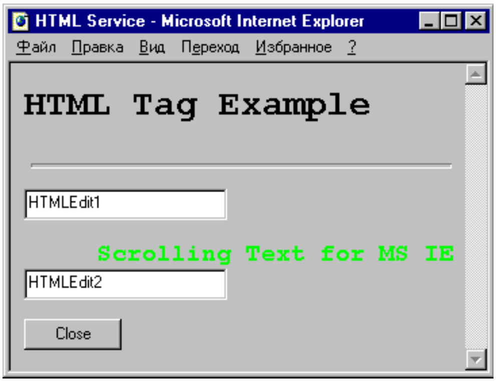


Figure 28. Form 1 at Design Time

The *HTMLTag* component has a *Caption* property in which "Marquee Control" is specified. The *Script* property contains the text for the *Marquee* tag:

```
<b><font size=5 color=#00FF00><marquee> Scrolling Text for MS IE! </marquee></
font></b>
```

In Microsoft's Internet Explorer, the form will appear as follows:


HTML Service - Microsoft Internet Explorer

HTML Tag Example

HTMLEdit1

Scrolling Text for MS IE

HTMLEdit2

Figure 29. Marquee in IE Browser

Passing of Data From Client To Server (*HTMLPutFile*)

Originally, the HTML standard was conceived as a tool for the passing of resources only from the server on the user request. But with the advent of application software capable of executing under server control it became necessary to organize the flow of data in the opposite direction as well. Partially, this problem can be solved in the HTML through the use of forms and input/edit fields like *Edit* and *Memo*. However, this technique is applicable basically to the passing of text, and then only limited amounts of it. The situation becomes much more difficult when binary data is to be transmitted.

At this time, only the browsers marketed by Netscape (2.0 and above) and Microsoft (3.02 with patches and above) can be used to transmit binary files of any type and of virtually any size to the server. These browsers support a special document type known as "multipart/form-data" and a type *File* input tag. This tag is displayed in a browser as an input field with a "Browse" button. This enables the user to select a file on his/her computer and then send it to the server by clicking a Submit-type button on the screen form. BAIKONUR HTML Controls library also supports the "multipart/form-data" document type and the *File* tag.

Let us consider an example whereby the user must fill in a registration form and send his/her photo. The user-originated data is to be saved in a table.

Place the components on the form as shown in Figure 30. Tune the table and the *HTMLDBEdit* components. The table must include the *FIO*, *Email* and *Photo* fields of the BLOB type.

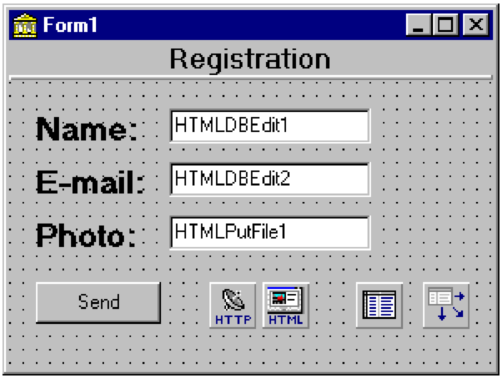


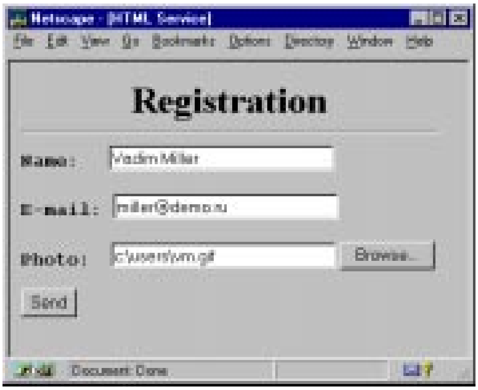
Figure 30. View of Project at Design Time

Next, you must set the *MaxSize* property of the *HTMLPutFile1* component to 50 (implying that it will be impossible to send a file larger than 50K bytes) and specify its *DstFileName* property, for example as 'Photo.img' (which means that the data will be temporarily saved into a file named PHOTO.IMG).

Write the button click event handler:

```
procedure TForm1.HTMLButton1Click(Sender: TObject);

begin

  if Table1.State<>dsBrowse then Table1.Edit;

  Table1Photo.LoadFromFile(“photo.img”);

  Table1.Post;

end;
```

The project is ready. In a browser, the filled-in form will appear like this:


Netscape - [HTML Service]
File Edit View Go Bookmarks Options Directory Window Help
Browse...
Send
Document: Done

Registration

Name: Vadim Miller

E-mail: miller@demo.ru

Photo: c:\users\vm.gif

Figure 31. View of Form in Browser

If the user clicks the "Send" button, the browser will generate the appropriate data (which will include the above-mentioned file), and send it to the program. The *HTMLPutFile* component will place the image into a temporary file on the server's hard disk, wherefrom it can be read and copied into a type BLOB field of the table. Besides, you can use the *OnFileReceive* event of the *HTMLPutFile* component to fill the BLOB field.

There exist other techniques of passing files to the server as well, but special programs that execute on the client's computer need be developed to implement these. You can do it with the help of the *HTTPClient* component, which is part of the BAIKONUR Intranet Suite delivery package. A disadvantage of this approach is that these programs will work under Windows only. And, of course, you will have to make sure that the programs are indeed installed on the client's machine.

A Non-HTML System

It is sometimes necessary to develop a system whereby the client's application runs as a stand-alone software and requests data from a remote application on a time-to-time basis only. For example, BAIKONUR Web Server and a special server-based application can be used in the capacity of a data replication server or a report printing server. In that case, there is no need to develop a fully-fledged server application (an Web application) with a user interface – the functions of a client can be performed by any application that "knows" how to send out a request and receive a response.

Using the *HTMLControl* component, you can develop an application that will not use the HTML standard to service a client. The functions of a client in this case are performed by an application that uses the HTTP protocol to communicate with the BAIKONUR server, so that the request and response data can be in any (including HTML) format. To organize the passing of data to/from the server, use can be made of the *HTTPClient* component. This component is a standard part of BAIKONURstarting from its Enterprise version. In the version under discussion, the developer has to care about the exchanged data format himself.

Let us discuss a simple example of a Delphi client application that accesses the server application to receive from it a list of fish species names from the BIOLIFE table and, when the user selects a specific kind of fish, display a picture of it on the screen (see the GetFish example).

First, you must build a server application. Place the *THTMLControl* and *TTable* components on the form, and tune *TTable* to the BIOLIFE.DB table from the DBDEMOS alias. For the application to be able to respond to the client's requests, you must process the *OnClick* event of the *THTMLControl* component:

```
procedure TForm2.HTMLControl1Receive(Sender: TObject;
var Form: TForm;

  UserID: Integer; var Data: string; var Action:
TReceiveAction);

var

  s: string;

  BS: TBLOBStream;

  Buff: PBuff;

begin



  {is it a GetFishNames request?}

  if Pos("GetFishNames", Data)<>0 then begin

    Action:=raCancel;

    {generate response - a string with species names}

    Table1.First;

    s:='';

    repeat

      s := s + Table1.FieldByName("Species
Name").AsString + #13#10;

      Table1.Next;

    until Table1.EOF;
  {send out response}
  HTMLControl1.SendResource(UserID, @s[1], Length(s),
“text/plain”);
 end;


 {is it a GetFishImage request?}
 if Pos(“GetFishImage=”, Data)<>0 then begin

  s:=UnEscapeText(Data);
  s:=Copy(s, Pos(“«“, Data)+1, Length(s)-Pos(“«“,
Data));
  s:=Copy(s, 1, Pos(“«“, s)-1);

  Table1.First;
  {generate response - species image}
  repeat
   if Table1.FieldByName(“Species Name”).AsString=s
then begin
    Action:=raCancel;
   BS:=TBLOBStream.Create(Table1.FieldByName(“Graphic”)
as TBLOBField, bmRead);
    GetMem(Buff, BS.Size);
    try
     BS.Read(Buff^, BS.Size);
     {send out response}
     HTMLControl1.SendResource(UserID, @Buff^[8],
BS.Size-8, “image/bitmap”);
```

```
        finally
          FreeMem(Buff, BS.Size);
          BS.Free;
        end;
        Break;
      end;
      Table1.Next;
    until Table1.EOF;
  end;
end;
```

Your server application is ready. You can now try it out by issuing a request from the browser, specifying the URL as:
http://web_serv/project1.exe?GetFishNames
The browser should display a list of fish names.
Next, you must build a client application. This application will use the *HTTPClient* component supplied with the *HTMLControls* library starting from BAIKONUR Enterprise version. You could, of course, do without that component, but then you would have to know how to work with sockets and send data in HTTP 1.0 format.
Figure 32 shows how the client application's first form will appear at design time.
Do not forget to specify the IP address of the server, port number, and name of the program on the server (FISHSERV.EXE) in the *HTTPClient* component's properties.

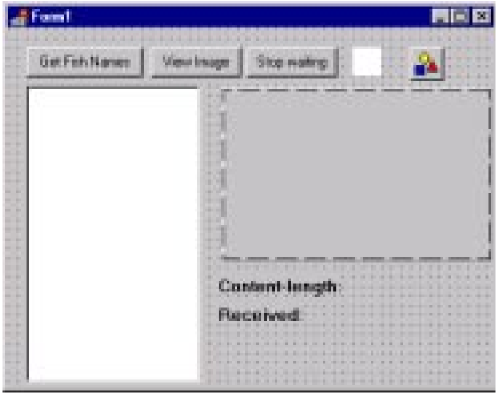


Figure 32. Client Application Form at Design Time

Write the “Get Fish Names” button click event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  with HTTPClient1 do
    {send request}
    if SendData(“GetFishNames”) then begin
      {analyze received data}
      s:=ReceivedData;
      {fill in fish names list}
      ListBox1.Clear;
      while Pos(#13#10, s)<>0 do begin
        ListBox1.Items.Add(Copy(s,1,Pos(#13#10, s)-1));
        Delete(s,1,Pos(#13#10, s)+1);
      end;
    end;
end;
```

Write the “View Image” button click event handler:

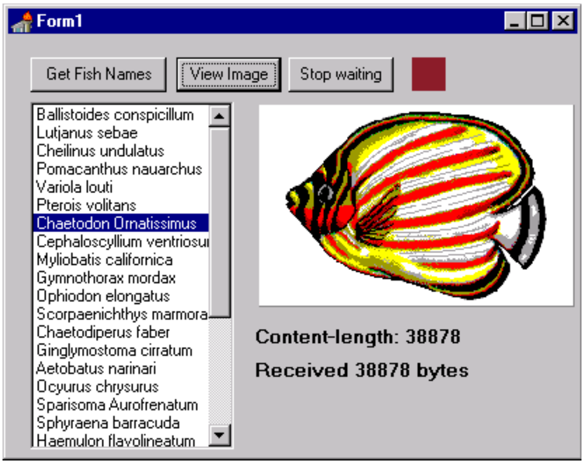
```
procedure TForm1.Button3Click(Sender: TObject);
var
  F: File;
  s: string;
begin
```

```
    if ListBox1.ItemIndex>=0 then begin
      {generate request}
     s:='GetFishImage=»'+ListBox1.Items[ListBox1.ItemIndex]+'»';
      {send request}
      if HTTPClient1.SendData(s) then begin
        {received data ...}
        s:=HTTPClient1.ReceivedData;
        {... save into a temporary file}
        AssignFile(F, “temp.bmp”);
        Rewrite(F, 1);
        BlockWrite(F, s[1], Length(s));
        CloseFile(F);
        {display received picrure}
        Image1.Picture.LoadFromFile(“temp.bmp”);
      end;
    end;
  end;
```

Write the “Stop waiting” button click event handler:

```
  procedure TForm1.Button2Click(Sender: TObject);
  begin
    HTTPClient1.CancelOperation;
  end;
```

At run time, the client application’s form will appear like this:



Ballistoides conspicillum
Lutjanus sebae
Cheilinus undulatus
Pomacanthus nauarchus
Variola louti
Pterois volitans
Chaetodon Ornatissimus
Cephaloscyllium ventriosu
Myliobatis californica
Gymnothorax mordax
Ophiodon elongatus
Scorpaenichthys marmora
Chaetodiperus faber
Ginglymostoma cirratum
Aetobatus narinari
Ocyurus chrysurus
Sparisoma Aurofrenatum
Sphyraena barracuda
Haemulon flavolineatum

Content-length: 38878

Received 38878 bytes

Figure 33. Client Application Form at Run Time

Chapter III

This Chapter discusses the ways of creating custom HTML components that can be used for developing BAIKONUR applications.

Creating Custom Components

The principal function of the *HTMLPage* component is to generate an HTML script describing a given form. The page is created by polling the HTML components parented by that form and representing them as an HTML script. This is done by calling the *htmlScript* method of each component's parent (if any). In situations when the component's parent has no *htmlScript* method, the method of the component itself is called.
There are several kinds of HTML components:

- components designed only to display information (for example, *HTMLLabel*, *HTMLHeader* or *HTMLRuler*)
- components allowing information to be input, modified (edited) and displayed (for example, *HTMLEdit* or *HTMLCheckBox*)
- components allowing images to be displayed on a form (for example, *HTMLImageRes*)
- components allowing resource references to be placed on an HTML page (for example, *HTMLDBGIF*)
- components representing Submit-type buttons (for example, *HTMLButton* or *HTMLImageButton*)
- component for placing text in the header of an HTML page (for example, JavaScript)

There are complex components, for example, HTMLImageButton or HTMLDBGrid, which contain images, response on Submit and place JavaScript on a page.

Requirements to Custom HTML Components

The process of creating a custom HTML component is practically no different from that of creating a conventional component. Custom HTML components should met the following requirements:

I. EVERY component should have in its PUBLISHED section a htmlScript function declared as follows:

function htmlScipt : **string**;

This function should returm a string containing the description of an object in the HTML format. For example, the *HTMLButton* object can return a string like “<input type=submit name=»HTMLButton1" value=» OK «>’

To generate a string, use can be made of the set of functions of the *HTMLWriter* object (from the *HTMLOpen* module) existing when the *htmlScipt* function is called. Example Syntax:

```
THTMLLabel = class(TCustomLabel)

private

  FDistance : Integer;

  FPreformat : Boolean;

public

  constructor Create(AOwner: TComponent); override;

published

  property Align;

property AutoSize default False;

  function htmlScript:string;

  property Caption;

  property Color;

  property Distance : Integer read FDistance

                             write FDistance;

  property Font;

  property Preformat : Boolean read FPreformat

                              write FPreformat

                              default True;
  property Visible;
end;
...
constructor THTMLLabel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Preformat:=True;
end;

function THTMLLabel.htmlScript:string;
begin
  Result:='';
  if Not Visible then Exit;
  if Not Preformat then
    Result:=Result+HTMLWriter.TextEffectEnd(efPreformat);
  Result:=Result+StartFontDef(Font, HTMLWriter);
  Result:=Result+Space(Distance);
  if Preformat then
    Result:=Result+HTMLWriter.EscapeText(Caption)
  else
    Result:=Result+Caption;
  Result:=Result+EndFontDef(Font, HTMLWriter);
  if Not Preformat then
   Result:=Result+HTMLWriter.TextEffectStart(efPreformat);
```

```
end;
```

II. If the object represents an input field (like Edit, ListBox, Memo, etc.), then the *htmlUpdate* procedure should be declared in the component's PUBLISHED section:

procedure *htmlUpdate(RList : TStringList);*
The procedure is passed a list of parameters arriving from the client when he/she clicks a Submit-type button. The parameters describe the form's status (i.e., contents of the input fields), and are usually specified as <OBJECT NAME>=<NEW VALUE>. You use the *htmlUpdate* procedure to select the parameter corresponding to an object from that list, and update the object's status accordingly.
Example Syntax:
The *HTMLEdit1* object of the *THTMLEdit* type generates the following string in the *htmlScript* function:
"<input type=text name=»HTMLEdit1" value=»Some Text» size=15>'
The *Rlist* will return something like this in one of the strings:
HTMLEdit1=New text
The *htmlUpdate* procedure executes the following for the *THTMLEdit* object:

```
procedure THTMLEdit.htmlUpdate(RList : TStringList);

begin

  if Visible then

    Text:=RList.Values[Name];

end;
```

Example Syntax:

```
THTMLCheckBox = class(TCustomCheckBox)

private

  FDistance : Integer;

public

  constructor Create(AOwner: TComponent); override;

published

  function htmlScript:string;
  procedure htmlUpdate(RList : TStringList);
  property Align;
  property Caption;
  property Checked;
  property Distance : Integer read FDistance
                      write FDistance;
  property Font;
  property Visible;
  property OnClick;
end;
...
function THTMLCheckBox.htmlScript : string;
begin
  Result:='';
  if Not Visible then Exit;
  Result:=StartFontDef(Font, HTMLWriter);
  Result:=Result+Space(Distance);
  Result:=Result+HTMLWriter.CheckboxField(Name, Name,
    Checked);
  Result:=Result+HTMLWriter.EscapeText(Caption);
  Result:=Result+EndFontDef(Font, HTMLWriter);
end;

procedure THTMLCheckBox.htmlUpdate(RList : TStringList);
```

```
begin
  if Visible then
    Checked:=RList.Values[Name]<>'';
end;
```

III. If the component represents a Submit-type button, it must include the *htmlClick* procedure in its PUBLISHED section:

procedure htmlClick(RList:TStringList);

The procedure is used to check whether the given button was clicked:

```
if RList.Values[Name]<>'' then Click;
```

Example Syntax:

```
THTMLButton = class(TButton)
private
  FButtonType : TButtonType;
  FDistance : Integer;
published
  function htmlScript:string;
  procedure htmlClick(RList : TStringList);
  property Align;
  property Distance : Integer read FDistance write
FDistance;
  property ButtonType : TButtonType read FButtonType
write FButtonType;
end;
...
function THTMLButton.htmlScript:string;
```

```
begin

  Result:='';

  if Not Visible then Exit;

  Result:=Space(Distance);

  if ButtonType=btSubmit then

    Result:=Result+HTMLWriter.SubmitField(Name,

      HTMLWriter.EscapeText(Caption))

  else

    Result := Result +
HTMLWriter.ResetField(HTMLWriter.EscapeText(Caption));

end;



procedure THTMLButton.htmlClick(RList : TStringList);

begin

  if RList.Values[Name]<>'' then

    Click;

end;
```

IV. The component can place in the HTML script a reference to the resource which the program must send to the client on a request from the browser. A resource reference can be of two types:

1. Reference to a physical file that must be sent on the browser's request without the program's participation. It may, for instance, be a HTML-page reference generated with the help of the *HTMLLabel* component (the server will send this page will to the user when he/she clicks on the reference in the browser). Or it may be a GIF-image reference generated with the help of the *HTMLImageRef* component. If the appropriate option is checked, the referenced image will be automatically requested by the browser and send by the server without the program's participation. A reference of this type will have the following syntax:

   «http://w_serv/images/dog.gif»

2. Reference to a resource stored in-program. Such a reference may, for example, have a syntax like http://w_serv/demo.exe?IR1. In this case, the browser-originated request will be addressed to the DEMO.EXE program, and it is this program that will have to send the requested resource to the browser. One example of such a component is *HTMLImageRes*. Besides, you can also create a custom component that would represent the HTML pages stored in a database to the client.

   If the component places into an HTML script a resource reference (for example, a GIF image) of the second type which must be send by the program to the client on the browser's request, the PUBLISHED section of that component should include the *htmlResource* function of the following syntax:

function htmlResource(s: **string**): boolean;

Here, the S string is used to pass the name of the resource to be sent in. If this name points to a resource containing the requested object, the function should form a buffer to accommodate the data, call the *SendHTMLResource* procedure of the *HTMLOPEN* module, and return True.

Example Syntax:

Object IR1 of type *THTMLImageRes* generates a string of the following pattern:

“<img src=»http://W_SERV/demo.exe?IR1" border=0 alt=»IR1">’

The *htmlResource* function generates a GIF-image buffer and call the “send” procedure:

```
...
 if Name=s then
 ...
   SendHTMLResource(Buffer, Size, MIME_GIF);
   Result:=True;
 ...
```

Example Syntax:

The *THTMLImageRes* component is a descendant of the *TCustomHTMLImageRes* class that encapsulates a *Buf* field containing a GIF image and a *BufSize* field specifying the buffer size.

```
THTMLImageRes = class(TCustomHTMLImageRes)
private
  FID : string;
published
  function htmlScript:string; override;
  function htmlResource(s:string): boolean;
  property AltName;
  property ID : string read FID write FID;
  property URL;
end;
. . .
function THTMLImageRes.htmlScript:string;
begin
  Result:=’’;
  if Not Visible then Exit;
  Result:=Space(Distance);
  if URL=’’ then
   Result:=Result+HTMLWriter.Image(BaseURLStr+’?’+Name+ID,
      AltName, AlignImage)
  else
    Result:=Result+HTMLWriter.Link(URL, “”,
      HTMLWriter.Image(BaseURLStr+’?’+Name+ID, AltName,
      AlignImage));
end;

function THTMLImageRes.htmlResource(s:string): boolean;
begin
  Result:=False;
  if s=Name+ID then begin
```

```
    SendHTMLResource(Buf, BufSize, “image/gif”);

    Result:=True;

  end;

end;
```

V. It is sometimes necessary create a custom component that would place some text into the HTML script’s header. To be able to do so, the component should contain a declaration of the *htmlHeader* function in its PUBLISHED section:

function htmlHeader: **string**;

This function must return the text to be placed into the header.

Example Syntax:

The *TJavaScript* component places the following Java-script text into the script header:

```
TJavaScript = class(TComponent)

  private

    FScript: TStringList;

procedure SetScript(Value: TStringList);

  public

    constructor Create(AOwner: TComponent); override;

    destructor Destroy; override;

  published

    function htmlHeader: string;

    function htmlResource(s:string): Boolean;

    property Script: TStringList read FScript write
SetScript;

end;

. . .
```

```
function TJavaScript.htmlHeader: string;

var

  i: Integer;

begin

  if FScriptPlace=spPage then begin

    Result:=#13#10'<SCRIPT
LANGUAGE=»JavaScript»>’#13#10'<!— Java Script’#13#10;

    for i:=0 to FScript.Count-1 do

      Result:=Result+FScript.Strings[i]+#13#10;

    Result:=Result+’// end of Java Script !—>’#13#10'</
SCRIPT>’#13#10;

  end

  else

    Result:=#13#10'<SCRIPT LANGUAGE=»JavaScript» SRC=»’
+ BaseURLStr + “?” + Name

            + “«>’#13#10 + “</SCRIPT>”#13#10;

end;
```

Auxiliary Modules

HTMLOPEN

When creating a custom component, you should generate an HTML tag for that component in the *htmlScript* method. To achieve this, you can (although not necessarily) use the *HTMLWriter* object that exists when the given method is called and is declared in the given module.

The same module also contains the *SendHTMLResource* procedure, which is employed to send the requested HTML resource from the component’s *htmlResource* procedure or from the *OnResource* event handler of the *HTMLPage* component.

You can use the *TMIMEType* type and the corresponding array of *MIMETable* strings containing names of the MIME types when calling the *SendHTMLResource* procedure, and adapt it to your specific needs.

HTMLWRTR

This module contains the *THTMLWriter* class, which encapsulates methods for generating HTML tags. An *HTMLWriter* object becomes accessible whenever the *htmlScript* method is called. The functions that these methods perform are self-explanatory from their names. For instance, to obtain a tag describing, say, a reference, you must use the *Link* method. Thus, a call like
s:=HTMLWriter.Link(“pictures/car.gif”, “”, “View CAR image”);
will generate a string of the following pattern:
<a href=»pictures/car.gif»>View CAR image</a>

ANNEX A

Delphi HTML Controls Library

TUserInfo Auxiliary Class

Purpose

An instant of this class, returned in the application by the CurrentUser and Users properties or the GetUserInfo method of the HTMLControl component, contains information on the user.

Properties

ActiveForm -	(TForm). The form which the user currently works with. The developer can, if necessary, re-define the current user form (see expamples).
BaseURLStr -	(String). Read only. The URL which the user specified in the browser.
FinalURL –	*(*String). If the *FinalURL* property contains a reference to a certain page, the user, upon closing the application, will receive this page.
ID -	(DWord) Read only. The user ID; can have different values in different applications.
IPAddr -	(DWord). Read only. The user's IP address.
Name -	(String). Read only. The name which the user supplied in the browser when he/she first logged into the Baikonur server.
Password -	(String). Read only. The user's login password.
Params -	(String). Task launching parameters. See Examples for the format employed to pass the parameters in an application. The parameters for one and the same task can vary in the course of the user's work.

Tag –	(Longint). The field to be used by the developer (for example, to store the pointer to a structure associated with the given user).
TimeOut –	(Integer). Determines the application's idling time (in seconds) for the given user, after the expiration of which the user information will be automatically deleted. If the user is the last one, the application will be closed. *TimeOut*=-1 means no time out.

Control Components

THTMLControl

Purpose
The THTMLControl component receives the user request from the Baikonur Web server and sends the prepared HTML page back to the server. Besides, this component determines the type and behavior of the application. The HTMLControl component must be located on the application's main form; there should be only one instance of it.

Properties

CurrentUser –	(TUserInfo) A run-time and read only property. Describes the current user. You can access this property in order to receive a correct information on the user only during the time interval between the arrival of a request and the sending of a response to the user (between the *OnReceive* and *OnSend* events). At other times, you must use the *Users* property or the *GetUserInfo* method.
FinalPage -	(string) The name of the file containing the HTML page to be passed to the user when he/she closes the application. If this property and FinalURL property are empty or an incorrect file name is specified, the default final page is passed.
FinalURL –	*(*String). If the *FinalURL* property contains a reference to a certain page, the user, upon closing the application, will receive this page. This property is more prefferable to use than FinalPage.
HideApp -	(Boolean). If True, the application's icon will be hidden and will not appear in the task bar.
HTTPAdd –	(TStringList). Serves for placing additional lines of text into the HTTP header. Can be employed for passing additional application-specific information to the client, for caching control purposes, and for performing other tasks. Use this property cautiously, since an incorrectly constructed header may lead to application run-time errors.

HTTPHeader –	(TStringList). A run-time and read only property. Contains the received HTTP header from user's browser.
MultiUser -	(Boolean). If False, a separate instance of the application is launched for every user. If True, the application becomes a multi-user one (which means that all users work with a single instance of Web application). See Examples to learn more about the methods employed to implement multi-user applications.
ServInfo -	(TStringList). A run-time and read-only property. Contains the information on the Baikonur server passed to the application by the server itself (version 1.2 and later) along with the user-originated request.
TimeOut -	(Integer). Determines the application's idling time (in seconds) for the given user, after the expiration of which the user information will be automatically deleted. If the user is the last one, the application will be closed. If *TimeOut*=-1, the application will never be closed. If *TimeOut*=0, the application will start up, send out one screen, and close.
UserCount -	(Integer). A run-time and read-only property. Specifies how many users can work with the given application program.
Users –	(Array of TUserInfo). A run-time and read-only property. Contains an array describing all the users working with the given application program.

Events

OnCommand – Invoked when the application receives an HTTP command other than GET or POST (such as, for example, PUT or DELETE). If there is no appropriate event handler (or *Processed* was not set to True), the application will respond by returning the "501 Not implemented" message to the client. Normally, the *SendResponse* method must be employed to return the response.

Declaration:

TOnCommand=**procedure**(Sender:TObject; UserID: Integer; HTTP_Command: TSendMethod; **var** Data: **string**; **var** Processed: Boolean) **of object**;
UserID – User ID; a more detailed information on the user can be obtained by invoking the component's *GetUserInfo* method.
HTTP_Command - (smPut, smDelete, smUpdate, smUnknown). Determines the command contained in the incoming request. If *HTTP_Command*=smUnknown, the command can be obtained from the *Data* string (it will be the first word in that string).
Data – The string containing the entire text of the request arriving from the client, including the HTTP header.
Processed - (Boolean). If you set *Processed* to False in the handler (and send no response to the client), the application will respond be sending the "501 Not implemented" message to the client. If an appropriate response to the command was generated in the handler and sent to the client (SendResource), the *Processed*

parameter should be set to True.

OnException – Invoked in the event of an exceptional situation during the processing of the received request. If no event handler for a given exception is defined, the default script is generated and sent to the user. See “Exceptional Situations Handling” chapter for details.

Declaration:

TExceptionEvent = **procedure** (Sender: TObject; E: Exception) **of object**;
property OnException : TExceptionEvent;
OnNewUser – Invoked when a new user gets connected to the application.

Declaration:

TOnNewUser = **procedure**(Sender: TObject; UserID: DWord) **of object**;
UserID – User ID; a more detailed information on the user can be obtained by invoking the component’s *GetUserInfo* method.

OnReceive – Invoked upon receipt of a user request.

Declaration:

TOnReceive = **procedure**(Sender: TObject; **var** Form: TForm; UserID: DWord; **var** Data: **string**; **var** Action: TReceiveAction) **of object**;
Form – The given user’s current form for which the request has arrived; can be modified, although this is normally unnecessary.
UserID – User ID; a more detailed information on the user can be obtained by invoking the component’s *GetUserInfo* method.
Data – The string containing the text of the request complete with the header.
Action - (raProcess, raCancel). If *Action* is set to raCancel, no further processing of the incoming request will take place. The developer must then himself generate the data for the client and have it sent out with the aid of the *SendResource* method of the component.

OnScript – Invoked before the HTML script is received from the HTMLPage object.

Declaration:

TOnScript = **procedure**(Sender: TObject; **var** Form: TForm; UserID: DWord) **of Object**;
Form – The given user’s current form for which the HTML script is to be generated; can be modified, although this is normally unnecessary.
UserID – User ID; a more detailed information on the user can be obtained by invoking the component’s *GetUserInfo* method.

OnSend - Invoked before the HTML script is sent to the user.

Declaration:

TOnSend = **procedure**(Sender: TObject; UserID: DWord; **var** Data: **string**) **of Object**;
UserID - User ID; a more detailed information on the user can be obtained by invoking the component's *GetUserInfo* method.
Data – The string containing the text of the HTML script being sent (without the header).

OnUserGone – Invoked when the user closes the program.

Declaration:

TOnUserGone = TOnNewUser; TOnNewUser = **procedure**(Sender: TObject; UserID: DWord) **of object**;
UserID - User ID; a more detailed information on the user can be obtained by invoking the component's *GetUserInfo* method.

Methods
DeleteUserByID procedure
Declaration:
procedure DeleteUserByID(ID: Integer);
Used for deleting a user from the program. No message is sent in this case to the user. Do not resort to this method unless absolutely necessary (for example, for implementing your own *TimeOu*t). Normally, use should be made of the *UserClose* method.
ID – User ID in the program (for example, HTMLControl1.Users[i].ID).

GetUserInfo function

Declaration:

function GetUserInfo(ID : DWord): TUserInfo;
Returns an instance of the *TUserInfo* class describing the user by his/her ID.

SendErrorScript procedure

Declaration:

procedure SendErrorScript(UserID: DWord; S : **string**);
Employed for sending the default error-message HTML script. Should be used if the client's incoming request is processed "manually", such as by the *OnResource* event handler of the *HTMLPage* component or the *OnCommand* event handler of the *HTMLControl* component.
UserID – ID of the user whom the error message is sent to.
S – The text of the error message.

SendMultiPage procedure

Declaration:

procedure SendMultiPage(Page : THTMLPage; StopPage : Boolean);
The *SendMultiPage* procedure is employed for sending a next form to the users in situations involving documents of the 'multipart/mixed' type (see description of the *MultiPart* property of the *HTMLPage* component). The procedure is invoked when it is necessary to send an updated page to the users without a request on their part. Such a page will be received by all the users who are currently working with it in the browser and did not break the connection.
IMPORTANT*.* At this time, documents of the 'multipart/mixed' type are supported only by Netscape Navigator 2.0 and later.
Page – The *HTMLPage* component responsible for generating the page to be sent out.
StopPage - If True, the page will be sent out and the connection will be broken; if False, the connection will not be broken.

SendResource procedure

Declaration:

procedure SendResource(UserID: Integer; Buffer: Pointer; Size: Longint; MIME_Type: string);
Employed for sending a resource (usually a GIF image, text, or HTML) to a user in response to a browser-originated request. *SendResource* is usually employed for processing the *OnCommand* event or creating a custom HTML component that places a resource reference into the HTML script. For details, see the chapter devoted to the writing of custom components. Besides, if the developer places a resource reference into a form himself, he must invoke this procedure in the *OnResource* event handler of the HTMLPage component to have the resource sent out.
UserID –ID of the application's current user (HTMLControl.CurrentUser).
Buffer – The pointer to the buffer containing the data to be sent. It is the developer's responsibility to allocate memory for the buffer and clear it.
Size – The size of the data to be sent.
MIME_Type – The type of the data to be sent (for example, 'image/gif').

SendResponse procedure

Declaration:

procedure SendResponse(UserID: DWord; StatusCode: **string;** http_Add, Content, ContentType: **string**);
The *SendResponse* procedure is usually employed in the *OnCommand* event handler for sending a response to the client upon receipt of an HTTP command other than GET or POST, although it can also be used in the *OnReceive* event handler of the *HTMLControl* component.

UserID - ID of the application’s current user (HTMLControl.CurrentUser).
StatusCode – The string containing the response in an HTTP-compliant format, for example ‘200 OK’.
http_Add – The string containing additional text for the returned HTTP header, for example ‘WWW-Authentication: Basic realm=”/”’.
Content – The string containing contents of the application’s response; the type of data to be returned is determined by the following parameter.
ContentType – The string containing the MIME type of the returned response, for example ‘text/html’.

SendUserMultiPage procedure

Declaration:

procedure SendUserMultiPage(UserID: Integer; Page: THTMLPage; StopPage: Boolean);
The *SendUserMultiPage* procedure is employed for sending a next form to a specific user in situations involving documents of the ‘multipart/mixed’ type (see description of the *MultiPart* property of the *HTMLPage* component). The procedure is invoked when it is necessary to send out an updated page to the user without a request on his/her part. The user will then receive that page if he/she didn’t break the connection.
IMPORTANT. At this time, ‘multipart/mixed’ documents are supported only by Netscape Navigator 2.0 and later.
UserID - ID of the application’s user (HTMLControl.CurrentUser).
Page – The *HTMLPage* component responsible for generating the page to be sent out.
StopPage - if True, the page will be sent out and the connection will be broken; if False, the connection will not be broken.

UserClose procedure

Declaration:

procedure UserClose;
The *UserClose* procedure is employed when the user closes the application, for instance by clicking a button in the application. In such a case, the page specified in the *FinalPage property* or (if no such page is specified) the default page is sent. The user can also close the application by sending out a URL request of the following type: *http://www.someweb.com/demo.exe.!*
This request would close the program launched in response to the *http://www.someweb.com/demo.exe* request.

THTMLPage

Purpose

THTMLPage is responsible for updating the form's status upon receipt of a user-originated request, as well as for generating an HTML script on the basis of the current status of the active (for the given user) form. There must be one such component on every single form in the application. If it is absent, the user will receive an appropriate error message.

Properties

BackImage - (string) The URL of the file containing the background image (GIF or JPEG) for the page.

CheckFrame - (Boolean). Indicates whether the form's actuality must be checked. When you work with an Web application, each page being sent out is assigned a unique number by default. Whenever a next request is received from the user, a check is made of whether this request refers to the current form. If not, an error message is produced. An error situation may, for example, occur if the user clicks on the "Back" button in the browser to go back to the application's previous page. The user might then click a Submit-type button on that "obsolete" form, thereby originating an erroneous (in the context of the current program) request. Usually, the default value of this property needs not be altered. However, there exists a variety of forms the actuality of which is not essential (such as, for example, forms containing nothing but buttons).

clrActive, clrBackground, clrLinks, clrText, clrVisited - Type TColor. The colors to be used in displaying text on the form.

Fixed - (Boolean). Indicates whether the background image must scroll when the page in the browser is scrolled.

FontFace - (Boolean). Indicates whether the FACE attribute will include to the HTML script (<FONT> tag).

FontSize – Assumes a value in the range of -1 to 7. Denotes the size of the base font in the browser.

JSOnLoad - (string) The JavaScript methods or functions to be invoked when the page is loaded in the browser.

JSOnSubmit - (string) The JavaScript methods or functions to be invoked when the page is sent to the server.

JSOnUnload - (string) The JavaScript methods or functions to be invoked when the page is unloaded from the browser.

HeaderAdd - (TStringList). Employed to place additional tags into the HTML page header area (i.e., into the <header> ... </header> area).

HideForm - (Boolean). Indicates if the form must be hidden when the application is launched on the server.

MultiPart - (Boolean). Determines the document's type. If False, the sent-out page will have the type ‘text/html’. If True, the page will have the type ‘multipart/mixed’.

IMPORTANT. At this time, ‘multipart/mixed’ documents are supported only by Netscape Navigator 2.0 and later.

MultiPartInterval – (Byte). Determines the minimum time interval, in seconds, that must elapse before a next ‘multipart/mixed’ type document will be sent to the client. Normally, this property can be left unused, but in any case it is good practice to have the documents sent not oftener than the browser can redraw them.

RefreshInterval - (Integer). Specifies the time interval, in seconds, that must elapse before the browsers will automatically repeat their request to the application for a new form. If the value of *RefreshInterval* is less than zero, no repeated requesting takes place; if it is equal to zero, repeated requesting takes place as often as possible.

IMPORTANT. We recommend that you do not use this capability, since it may cause network loading problems.

Title - (string) The title of the application to appear in the browser’s caption.

Events

OnDataArrive – Invoked upon receipt of a user request.
Declaration:
TOnDataArrive = ***procedure***(Sender: TObject; ***var*** Data: ***string***; UserID: DWord) ***of object***;
Data – The string containing text of the request (without the HTTP header).
UserID – User ID; a more detailed information on the user can be obtained by invoking the component’s *GetUserInfo* method. Besides, the information on the addressee user is stored in the *CurrentUser* property of the *HTMLControl* component.

OnDataSend – Invoked before the HTML script is sent to the user.
Declaration:
TOnDataSend = **procedure**(Sender: TObject; UserID: Integer; **var** Data: **string**) **of object**;
UserID – User ID; a more detailed information on the user can be obtained by invoking the component’s *GetUserInfo* method. Besides, the information on the addressee user is stored in the *CurrentUser* property of the *HTMLControl* component.
Data – The HTML script (without the HTTP header) sent to the user.

OnResource – Invoked when a resource request is received from the browser.

Declaration:

TOnResource = **procedure**(Sender: TObject; ResName: **String;** UserID: DWord; **var** ResourceWasSent: Boolean) **of object**;
ResName – The name of the requested resource. It is expedient to process this event in situations when the developer has himself placed a reference to some resource onto the form (into the HTML script), for he must then himself care about having this resource sent in response to a browser request. We recommend that the *SendResource* method of the *HTMLControl* component be used for sending the necessary data. A resource reference should follow the pattern <URL of Web application>?<resource name>, for example: *project1.exe?MyResource1-22-37*. The URL of the given Web application is stored in the *BaseURLStr* variable. The resource name should be unique for the given form.
UserID – ID of the user requesting the resource.
ResourceWasSent – If you send the resource to the user yourself in the given event handler, you must set this flag to True.

OnScript - Invoked every time before an HTML script is created.

Declaration:

TPageOnScript = **procedure**(Sender: TObject) **of Object**;

OnUpdate - Invoked when the form's data is to be updated.
Declaration:
TOnUpdate = **procedure**(Sender: TObject; ValueList: TStringList) **of Object**;
ValueList – The list of the form's components along with their values. Each line of the list contains a description of the form '*HTMLEdit1=New Value'.* It is expedient to process this event in situations when the developer has himself placed some object that needs updating into HTML script, for he must then himself care about having the values of that object timely updated. The object's name should be unique for the given form.

Methods

HTMLPageScript function

Declaration:

function HTMLPageScript: **string**;
Returns the HTML script describing the form that contains the given component.

Visual components ("HTML" Page)
Properties Common to Visual Components
Align – The alignment of the component at design time.
Distance – The distance, in characters, from the element located on the left.
Visible – Determines whether the component is to be visible on the form.

THTMLLabel

Purpose

An analog of TLabel. Places text into the HTML script. Also employed for placing hypertext referencesto the page.

Properties

Caption -	(string) The text of the image caption.
Font -	(TFont). The font in which the component's text is to be displayed in the browser.

IMPORTANT. Different browsers support fonts differently.

JSOnClick -	(string) The JavaScript to be invoked when a reference is clicked (implying that the URL is not empty).
JSOnMouseOver -	(string) The JavaScript to be invoked when the mouse cursor is moved over the reference (implying that the URL is not empty).
Preformat -	(Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary.
URL -	(string) The hypertext reference. If empty, only text is placed onto the page.

THTMLTag

Purpose

With THTMLTag, you can place onto an HTML page those tags that have not been included into the library (for example, the <marquee> tag for MS Internet Explorer or ActiveX).
Properties
Caption - (string) The text of the tag caption, if the *Script* property is empty; ignored if otherwise.
Font - (TFont). The font in which the tag's text is to be presented in the browser.
Script – The text of the tag script. Can be a multi-line text and is used for inserting text, HTML tags, Java applets and ActiveXs into the page.
Tag Example
<marquee>Scrollable text</marquee>

THTMLButton

Purpose

A button control. An object for which there is a corresponding click event that causes the form’s data to be sent from the browser to the server.

Properties

ButtonType -	If btSubmit, then licking on the button will cause the request containing the form’s description to be sent to the server. Specifying btCancel disables the form editing (no data will be sent).
JSOnClick -	(string) The JavaScript to be invoked when the button is clicked.

Events

OnClick - An analog of the *OnClick* event for TButton.
Tag Example
<input type=submit name=»HTMLButton1" value=»Search»>

HTMLImageButton

Purpose

A button in the form of a picture. An object for which there is a corresponding click event that causes the form’s data to be sent from the browser to the server. The picture is stored in the component as a resource, which means that the picture file needs not necessarily be available at runtime.

Properties

AlignHoriz -	(ahDefault, ahLeft, ahCentre, ahRight). The alignment of the object on the page in the browser.
AltName -	(string) The alternate name that must appear in the browser in place of the picture while the latter is being loaded. **Note: It must have unique value or be empty.**
BorderWidth -	(Byte). The width of the border around the picture.
FileName –	The name of the picture file. In Delphi 2.x, only GIF-format images are accessible. Delphi 3.0 permits the use of pictures in BMP, GIF and JPEG formats.

Events

OnClick – Occurs when the given object is clicked with the mouse in the browser.

Declaration:

TOnClickImage = **procedure**(Sender : TObject; X,Y : Integer) **of object**;
X, Y – the coordinates of the point in which the mouse was clicked.
Tag Example
<input type=image name=»ImgBtn1" src=»demo.exe?ImgBtn1">

THTMLEdit

Purpose

A single-line text input/edit field. An analog of TEdit. The size of the field in the browser is determined by the number of characters placed in the component onto the application form.

Properties

JSOnBlur –	(String). The JavaScript to be invoked when the object becomes unfocused.
JSOnChange –	(String). The JavaScript to be invoked when a text is input into the object.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.
JSOnSelect -	(string) The JavaScript to be invoked when some text is highlighted in the object.
MaxLength –	The maximum length of input text in characters. If *MaxLength*=0, the length is unlimited.
Password –	(Boolean). If True then asterisk symbol (*) will display in place of the actual characters typed in the control.

Events

OnChange - An analog of the Tedit’s *OnChange*.
Tag Example
<input type=text name=»HTMLEdit1" value=»Hi, world!» size=26>

THTMLMemo

Purpose

A multiple-line text input/edit field. An analog of TMemo. Also serves for placing large amounts of text onto the page.

Properties

Alignment –	The alignment of text on the HTML page (when *Style*=msText).
Cols -	(Byte). The width of the text input field in character positions (when *Style*=msMemo).
Font -	(TFont). The type of the text font (when *Style*=msText).
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when a text is input into the object.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.
JSOnSelect -	(string) The JavaScript to be invoked when some text is highlighted in the object.
Lines –	(TStrings). The lines of text to be displayed.
Preformat -	(Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary (when *Style*=msText).
Rows -	(Byte). The number of text rows (when *Style*=msMemo).
Style -	(msMemo, msText). The text is displayed as an input/edit field if *Style*=msMemo, and as plain text otherwise.

Events

OnChange - An analog of OnChange for TMemo.

Tag Example
<textarea name=»HTMLMemo1" rows=5 cols=20> Text to display </textarea>

THTMLCheckBox

Purpose

An analog of TCheckBox.

Properties

Font -	The text font.
JSOnClick -	(string) The JavaScript to be invoked when the object is clicked with the mouse.
Preformat -	(Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary (when *Style*=msText).

Tag Example
<input type=checkbox name=»ChkBx1" value=»ChkBx1"> I’m using Netscape

Navigator

THTMLRadio

Purpose

An analog of TRadioButton.

Properties

Font –	The text font type.
JSOnClick -	(string) The JavaScript to be invoked when the object is clicked with the mouse.
GroupIndex -	(Integer). Determines the group which the object belongs to.
Preformat -	(Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary (when *Style*=msText).

Tag Example
<input type=radio name=»Radio1" value=»Radio1"> Variant No 1

THTMLListBox

Purpose

An analog of TListBox.

Properties

BlankChar -	(Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another character.
Cols, *Rows* -	(Byte). The dimensions of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.

Tag Example
<select name=»HTMLListBox1" size=5>
<option value=»0">Item 1
<option value=»1">Item 2
<option value=»2">Item 3
</select>

THTMLComboBox

Purpose

An analog of TComboBox.

Properties

BlankChar -	Type Character (Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another character.
Cols -	(Byte). The size of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.

Tag Example

```
<select name=»HTMLComboBox1" size=1>
<option value=»0">Item 1
<option value=»1">Item 2
<option value=»2">Item 3
</select>
```

THTMLHeader

Purpose

Places a header on the form.

Properties

Alignment –	The alignment of the header on the HTML page.
Caption -	(string) The caption text.
HeaderLevel –	Determines the size of the header on the HTML page. Valid values are 0 to 7.

Tag Example

```
<h1 align=left>My home page</h1>
```

THTMLRuler

Purpose

Something akin to TBevel. The width of the ruler on the HTML page is determined automatically.

Properties

Height –	Determines the object's vertical size.
HorizAlign –	The horizontal alignment on the HTML page.
VertAlign -	An analog of the Align property, but for design-time only.
Shade -	(Boolean). The appearance on the HTML page.

Tag Example
<hr size=3 width=100%>

THTMLImageRef

Purpose

Places onto the HTML page a hypertext reference to a file containing a GIF- or JPEG-format image.
IMPORTANT. Note that the reference specifies a relative path. The image should therefore be located in the same directory as the program itself, because otherwise the server will not be able to find it at run time. If the program is launched via an alias, the image must be located at design time in the sub-directory named exactly as the alias.
IMPORTANT. In Delphi 2.x, only GIF-format images can be displayed at design time.

Properties

AlignImage –	The alignment of the object on the HTML page relative to other elements.
AltName -	(string) The alternate name that must appear in the browser in place of the picture while the latter is being loaded.
FileName –	The name of the image file. When a reference is placed into the HTML script, it is converted into a relative path.
JSOnClick -	(string) The JavaScript to be invoked when the picture is clicked with the mouse, if there is a reference to that picture (i.e., if the URL is not empty).
JSOnMouseOver -	(string) The JavaScript to be invoked when the mouse cursor is moved over the picture, if there is a reference to that picture (i.e., if the URL is not empty).

URL -	(string) The hypertext reference. If the URL is specified, clicking on the picture will take the user to the appropriate page.

Tag Example
<img src=»GIFs\Car.gif» border=0 alt=»The big car.»>

THTMLImageRes

Purpose

An analog of *THTNMImageRef*, but places a reference to a resource into the HTML script. The image is stored within the program and is sent to the browser by the program itself.

Properties

AlignImage –	The alignment of the object on the HTML page relative to other elements.
AltName -	(string) The alternate name that must appear in the browser in place of the picture while the latter is being loaded.
FileName –	The name of the picture file. In Delphi 2.x, only GIF-format pictures are supported, while Delphi 3.0 permits the use of pictures in the BMP, GIF and JPEG formats.
ID -	(string) ID of the image. The browser usually stores the pictures it receives in cache memory. Accordingly, when it receives a page which contains a picture that has already been received earlier, the browser does not request it from the server anew, but rather uses the old one. Therefore if the contents of the given component changes, the *ID* property needs to be changed as well to make the browser repeatedly request the picture from the server.
JSOnClick -	(string) The JavaScript to be invoked when the picture is clicked with the mouse, if there is a reference to that picture (i.e., if the URL is not empty).
JSOnMouseOver -	(string) The JavaScript to be invoked when the mouse cursor is moved over the picture, if there is a reference to that picture (i.e., if the URL is not empty).
URL -	(string) The hypertext reference. If the URL is specified, clicking on the picture will take the user to the appropriate page.

Tag Example
<img src=»demo.exe?HTMLImageRes1" border=0 alt=» The big car.»>

THTMLRadioGroup

Purpose

An analog of TRadioGroup.

Properties

Orientation – (orVert, orHoriz) Defines a radio buttons orientation.
JSOnClick - (string) The JavaScript to be invoked when the radiobutton is clicked.

Visual Components (“HTML Add” Page)

THTMLHidden

Purpose

Implements a “hidden field” on the page. Its value is not displayed on the form. Can serve for the program’s housekeeping purposes and returning the JavaScript execution result to the browser.

Properties

Caption - (string) The caption field value.

Events

OnChange - Invoked every time the field value is changed.
Declaration:
TOnChange =TNotifyEvent;

Tag Example
<input type=hidden name=»HTMLHidden1" value=»222">

THTMLList

Purpose

A numbered list. Can contain hypertext references. Supports single-level lists.

Properties

Items -	Type TStrings. The items list.
NumberScheme –	The items' numbering scheme.
Ordered -	(Boolean). Indicates whether the list is to be ordered.
URLs -	Type TStrings. The list of hypertext references; corresponds to the *Items* property. A list item is displayed as plain text if the corresponding line in this list is empty, and as a reference otherwise.

Tag Example

```
<ul>
<li>Item 1
<li>Item 2
<li><a href=»www.xyz.com/homepage.htm»>Link 3</a>
</ul>
```

THTMLTable

Purpose

This component serves for placing tables onto the HTML page. At design time, you can invoke the table editor and adjust parameters of the table cells as needed.

Properties

AddHeade –	(string) Serves for placing additional attributes into the table header. Use this property cautiously, since an incorrectly constructed header may lead to incorrect view of table.
Alignment –	Defines table alignment ("alignment" attribute of Table header).
BackImage -	(string) The URL of the file containing the background image (GIF or JPEG) for the table.
BgColor -	Type TColor. Determines the background color of the table cells. If this color coincides with the form's background color, transparent background is used.
Border -	(Byte). Determines the width of the table border in the browser. If *Border*=0, the table is drawn without cell-separating rules. *CellPadding*, *CellSpacing* – (Byte). Values for corresponding table header attributes.
Cells –	Design time only. The property for editing the table cells at design time.
Cols –	Determines the number of columns in the table. Valid values are 1 through 40.
HeightInPixels -	(Integer). Determines the minimal height of the table in the browser in pixels. If *HeightInPage<=0*, the *HeightOnPage* property works.
HeightOnPage -	(Byte). Determines the height of the table in the browser

	when *HeightInPixels*<=0; *HeightOnPage* is specified in percent of the browser window's height. If *HeightOnPage*=0, the browser determines the table height itself.
Rows -	Determines the number of rows in the table. Valid values are 1 through 30.
WidthtInPixels -	(Integer). Determines the minimal width of the table in the browser in pixels. If *WidthInPage<=0*, the *WidthOnPage* property works.
WidthOnPage -	(Byte). Determines the width of the table in the browser when WidthInPixels<=0; *WidthOnPage* is specified in percent of the browser window's width. If *WidthOnPage*=0, the browser determines the table width itself.

THTMLDynamicTable

Purpose

This component serves (primarily) for displaying text as an HTML table, and makes it possible to alter the table's appearance at program run time (insert and delete rows and columns, edit text, change its font and background color, etc.). Apart from providing access to the properties of individual cells, this component allows the properties of an entire row or column to be specified. At design time, you can invoke the table editor and adjust parameters of the table cells as necessary.

Properties

AddHeade –	(string) Serves for placing additional attributes into the table header. Use this property cautiously, since an incorrectly constructed header may lead to incorrect view of table.
Alignment –	Defines table alignment ("alignment" attribute of Table header).
BackImage -	(string) The URL of the file containing the background image (GIF or JPEG) for the table.
BgColor -	Type TColor. Determines the background color of the table cells. If this color coincides with the form's background color, transparent background is used.
Border -	(Byte). Determines the width of the table border in the browser. If *Border*=0, the table is drawn without cell-separating rules.
Cell[i,j] –	A run-time property. Serves to gain access to an individual cell in the two-dimensional array of the table cells. This property returns an instance of the TDynamicCell class. A cell can have the following properties:
Align -	(haDefault, haCenter, haLeft, haRight). The horizontal alignment of text within the cell.
BGColor -	Type TColor. The cell background color.
Caption -	(string) The text to be displayed in the cell.

ColSpan -	(Byte). The number of columns to be spanned by the given cell.
NoWrap -	(Boolean). Determines whether text wrapping is allowed in the cell.
Preformat -	(Boolean). Determines whether the text is to be preformatted.
RowSpan -	(Byte). The number of rows to be spanned by the given cell.
Visible -	(Boolean). Determines whether the cell is to be visible in the table.
Valign -	(vaDefault, vaCenter, vaTop, vaBottom). The vertical alignment of text within the cell.
Width -	(Byte). The width of the cell in percent of the table width.

***Example*:** Cell[1,1].BGColor:=clRed sets background color of cell [1,1] to clRed.

CellPadding, *CellSpacing* – (Byte). Values for corresponding table header attributes.

Col[i] – A run-time property. Serves to gain access to an individual column in the one-dimensional array of the table columns. This property returns an instance of the TDynamicCol class. A column has only one property:

Visible - (Boolean). Determines whether the column is to be visible in the table.

Cols - (1..32767). Determines the number of columns in the table.

CoolTable - (Boolean). Determines how the table will be presented. If *CoolTable*=False, the color, font and alignment settings will have no effect on the table's appearance, and the HTML script will then be somewhat smaller.

DynCells – The property for editing the table cells at design time.

HeightInPixels - (Integer). Determines the minimal height of the table in the browser in pixels. If *HeightInPage<=0*, the *HeightOnPage* property works.

HeightOnPage - (Byte). Determines the height of the table in the browser when *HeightInPixels*<=0; *HeightOnPage* is specified in percent of the browser window's height. If *HeightOnPage*=0, the browser determines the table height itself.

Row[i] – A run-time property. Serves to gain access to an individual row in the one-dimensional array of the table rows. This property returns an instance of the TDynamicRow class. A row can have the following properties:

Align -	(haDefault, haCenter, haLeft, haRight). The horizontal alignment of text within the row's cell.
BGColor -	Type TColor. The background color of the row's cells.
Header -	(Boolean). Determines if the row represents a header. If True, the text is displayed in bold face.
Preformat -	(Boolean). Determines if the text is preformatted.
Visible-	(Boolean). Determines whether the row is to be visible in the table.
VAlign -	(vaDefault, vaCenter, vaTop, vaBottom). The vertical alignment of text within the row's cell.

Rows - Values (1 ... 32767). Determines the number of rows in the table.

WidthtInPixels - (Integer). Determines the minimal width of the table in the browser in pixels. If *WidthInPage<=0*, the *WidthOnPage* property works.

WidthOnPage - (Byte). Determines the width of the table in the browser when WidthInPixels<=0; *WidthOnPage* is specified in percent of the browser window's width. If *WidthOnPage*=0, the browser determines the table width itself.

This version of the library allows you to control basic parameters (color, style, size) of the font in the table cells and rows at program run time with the aid of the *FPDynCell* field (for cells) or the *FPDynRow* field (for rows). Either field is essentially a pointer to a certain structure that describes the respective cell or row, and includes font-related fields *FontName*, *FontSize*, *FontColor* and *FontStyle*.
For example, the following code alters the color and style of the text in the cell:

```
DynTabl.Cell[1,1].FPDynCell^.FontColor:=clLime;
DynTabl.Cell[1,1].FPDynCell^.FontStyle:=[fsItalic];
```

Methods

HideRow procedure
Hides a table row.

Declaration:

procedure HideRow(RowN: TDynColNumber);
RowN – The number of the row in the table (valid values are 1 through 32767).

HideCol procedure
Hides a table column.

Declaration:

procedure HideCol(ColN: TDynColNumber);
ColN - number of the column in the table (valid values are 1 through 32767).

ShowRow procedure
Unhides the hidden row in the table.

Declaration:

procedure ShowRow(RowN: TDynColNumber);
RowN – The number of the row in the table (valid values are 1 through 32767).

ShowCol procedure
Unhides the hidden column in the table.

Declaration:

procedure ShowCol(ColN: TDynColNumber);
ColN – The number of the column in the table (valid values are 1 through 32767).

AppendRow procedure
Appends an empty row to the table.

Declaration:

procedure AppendRow;

AppendCol procedure
Appends an empty column to the table.
Declaration:
procedure AppendCol;

InsertRow procedure
Inserts an empty row into the table at the specified location.

Declaration:

procedure InsertRow(RowN: TDynColNumber);
RowN – The number of the row in the table (valid values are 1 through 32767).

InsertCol procedure
Inserts an empty column into the table at the specified location.

Declaration:

procedure InsertCol(ColN: TDynColNumber);
ColN – number of the column in the table (valid values are 1 through 32767).

DeleteRow procedure
Deletes the specified row from the table.

Declaration:

procedure DeleteRow(RowN: TDynColNumber);
RowN – The number of the row in the table (valid values are 1 through 32767).

DeleteCol procedure
Deletes the specified column from the table.

Declaration:

procedure DeleteCol(ColN: TDynColNumber);
ColN – The number of the column in the table (valid values are 1 through 32767).

Clear procedure
Empties text from all cells of the table, and sets the color and other parameters to their default values.

Declaration:

procedure Clear;

ClearRow procedure
Empties text from all cells of the specific row of the table, and sets the color and other parameters to their default values.

Declaration:

procedure ClearRow(RowN: TDynColNumber);

ClearCol procedure
Empties text from all cells of the specific column of the table, and sets the color and other parameters to their default values.

Declaration:

procedure ClearCol(ColN: TDynColNumber);

THTMLFileListBox, THTMLDirListBox, THTMLDriveComboBox, THTMLFilterComboBox

Purpose

These four components are full analogs of the respective TFileListBox, TDirectoryListBox, TDriveComboBox, TFilterComboBox standard components, and can be used for constructing a file select dialog form. However, since changes in type ComboBox or ListBox objects do not lead to the passing of appropriate data to the server when you work in the browser (if *JavaScriptOn* is False), you must place a Submit type button onto the form.

Properties

JavaScriptOn –	(Boolean) Determines whether a Submit event executed by the Java script will occur following a change in the component.

TJavaScript

Purpose

Serves for storing the JavaScript program placed into the HTML page header.

Properties

Script –	(TStrings). The JavaScript program proper.

THTMLChart

Purpose

This component is accessible in Delphi 3.x only. It serves for displaying graphs on the page. THTMLChart is a full analog of the TChart component. At application run time, it converts the picture into JPEG format.

Properties

AlignImage –	The alignment of the object on the HTML page relative to other elements.
AltName -	(string) The alternate name that must show in the browser in the place of the picture while the latter is being loaded.
CompressQuality –	Image quality. Depends on the size of the picture being loaded. Valid values are 1 through 100).
ID -	(string) ID of the image. The browser usually stores the pictures it receives in cache memory. Accordingly when it receives a page which contains a picture that has already been received earlier, the browser does not request it from the server anew, but rather uses the old one. Therefore if the contents of the given component changes, the *ID* property needs to be changed as well to make the browser repeatedly request the picture from the server.
JSOnClick -	(string) The JavaScript to be invoked when the picture is clicked with the mouse, if there is a reference to that picture (i.e., if the URL is not empty).
JSOnMouseOver -	(string) The JavaScript to be invoked when the mouse cursor is moved over the picture, if there is a reference to that picture (i.e., if the URL is not empty).
URL -	(string) The hypertext reference. If the URL is specified, clicking on the picture will take the user to the appropriate page.

THTMLPutFile

Purpose

This component makes it possible to select and send a file from the user's browser. It works in Netscape browsers starting from version 2.0 and in Microsoft Explorer starting from version 3.02 (with updates).

Properties

DstFileName –	(string) The name of the destination file on the server in which the sent file must be saved.
MaxSize –	(Integer) The maximum size of the received file in kilobytes. If the file exceeds the specified size, two situations are possible. If the file is small enough to be received completely (this depends on the Baikonur server's settings, see the MaxReceiveBuffer parameter), the program will post a corresponding message to the user. If the file is too large to be received completely, the browser will notify the user that the connection with the server has been terminated.

Events

OnFileReceive – This event occurs after the file is received completely.

Declaration:

TOnFileReceive **= procedure**(Sender: TObject; SrcFileName: **string**; Data: Pointer; Size: Integer; **var** Action: TFileReceiveAction) **of object;**
SrcFileName – The name of the source file on the user's machine.
Data – The pointer to the memory area containing the sent file.
Size – The size of the data received into the buffer.
Action – Determines the component's further actions. If you assign frCancel, the data will not be saved into a file. This is necessary if the event handler has already saved the data (for example, into a database).

THTMLTreeView

Purpose

A tree view control displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. THTMLTreeView can display items in the form of hypertext references or any HTML tags, defined by user.

Properties

Font –	(TFont). Tree node font.
GoByClick -	(Boolean). If True and OnGetNodeURL and OnGetNodeScript are not assigned or return empty values for current node, it is displayed as hypertext reference.

HideSelection - (Boolean). If True, there are no selected node font changing.

IndentCols – (Byte). Number of space symbols between parent and child nodes. If *Preformat* is False, it is ignored. If value is 0, the special tag is applied.

MaxCols – (Integer). Maximum number of symbols displayed in a tree node. If the node caption exeeds *MaxCols*, the string is cut and three dots are add («...»).

MaxVisibleNodes – (Integer). Maximum number of tree nodes displayed. If ShowAll is False, this property is ignored. If property value is less than zero (-1), all nodes are displayed.

Preformat - (Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary.

SelectedFont – (TFont). Selected tree node font. If *HideSelection* is True, this property is ignored.

ShowAll - (Boolean). If True, the number of tree nodes defined in *MaxVisibleNodes* and *TopVisibleNode* properties is displayed. Else it depends of the TreeView size.

ShowButtons - (Boolean). If True, every tree node has "+" or "-" button, depending on its state (expanded or collapsed).

TopVisibleNode – (TTreeNode). Run-time only. If ShowAll=True and MaxVisibleNodes>0, it defines the first displayed node.

The following properties exist in Delphi 3.x only.

BMPCompressQuality - (1..100) Defines image JPEG compression quality.

Images – (TImageList). Images determines which image list is associated with the tree view.

StateImages – (TImageList). StateImages determines which image list to use for state images.

Events

OnGetScript - This event occurs when the control tries to get an user-defined HTML-script for displaying a tree node.

Declaration:

THTMLTVGetNodeScriptEvent = **function**(Sender: TObject; Node: TTreeNode): **string of object**;

OnGetURL – This event allows to define hypertext reference for a tree node.

Declaration:

THTMLTVGetNodeURLEvent = **function**(Sender: TObject; Node: TTreeNode): **string of object**;

Visual Components (“HTML DB” Page)

THTMLDBGrid

Purpose

This component serves for displaying (and modifying) database data in the form of a table. On the HTML page, the usual DBGrid resembles a table only remotely. To navigate between the database records, use should be made of the HTMLNavigator. If a field is editable, it is shown in the browser as an HTMLEdit. If PickList for that field is defined, it will be displayed as a combo box.

Properties

AddHeade –	(string) Serves for placing additional attributes into the table header. Use this property cautiously, since an incorrectly constructed header may lead to incorrect view of table.
Alignment –	Defines table alignment (“alignment” attribute of Table header).
BackImage -	(string) The URL of the file containing the background image (GIF or JPEG) for the table.
Border -	(Byte). Determines the width of the table border in the browser. If *Border*=0, the table is drawn without cell-separating rules. *CellPadding*, *CellSpacing* – (Byte). Values for corresponding table header attributes.
Color -	Type TColor. Determines the background color of the grid cells. If this color coincides with the form’s background color, transparent background is used.
CustomGrid -	(Boolean). Determines how the table will be presented. If *CustomGrid*=False, the color, font and alignment settings for columns will have no effect on the grid’s appearance, and the HTML script will then be somewhat smaller. See also *FontFace* property of *HTMLPage* component.
FixedColor –	(TColor). Defines the background color of the header row and the indicator (most left) column.
GoByClick –	(Boolean). If True, the user can change the current record by clicking on the indicator field of the required record in the grid.
GoByClickImageURL -	(string) The URL of the file containing the non-selected row indicator image (GIF or JPEG) for the grid. If no URL assigned, the default transparent picture is displayed.
GoByClickAltName –	(string). AltName for indicator image. At run-time a row number is attached to AltName.
HeightInPixels -	(Integer). Determines the minimal height of the grid in the browser in pixels. If *HeightInPage<=0*, the *HeightOnPage* property works.

HeightOnPage -	(Byte). Determines the height of the grid in the browser when *HeightInPixels*<=0; *HeightOnPage* is specified in percent of the browser window's height. If *HeightOnPage*=0, the browser determines the grid height itself.
ShowAll -	(Boolean). If set to True, all records of the data source will be shown on the page, there will be no current record indicator, and it will be impossible to navigate through the data set.
Options –	Determines how the table will appear on the page: dgTitles determines whether the table will have a header, and dgIndicator determines whether the table will feature a current record indicator.
TitleFont –	(TFont). Header row and indicator column font. **Note**: TitleFont property of DBGrid is not stored in a DFM file. You should set title font for DBGrid and HTMLDBGrid at run time.
WidthtInPixels -	(Integer). Determines the minimal width of the grid in the browser in pixels. If *WidthInPage<=0*, the *WidthOnPage* property works.
WidthOnPage -	(Byte). Determines the width of the grid in the browser when WidthInPixels<=0; *WidthOnPage* is specified in percent of the browser window's width. If *WidthOnPage*=0, the browser determines the grid width itself.

THTMLNavigator

Purpose

This is an analog of TDBNavigator. By default, five button controls (First, Prior, Next, Last and Refresh) are visible, although any other button(s) can be included as well. The text to be displayed on the buttons (button captions) is stored in the NavCap array of the HTMLGRDS module.

THTMLDBText

Purpose

An analog of TDBText.

Properties

Preformat -	(Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary.

THTMLDBEdit

Purpose

An analog of TDBEdit. The field which the component is linked to is updated if the data in the browser was modified by the user and the field updating option is enabled (i.e., the *ReadOnly* property of the *TDBEdit* component is set to False and the data set can be changed to the edit mode).

Properties

JSOnBlur - (string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange - (string) The JavaScript to be invoked when a text is input in the object.
JSOnFocus - (string) The JavaScript to be invoked when the objects has the focus.
JSOnSelect - (string) The JavaScript to be invoked when some text is highlighted in the object.
ReadOnly - (Boolean). If True, updating is enabled.

THTMLDBMemo

Purpose

This is an analog of TDBMemo.

Properties

Cols - (Byte). The width of the text input field in character positions (when *Style*=msMemo).
JSOnBlur - (string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange - (string) The JavaScript to be invoked when a text is input in the object.
JSOnFocus - (string) The JavaScript to be invoked when the objects has the focus.
JSOnSelect - (string) The JavaScript to be invoked when some text is highlighted in the object.
Preformat - (Boolean). If False, any «extra» white spaces are deleted, and line feed characters are added if necessary. (when *Style*=msText).
Rows - (Byte). The number of rows (when *Style*=msMemo).
Style - (msMemo, msText). The text is displayed as an input field if *Style*=msMemo, and as plain text otherwise.

ReadOnly -	(Boolean). If True, modification is enabled.
UpdateMode -	(umNoUpdate, umAlways, umModified). Determines the memo field's update mode. If *umNoUpdate* is specified, the field is never updated. If *umAlways* is specified, the field is always updated. If *umModified* is specified, the field is updated only if the data has been modified.

THTMLDBCheckBox

Purpose

An analog of TDBCheckBox.

Properties

JSOnClick -	(string) The JavaScript to be invoked when the object is clicked with the mouse.

THTMLDBListBox

Purpose

An analog of TDBListBox.

Properties

BlankChar -	(Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another character.
Cols, *Rows* -	(Byte). The dimensions of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.

THTMLDBComboBox

Purpose

An analog of TDBComboBox.

Properties

BlankChar -	(Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another character.
Cols -	(Byte). The size of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.
NoValue –	(string). If DataField contains a value not defined in the *Items* property of the ComboBox control, the *NoValue* string is displayed in ComboBox. If user selects *NoValue* item in the ComboBox, the DataField value will not change.

THTMLDBLookupListBox

Purpose

An analog of TDBLookupListBox.

Properties

BlankChar -	(Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another character.
Cols, *Rows* -	(Byte). The dimensions of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.

THTMLDBLookupComboBox

Purpose

An analog of TDBLookupComboBox.

Properties

BlankChar -	Type Character (Char). Since all «extra» white spaces are automatically deleted, they can be replaced with another

	character.
Cols -	(Byte). The size of the object on the HTML page.
JSOnBlur -	(string) The JavaScript to be invoked when the object becomes unfocused.
JSOnChange -	(string) The JavaScript to be invoked when another item of the list is selected.
JSOnFocus -	(string) The JavaScript to be invoked when the objects has the focus.
NoValue –	(string). If DataField contains a value not defined in the *Items* property of the ComboBox control, the *NoValue* string is displayed in ComboBox. If user selects *NoValue* item in the ComboBox, the DataField value will not change.

THTMLDBRadioGroup

Purpose

An analog of TDBRadioGroup.

Properties

Orientation –	(orVert, orHoriz) Defines a radio buttons orientation.
JSOnClick -	(string) The JavaScript to be invoked when the radiobutton is clicked.

THTMLDBGIF

Purpose

Serves for displaying the GIF image stored in the table. An analog of TDBImage, except that it can handle images only in the GIF format.

Properties

AltName -	(string) The alternate name that must appear in the browser in place of the picture while the latter is being loaded.
ID -	(string) The picture ID employed for generating the component's name in the HTML script if the *IDType* property is set to itID or itKeyID. The browser usually stores the pictures it receives in cache memory. Accordingly, when it receives a page which contains a picture that has already been received earlier, the browser does not request it from the server anew, but rather uses the old one. Therefore if the contents of the given component changes, the *ID* property needs to be changed as well to make the browser repeatedly request the picture from the server.

IDType –	Valid value is itUnique, itKey, itID or itKeyID. Determines the method by which component’s name in the HTML script will be generated:
itUnique –	in every request instance, a unique name is generated for the component; the picture is sent to the browser every time the current record is accessed.
itKey –	the name is generated based on the record’s original key; the picture for the current record is sent to the browser only once; however, if the picture in that record is modified, it will not be sent to the browser but rather retrieved by the browser from its cache memory.
itID -	the name is generated based on the ID property; unless its value is changed, the browser will not request the picture but will rather retrieve it from its cache memory; it is the developer’s responsibility to timely track changes in the picture ID.
itKeyID -	the name is generated based on the record’s original key and the ID property; this means that when the picture in the record is changed, it suffices to change the ID of that record; the ID can be of any nature (such as, for example, a DataStamp-type field).
JSOnClick -	(string) The JavaScript to be invoked when the picture is clicked with the mouse, if there is a reference to that picture (i.e., if the URL is not empty).
JSOnMouseOver -	The JavaScript to be invoked when the mouse cursor is moved over the picture, if there is a reference to that picture (i.e., if the URL is not empty).
URL -	(string) The hypertext reference. If the URL is specified, clicking on the picture will take the user to the appropriate page.

Tag Example
<img src=»demo.exe?DBGIF135353,5108045139" border=0 alt=»Picture from table»>

THTMLDBImage

Purpose

THTMLDBImage is a complete analog of the THTMLDBGIF component, except that it is available in Delphi 3.0 only and can handle images in the BMP, GIF and JPEG formats.

THTMLDBChart

Purpose

The component is accessible in Delphi 3.0 only, and serves for displaying charts based on the database table of the page. A complete analog of the TDBChart component. Analogous of the THTMLChart component in terms of its additional properties.

ANNEX B

HTML Elements and Components of Delphi HTML Controls Library

Table B.1. Summary Table of HTML Elements

Tag	Description
!—...—	Comments. Any text surrounded by tags will be output in a browser
!DOCTYPE	Describes the HTML version used in the current document
A	Sets an anchor. HREF=*attribute* creates a hyperlink pointer. NAME=*attribute* creates references by name
ADDRESS	Specifies the mail address
APPLET	Embeds a Java applet. See description of OBJECT
AREA	Determines appearance of the active area in a picture
B	Changes text face to bold. See description of STRONG
BASE	Determines the URL for the document
BASEFONT	Sets the base font value
BGSOUND	Adds background sounds to be played back during initial loading
BIG	Increases the font size
BLOCKQUOTE	Highlights quoted text as a separate block
BODY	Specifies the beginning and end of a document body. See description of HEAD
BR	Inserts a line break
CAPTION	Specifies a caption for a table. Valid only within the TABLE element
CENTER	Centers text and images
CITE	Points to a citation. CITE is used to represent a book, document or other published matter
CODE	Represents a program code example
COL	Sets the properties of a table column
COLGROUP	Sets the properties of one or more columns as a group
COMMENT	Point to a comment

DD	Points to a data definition. See description of DL and DT
DFN	Formats the term being defined
DIR	Specifies a directory-like list
DIV	Sets the document’s division(s). Groups related elements together within the document
DL	Denotes a list of definitions. DL is used with the list of terms being defined. See description of DT and DD
DT	Specifies the definition term. DT is used for formatting the term being defined. See description of DL and DD
EM	Marks up text, usually by presenting it in italics
EMBED	Specified embedded objects. See description of OBJECT
FONT	Formats font style, size and color
FORM	Specifies the form to be employed by the user to input data. See description of INPUT for a list of the form’s elements
FRAME	Specifies independent frames within a page. See description of
FRAMESET	Specifies the layout of frames within a page. See description of FRAME
Hn	Determines the style of the header text
HEAD	Marks up the header of an HTML document
HR	Draws a horizontal line. HR is used for separating sections
HTML	Specifies the file as having an HTML-document format
I	Presents text in italics
IMG	Insets an image file
INPUT	Determines the form’s control (such as, for example, an on/off button or a radio button). See description of FORM
ISINDEX	Points to the presence of a search index
KBD	Specifies the text to be input from keyboard
LI	Specifies a list item. Adds a special symbol or a digit, depending on the use. See description of UL and OL.
LINK	Establishes links between documents
LISTING	Presents text in fixed-width font
MAP	Maps the set of active areas within a picture
MARQUEE	Outputs text in a marquee (“scrolling text”) window
MENU	Specifies a menu-like list of items
META	Presents information about the document in HTTP header
NOBR	Disables line breaks
NOFRAMES	Indicates that the contents can be viewed only by browsers that do not support frames
OBJECT	Inserts OLE control element
OL	Specifies an ordered list of items. Each item on the list has an alphabetic or numeric reference. See description of UL and LI
OPTION	Specifies the item (option) selected in the list
P	Inserts an end-of-paragraph symbol and marks up the beginning of next paragraph
PARAM	Sets parameters of the object’s properties
PLAINTEXT	Presents text in fixed-width font without processing of tags
PRE	Outputs text precisely as it was input, including all line breaks and white

	spaces
S	Presents text in strikethrough face
SAMP	Defines a text sample. See description of CODE
SCRIPT	Defines the inclusion of a script as a page component part
SELECT	Designates a list or dropout list
SMALL	Reduces font size
SPAN	Specifies application of style information to the embedded text
STRIKE	Presents text in strikeout face. See description of S
STRONG	Highlights text, usually presenting it in bold face. See description of B
SUB	Presents text as subscript
SUP	Presents text as superscript
TABLE	Creates a table. See description of TH, TR and TD to learn how table rows and columns are defined
TBODY	Determines the body of a table
TD	Creates a table cell
TFOOT	Creates a single-line footnote in a table
TH	Creates a row or column header in a table
THEAD	Defines table header
TEXTAREA	Creates window in which the user can input and/or edit text
TITLE	Defines title of the document
TR	Creates a table row
TT	Designates a teletype mode. Outputs text in fixed-width font
U	Outputs text in underlined face
UL	Formats text with positional markers. See description of LI
VAR	Designates text to take place of a variable. Outputs text in small-size fixed-width font
WBR	Inserts a “soft” line break into a NOBR block of text
XMP	Specifies a text example. Outputs text in fixed-width font

<!— ... —>

Description

Indicates that the enclosed text is the program author’s comment. Any text enclosed between these tags will be ignored. You can include several lines of text between the opening and closing tags.

Example

```
<!— This line text, enclosed in an HTML page, will not print.
        This line of text will not print —>
```

Component

No direct analog. Programmers can use HTMLTag component

<!DOCTYPE>

Description

Specifies version of the HTML used in the document. !DOCTYPE is the first element in any HTML document. !DOCTYPE is a mandatory element for any document compatible with HTML 3.2 document.

Example
<!DOCTYPE HTML PUBLIC «-//W3C//DTD HTML 3.2//EN»>

Component
No direct analog. Programmers can use HTMLTag component

<A HREF=reference NAME=name REL=relationship REV=revision TARGET=window TITLE=title>

Description

Sets an anchor. Tags A and /A may enclose text or graphics. The properties of the elements following the A tag apply to the text or graphics enclosed in brackets. The A tag can be employed to anchor a hyperlink pointer to certain text or graphics by specifying HREF=*attribute*. The A tag can be employed to define text or graphics as a reference by name by specifying NAME=*attribute*. Anchors can be embedded.

HREF=***reference***

References the destination address or destination file. The destination address must be in URL format. The destination file should specify a file and have a format compatible with the given file system. If no search path or domain is specified, the file is searched for in the same location as the current document.

NAME=***name***

Sets a reference by name within the HTML document. Reference to a name can be made in the given document and external documents (by prefixing the name with the symbol #).

REL=***relationship***

Determines the relative relationship.

REV=***revision* - modification**

Specifies the version (revision) number.

TARGET=***window***

Indicates that the linked document must be loaded into the destination window. This attribute can be specified if you use frames and specify a frame in the FRAME element. Valid value for *window* can be one of the following:

window	Indicates that the linked document must be loaded into the destination window. To be valid, w*indow* should begin with an alphanumeric character, with the exception of the four destination windows described in more detail below.
_blank	Indicates that the linked document must be loaded into a new unfilled window. This window has no name.
_parent	Indicates that the linked document must be loaded directly into the parent document of the document containing the link pointer.
_self	Indicates that the linked document must be loaded into the same window in which the link pointer was selected with a click of the mouse.
_top	Indicates that the linked document must be loaded into entire window space.

TITLE=***name***
Names the title that must appear when the hyperlink pointer is selected.

Examples
<A HREF=»http://www.microsoft.com»>This is a pointer to a link to Microsoft.</A>
<A HREF=»home.htm»>This is a pointer to a link to a htm file named Home.htm in the same directory as this page.
</A>
<A TARGET=»viewer» HREF=»sample.htm»>Click here to load the link pointer into the viewer's window.
</A>

Component
A reference can be implemented with the help the HTMLLabel component, specifying the destination address in the URL property. You can use HTMLTag to set the anchor.

<ADDRESS>

Description

Specifies the mail address. This element is usually employed in the bottom part of a document. The address text is output in italics.

Example
<ADDRESS>This text will print in italics.
</ADDRESS>

Component
No direct analog. Programmers can use the HTMLTag component before and after the tagged text.

<APPLET
ALIGN=LEFT|RIGHT|CENTER
ALT=alternateText
CODEBASE=codebaseURL
CODE=appletFile
HEIGHT=pixels
HSPACE=pixels >
NAME=appletInstanceName
[<PARAM NAME = AttributeName >]
WIDTH=pixels
VSPACE=pixels </APPLET>

Description

Loads a Java applet into an HTML document.
ALIGN=***alignment***
Specifies alignment type for the object in the text.
ALT=***alternateAppletText***
Alternate representation of the text for text-only browsers that do not support Java.
CODE=***appletFile***
The name of the Java applet.
CODEBASE=***codebaseURL***
Specifies the applet's base URL (directory in which the applet is located).
HEIGHT=***pixels***
Specifies the initial height, in pixels, of the applet output area.
HSPACE=***pixels***
Specifies the horizontal size.
NAME=*appletInstanceName*
Assigns a name to identify the given applet among other applets within an HTML page.
PARAM NAME=***AttributeName***
Employed for passing applet-specific arguments from the HTML page.
VSPACE=***pixels***
Specifies the amount of empty space, in pixels, above the applet.
WIDTH=***pixels***
Specifies the initial width, in pixels, of the applet output area.

Component
No direct analog. Programmers can use the HTMLTag component.

<AREA
COORDS=*coords*
SHAPE=*shape-type*
NOHREF
HREF=*url*
TARGET=*window*>

Description
Determines how the active area will appear in pictures.
COORDS=***coords***
Specifies the coordinates determining the appearance of the active area.
HREF=***url***
Specifies the destination point referenced to the active area.
NOHREF
Indicates that a mouse click in this area will produce no action.
SHAPE=***shape-type***
Specifies type of the outward appearance (shape) of the area. Valid value for s*hape-type* can be one of the following:

RECT, RECTANGLE	Rectangle. Four coordinates (*x1*, *y1*, *x2* and *y2)* need be specified.
CIRC, CIRCLE	Circle. Three coordinates (x-center, y-center and radius) need be specified.
POLY, POLYGON	Polygon. Three or more coordinate pairs defining the polygon's area need be specified.

TARGET=***window***
See above.

Examples:
<AREA SHAPE=»RECT» COORDS=»50, 25, 150, 125" HREF=»http://www.sample.com»>
<AREA SHAPE=»RECT» COORDS=»50, 25, 150, 125" NOHREF>
<AREA TARGET=»viewer» HREF=»sample.htm» SHAPE=»CIRCLE» COORDS=»50, 25, 150, 125">

Component
No direct analog. Programmers can use the HTMLTag component.

<B>

Description

Outputs text in bold face.
Example

<B>This text will be displayed in bold face.</B>

Component
Use the Font property for text components.

<BASE HREF=*url* TARGET=*window*>

Description

Specifies the URL for the document.
HREF=***url***
Specifies the full URL for the document in case the latter is read in context-independent mode but the user wishes to make a reference to the original.
TARGET=***window***
See above.

Examples
<BASE HREF=»http:// www.sample.com/hello.htm»>
<BASE HREF=»http:// www.sample.com/hello.htm» TARGET=»viewer»>

Component
No direct analog. Normally needs not to be used in programming. Programmers wishing to add a header to their HTML page can use the *HeaderAdd* property of the HTMLPage component.

<BASEFONT COLOR=*color* NAME=*name* SIZE=*n*>

Description

Sets the base font size. Base font size is the size to be applied by default to the font of any text that is not formatted with the help of a styles list or with the aid of the FONT element.
COLOR=***color***
Specifies the base font color.
NAME=***name***
Specifies the base font name.
SIZE=***n***
Specifies the base font size. The value of *n* can be 1 through 7 inclusive. The default value is 3, and 7 is the maximum value. Accordingly, relative font size settings can be specified throughout the document (for example, by specifying <FONT SIZE=+3>).

Example
<BASEFONT SIZE=3> Set the base font size to 3.
<FONT SIZE=+4> The font size is now equal to 7.

<FONT SIZE=-1> Now, text will print in font size 2.

Component
No direct analog. Normally needs not be used in building programs.

<BGSOUND SRC=*url* LOOP=*n* >

Description

Adds background sounds or «sound tracks» to the page. Sounds recorded both in a sampled (WAV or AU) format or a MIDI format are acceptable.
SRC=***url***
Specifies the source address of the sound to be played back.
LOOP=***n***
Specifies the number of playback loops after activation. If *n* =-1 or *LOOP*=INFINITE is specified, the sound will be played back infinitely.

Component
No direct analog. Programmers can use the HTMLTag component.

<BIG>

Description

Makes the text one size bigger.

Example
<BIG>This text is bigger.
</BIG>

Component
No direct analog. Programmers can use the HTMLTag component.

<BLOCKQUOTE>

Description

Sets indents for the right and left margins. These are used to mark up citations in the text.

Example
<P>He said:
<BLOCKQUOTE>»Hi, there!»</BLOCKQUOTE>

Component
No direct analog. Programmers can use the HTMLTag component.

<BODY BACKGROUND=*url* BGCOLOR=*color* BGPROPERTIES=FIXED LINK=*color* TEXT=*color* TOPMARGIN=*n* VLINK=*color* >

Description

Specifies the beginning and end of the document's body. This element also allows you to specify background image, background color, color of links, and top and left margins of the page.

BACKGROUND=***url***

Specifies the background image of the page. This image is displayed tile-like behind the text or graphics on the page.

BGCOLOR=***color***

Sets the background color of the page. The value of c*olor* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BGPROPERTIES=**FIXED**

Specifies the «watermark» – an unscrollable (fixed-position) background picture.

LEFTMARGIN=***n***

Specifies the size of the left-hand margin for the entire body of the page and re-defines the default margin settings. If *LEFTMARGIN* is set to zero, the left-hand margin will be located exactly at the left-hand edge of the page.

LINK=***color***

Sets the color of pointers to the hyper links that have not been used yet. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

TEXT=***color***

Sets the color of text on the page. The value of *color* can be specified in hexadecimal or RGB format, or be a redefined color name. See description of Color.

TOPMARGIN=***n***

Sets the size of the top margin of the page and re-defines its default setting. If *TOPMARGIN* is set to zero, the top margin will be located exactly at the top edge of the page.

VLINK=***color or colorname***

Sets the color of the hyper link pointers that have already been used. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

Examples

The following HTML example inserts a background image into the current page:

<BODY BACKGROUND=»/ie/images/watermrk.gif» BGPROPERTIES=FIXED BGCOLOR=#FFFFFF TEXT=#000000 LINK=#ff6600 VLINK=#330099>

<HTML> <BODY>Here's a Web page!</BODY></HTML>

Component

The tag is automatically produced by the HTMLPage component. The color settings are determined by the FclrBackground, FclrText, FclrLinks, FclrVisited, and FclrActive properties, and the background image is determined by the Background property.

<BR CLEAR=align-type>

Description

Inserts a line break.

CLEAR=***align-type***

Inserts vertical white spaces so that the text to be printed next would be located below the «floating» images aligned to the left- or right-hand edge of the page. Valid value for *align-type* can be one of the following:

LEFT	Inserts white spaces so that the text to be output next would appear aligned to the left-hand edge of the page immediately below the left-aligned floating image.
RIGHT	Inserts white spaces so that the text to be output next would appear aligned to the right-hand edge of the page immediately below the right-aligned floating image.
ALL	Places text after all floating images.

Component

No direct analog. Programmers can use the HTMLTag component.

<CAPTION ALIGN=*align-type*>

Description

Defines a table caption.

ALIGN=***align-type***

Sets the alignment type for the table caption. Valid value for *align-type* is LEFT, RIGHT, TOP or BOTTOM. By default, the caption is centered and located at the bottom of the table.

This element is valid only within the TABLE element. Use of closing tag is mandatory.

Example

```
<TABLE>
<CAPTION ALIGN=BOTTOM>
This caption appears centered at the table bottom.
</CAPTION>
<TR>
```

```
    ....
</TR>
</TABLE>
```

<CENTER>

Description

Centers text and images.

Example

```
<CENTER>Hi, there!</CENTER>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<CITE>

Description

Indicates that this is a citation, and is employed to represent a book, a document or some other published source.

Example

```
<CITE>Book Title.</CITE>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<CODE>

Description

Indicates that this is a program code example. Presents text in small-size font. If no font type is specified, a fixed-width font is used.

Example

```
<CODE>Here is a text in small-size fixed-width font.
</CODE>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<COL ALIGN=*align-type* SPAN=*n*>

Description

Sets the properties of one or more columns of the table. Use this element in conjunction with the COLGROUP element to define properties of a column within a group of columns.
ALIGN=***align-type***
Specifies the alignment type for the text, in font recticles, within the column. Valid value for *align-type* is CENTER, LEFT or RIGHT.
SPAN=***n***
Sets the number of consecutive columns whose properties are being defined.
This element is valid only within the table. Use of closing tag is unnecessary and is not recommended.
The properties set with the COL element always override and re-define the properties specified with the preceding COLGROUP element.

Example

```
<TABLE>
<COLGROUP>
  <COL ALIGN=RIGHT>
  <COL ALIGN=LEFT>
<COLGROUP>
  <COL ALIGN=CENTER>
<TBODY>
   <TR>
  <TD>This is the first column in the group, and it is right-justified.
</TD>
  <TD> This is the second column in the group, and it is left-justified.
</TD>
  <TD>This column is in a new group, and it is centered.
</TD>
   </TR>
</TABLE>
```

<COLGROUP ALIGN=*align-type* SPAN=*n*>

Description

Sets the properties of a group of columns.
ALIGN=**align-type**
Specifies the alignment type for the text in the cells of the table column(s). Valid value for *align-type* is CENTER, LEFT or RIGHT.

SPAN=***n***

Sets the number of consecutive columns making up the group for which the properties are being defined.

This element is valid only within the table. Use of closing tag is unnecessary and is not recommended. If different properties need be assigned to the columns in your group, use COLGROUP in conjunction with one or more COL elements in order to assign specific properties for each individual column. If groups in the table element are defined with the use of RULES=*attribute*, this element also determines the way table rules will drawn (vertical rules will then be drawn between groups of columns rather than between individual columns).

Example

```
<TABLE>
<COLGROUP ALIGN=RIGHT>
<COLGROUP SPAN=2 ALIGN=LEFT>
<TBODY>
   <TR>
  <TD>This column is located in the first group, and will be right-justified.
</TD>
  <TD> This column is located in the second group, and will be left-justified.
</TD>
  <TD> This column is located in the second group, and will be left-justified.
</TD>
   </TR>
</TABLE>
```

<COMMENT>

Description

Defines the text as a comment. The text in the COMMENT element is not displayed in a browser if it doesn’t contain a HTML code.

Example

```
<COMMENT>This text will not be output to the screen.</COMMENT>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<DD>

Description

Points to a definition in the definitions list. Indicates that the text is the definition of a term, and must therefore be output in the right-hand column of the definitions list.

Example

```
<DL><DT>Cat<DD>A clever furred animal who purrs and loves milk.
```

<DT>Lizard<DD>A mysterious desert animal with a long tongue.
</DL>

<DFN>

Description

Indicates that this is a definition. Formats the term when it first appears in the document.

Example
<DFN>HTML stands for Hypertext Markup Language.
</DFN>

Component
No direct analog. Programmers can use the HTMLTag component.

<DIR>

Description

Denotes a directory-like list. Indicates that the following block of text is made up of individual items each of which begins with the <LI> element. The items must be no more than 20 characters long and be output column-wise.

Example
<DIR> <LI>Art
<LI>Hystory
<LI>Literature
<LI>Sports
<LI>Entertainment
<LI>Science
</DIR>

Component
No direct analog. Programmers can use the HTMLList component. See description of the <UL> tag.

<DIV
ALIGN=*align-type*
</DIV>

Description

Sets the document divisions. Groups related elements together.
ALIGN=***align-type***

Sets the alignment type for the block items within the DIV elements. Valid value for *align-type* is LEFT, CENTER or RIGHT. The default value is LEFT.

Example
```
<DIV>
This text represents a section (division).
</DIV>
<ex><DIV ALIGN=CENTER>
This text represents another division.
</DIV>
```

Component
No direct analog. Programmers can use the HTMLTag component.

<DL>

Description

Indicates that the following block of text is a definitions list (i.e., an automatically formatted two-column list whereby terms are listed on the left and their respective definitions appear opposite them on the right).

Example
```
<DL>
<DT>Cat
<DD>A clever furred animal who purrs and loves milk.
<DT>Lizard
<DD>A mysterious desert animal with a long tongue.
</DL>
```

Component
No direct analog. Programmers can use the HTMLTag component.

<DT>

Description

Indicates that this is a definitions list term. Indicates that the text is a term to be defined, and must therefore be output in the left-hand column of the definitions list.

Example
```
<DL> <DT>Cat<DD>A clever furred animal who purrs and loves milk.
<DT>Lizard<DD>A mysterious desert animal with a long tongue.
</DL>
```

<EM>

Description

Marks up the text, usually by presenting it in italic face.

Example
<EM>This text will print in italics.</EM>

Component
No direct analog. Programmers can use the HTMLTag component.

<EMBED HEIGHT=size of object NAME=programmatic name OPTIONAL PARAM=&quot;value&quot; ... OPTIONAL PARAM= PALETTE=foreground | background SRC=data to object WIDTH=size of object>

Description

Indicates that the object is an embedded one. Although OBJECT is a more preferable option for inserting objects, EMBED has been included for compatibility with the earlier versions of HTML documents. See description of OBJECT.
HEIGHT=***size of object***
Defines the height of the object on the page in pixels.
NAME=***programmatic name***
Specifies the name by which other objects or elements would reference the given object.
OPTIONAL PARAM=***value***
Indicates the optional parameters (if any) specific for the given object.
PALETTE=***foreground | background;***
Sets the foreground or background color palette.
SRC=***data to object***
Specifies the name of any source of code to be added to the object.
WIDTH=***size of object***
Specifies the width of the object on the page in pixels.

Example
<EMBED SRC=&quot;MyMovie.AVI&quot; WIDTH=100 HEIGHT=250 AUTOSTART=TRUE PLAYBACK=FALSE&gt;</code></font></font></pre>

Component
No direct analog. Programmers can use the HTMLTag component.

<FONT SIZE=*n* FACE=*name* COLOR=*color*>

Description

Sets font, size and color of the text.
COLOR=***color***
Sets the font color. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.
FACE=***»name [,name2[,name3]]»***
Sets the font face type. You can specify a list of font names. If the very first font on the list is accessible, the system will use it. Otherwise the system will attempt to use the second font on the list, etc. If none of the fonts on your list is accessible, the default font will be used.
SIZE=***n***
Specifies the font size. This is a number from 1 to 7, with 7 corresponding to the largest font size. The "plus" or "minus" sign prepended to the number specifies the font size relative to the base font's current setting (BASEFONT). Note that relative font size specifications are not cumulative, and therefore specifying <FONT SIZE=»+1"> twice in one line will not increase the font size by 2.

Component
Automatically generated for all text elements. Programmers can also use the HTMLTag component.

<FORM ACTION=*url* METHOD=*get-post* TARGET=*window*>

Description

Denotes a form.
ACTION=***url***
Specifies the address to be used for posting the results of actions with the form. If nothing is specified, the document's base URL will be used.
METHOD=***get-post***
Indicates how the form's data are to be posted to the server. Valid value for *get-post* can be one of the following:

GET	Adds arguments to the current URL and opens it just as if it were an anchor.
POST	Posts data via the HTTP post transaction.

TARGET=***window***
See above.

Example
<FORM TARGET=»viewer» ACTION=»http://www.sample.com/bin/search»>
...
</FORM>
Component
Automatically generated by the HTMLPage component.

<FRAME ALIGN=*align-type* FRAMEBORDER=*1*|0 MARGINHEIGHT=*height* MARGINWIDTH=*width* NAME=*name* SCROLLING=*yes*|no SRC=*address*>

Description

Describes one frame in a frame set. There is no corresponding closing tag, and it is not a container.
ALIGN=***align-type***
Sets the alignment type for the frame or surrounding text. Valid value for *align-type* can be one of the following:

TOP	The surrounding text is aligned to the top of the frame.
MIDDLE	The surrounding text is aligned to the middle of the frame.
BOTTOM	The surrounding text is aligned to the bottom of the frame.
LEFT	The frame is drawn as a left-aligned «floating frame», with the text placed around it.
RIGHT	The frame is drawn as a right-aligned «floating frame», with the text placed around it.

FRAMEBORDER=***0|1***
Indicates whether a 3D-style border is to be drawn around the frame. The default value is 1 (add border). 0 means that the frame is to have no border around it.
MARGINHEIGHT=***height***
Controls the height of the frame margin in pixels.
MARGINWIDTH**=*width***
Controls the width of the frame margin in pixels.
NAME**=*name* - name**
Provides a name for the frame.
NORESIZE
Disallows resizing of the frame by the user.

SCROLLING=***yes|no***
Creates a scrollable frame.
SRC=***address***
Specifies the address of the frame’s source text.

Example
<FRAME FRAMEBORDER=0 SCROLLING=NO SRC=»sample.htm»>

Component
Not implemented in this version of the library.

<FRAMESET COLS=*col-widths* FRAMEBORDER=*1*|0 FRAMESPACING=*spacing* ROWS=*row-heights*>

Description

A container that encapsulates the FRAME, FRAMESET and NOFRAMES elements.
COLS=***col-widths***
Creates a column-style frame. You can define column width in terms of percent (%), pixels or relative size (*).
FRAMEBORDER=***1|0***
Indicates whether a 3D-style border is to be drawn around the frame. The default value is 1 (draw frame). 0 means that the frame is to have no border around it.
FRAMESPACING=***spacing***
Defines the size (in pixels) of the additional spacing to be created between successive frames.
ROWS=***row-heights***
Creates a row-style document frame. You can define row height in terms of percent (%), pixels or relative size (*).
The FRAMEBORDER= and FRAMESPACING= attributes are inherited from any FRAMESET container element. This means that you only have to specify attributes for a single (outermost) FRAMESET tag to have them apply to all FRAME tags on the same page.

Example

```
<FRAMESET SCROLLING=YES COLS=»25%, 50%, *»>
   <FRAME SRC=»contents.htm»>
   <FRAME SRC=»info.htm»>
  <FRAME SCROLLING=NO SRC=»graphic.htm»>
</FRAMESET>
```

Component
Not implemented in this version of the library.

<Hn ALIGN=align-type>

Description

Presents the text in the header style. You can use values H1 through H7 to specify various header sizes and styles.
n
Sets the section level. This is an integer number from 1 to 6.
ALIGN=***align-type***
Sets the alignment type for the header text. Valid value for *align-type* is CENTER, LEFT or RIGHT. The default value is LEFT.
Use of closing tag is mandatory.

Example
<H1>Welcome to Internet!
</H1>

Component
Implemented in the HTMLHeader component. The header font size is determined by the HeaderLevel property.

<HEAD>

Description

Marks up the text as the HTML document's header.

Example
<HEAD>
<TITLE>A simple document</TITLE>
</HEAD>

Component
Automatically generated by the HTMLPage component.

<HR ALIGN=*align-type* COLOR=*color* NOSHADE SIZE=*n* WIDTH=*n*>

Description

Draws a horizontal line.
ALIGN=***align-type***
Draws a left-aligned, right-aligned or centered line. Valid value for *align-type* is LEFT, RIGHT or CENTER.

COLOR=***color***
Sets the color of the line of text. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.
NOSHADE
Draws a line without 3D-style shading.
SIZE=***n***
Sets the height of the line in pixels.
WIDTH=***n***
Sets the width of the line either in pixels or in percent of window width.

Example
<HR SIZE=5 WIDTH=80% NOSHADE>

Component
HTMLRuler. The appearance is determined by the Shade property. The horizontal and vertical sizes are calculated automatically.

<HTML>

Description

Marks up the file as an HTML document. This element has no attributes.

Example
<HTML><BODY> <P>This is an HTML document.</BODY> </HTML>

Component
Generated automatically by the HTMLPage component.

<I>

Description

Presents the text in italic face.

Example
<I>This text will be printed in italics.</I>

Component
Use the Font property for text components.

<IFRAME ALIGN=*align-type* FRAMEBORDER=*1*|0 MARGINHEIGHT=*height* MARGINWIDTH=*width* NAME=*name* SCROLLING=*yes*|no SRC=*address*>

Description

Specifies a floating frame.

ALIGN=***align-type***

Sets the alignment type for the frame or surrounding text. Valid value of *align-type* is one of the following:

TOP	The surrounding text is aligned to the top of the frame.
MIDDLE	The surrounding text aligned to the middle of the frame.
BOTTOM	The surrounding text aligned to the bottom of the frame.
LEFT	The frame is drawn as a left-aligned «floating frame», with the text placed around it.
RIGHT	The frame is drawn as a right-aligned «floating frame», with the text placed around it.

FRAMEBORDER=***0|1***

Indicates whether a 3D-style border is to be drawn around the frame. The default value is 1 (add border). 0 means that the frame is to have no border around it.

MARGINHEIGHT=***height***

Controls the frame height.

MARGINWIDTH=***width***

Controls the frame width.

NAME=***name***

Describes the frame name.

NORESIZE

Disallows resizing of the frame by the user.

SCROLLING***=yes|no***

Creates a scrollable frame.

SRC***=address***

Outputs the address of the frame’s source text.

Example

<IFRAME FRAMEBORDER=0 SCROLLING=NO SRC=»sample.htm»>

Component

Not implemented in this version of the library.

<IMG ALIGN=*align-type* ALT=*text* BORDER=*n* CONTROLS DYNSRC=*url* HEIGHT=*n* HSPACE=*n* ISMAP LOOP=*n* SRC=*address* START=*start-event* USEMAP=*map-name* VSPACE=*n* WIDTH=*n*>

Description

Inserts an image.
ALIGN=***align-type***
Sets the alignment type for the image or surrounding text. Valid value for *align-type* can be one of the following:

TOP	The surrounding text is aligned to the top of the image.
MIDDLE	The surrounding text aligned to the middle of the image.
BOTTOM	The surrounding text aligned to the bottom of the image.
LEFT	The picture is drawn as a left-aligned «floating image», with the text placed around it.
RIGHT	The picture is drawn as a right-aligned «floating image», with the text placed around it.

ALT=***text***
Specifies the text to be output in place of the picture if the *Show Pictures* option is unchecked.
BORDER=***n***
Determines the size of the border to be drawn around the image. If the image is a hyperlink pointer, the border is drawn in the corresponding color of that hyperlink pointer. If the image is not a hyperlink pointer, the border is invisible.
CONTROLS
If a video clip is available, a set of controls is output under it.
DYNSRC***=url***
Specifies the address of the video clip or VRML to be output in the window. It is used instead of the Dynamic Source.
HEIGHT=***n***
Determines, in conjunction with WIDTH=, the dimensions of the picture to be drawn. If the picture's actual dimensions differ from those specified, the picture is stretched so that its dimensions would correspond to what is specified. Internet Explorer also uses this to draw an image occupying the area of corresponding dimensions before loading the picture.
HSPACE=***n***
Determines, in conjunction with VSPACE=, the horizontal spacing for the image.

HSPACE= is similar to BORDER= except that the margins are not painted in a definite color when the image is a hyperlink pointer.

ISMAP

Clicking on the picture results in the mouse click coordinates being passed back to the server, which takes you to another page.

LOOP=***n***

Determines how many cycles of the video clip will be played back once it is activated. If *n*=1 or LOOP=INFINITE is specified, the clip will loop indefinitely.

SRC=*address*

Specifies the source address of the picture to be inserted.

START=***start-event***

Indicates when playback of the file specified as DYNSRC=*attribute* should start. Valid value for *start-event* can be one or both of the following:

FILEOPEN	Start playback immediately after the file is opened. This is a default value.
MOUSEOVER	Start playback when the user moves the mouse cursor over the animation.

You can specify both values, but they should be separated with a comma.

USEMAP=***map-name***

Determines the MAP to be used in performing the operations corresponding to the user-generated mouse clicks.

VSPACE=***n***

Determines, in conjunction with HSPACE=, the image spacing margins. VSPACE= is similar to BORDER=, except that the margins are not painted a definite color when the image is a hyperlink pointer.

WIDTH=***n***

Determines, in conjunction with HEIGHT=, the dimensions of the picture to be drawn. If the picture's actual dimensions differ from the specified ones, the picture is stretched so that it would eventually correspond to what is specified. Internet Explorer also uses this to draw an image occupying the area of corresponding dimensions before loading the picture.

Component

HTMLImageRef, HTMLImageRes, HTMLDBGIF. The image alignment type is determined by the AlignImage property. The alternative name is specified in the AltName property. The SRC is generated automatically. Support of other attributes is not implemented in this version of the library.

<INPUT ALIGN=*align-type* [CHECKED|] MAXLENGTH=*length* NAME=*name* SIZE=*size* SRC=*address* TYPE=*type* VALUE=*value*>

Description

Determines the form's control.

ALIGN=**align-type**

Used if TYPE=IMAGE is specified. Determines how the next line of text will be aligned relative to the image. Valid value for *align-type* is TOP, MIDDLE or BOTTOM.

CHECKED

Activates the CHECKED option to establish that the on/off or radio button will appear «checked» (selected) when the form is initially loaded.

MAXLENGTH=***length***

Indicates the maximum number of characters that can be input into the control.

NAME=***name***

Specifies the name of the control.

SIZE=***size***

Specifies the size of the control (in characters). For controls of TEXTAREA type, both height and width parameters can be specified if the «*width*,*height*» format is used.

SRC=***address***

Specifies the source address of the image to be used. It is employed if TYPE=IMAGE is specified.

TYPE=***type***

Indicates what type of control is to be used:

CHECKBOX	Used for simple Boolean attributes or attributes that can assume more than one value at a time. This control is presented in the form of several fields of on/off buttons (checkboxes), each of which has the same name. Each selected on/off button in the presented data generates a separate name/value pair, even if this results in duplicate names. The default value for on/off buttons is «on» («checked»).
HIDDEN	No field is made visible to the user, but the field's contents are passed together with the form to be executed. This value can be used for passing information on the status of the client/server interface.
IMAGE	The image field on which you can click to have it immediately activated. The selected point's coordinates are mapped and measured in pixels relative to the top left-hand corner of the image and are returned (together with the rest of the form's contents) in two name/value pairs. The *x*-coordinate is passed in the field name

	appended with «.x», while the *y*-coordinate is passed in the field name appended with «.y». All VALUE attributes are ignored. Parameters of the image proper are determined by the SRC attribute in exactly the same way as for the Image element.
PASSWORD	Similar to the TEXT attribute, except that the text input by the user is not displayed.
RADIO	Used for those attributes which assume one out of a set of alternative values. Each field of radio buttons in a group should be assigned a unique name. Only the selected radio button in the group generates a name/value pair in the presented data. Radio buttons require that the VALUE attribute be explicitly specified.
RESET	A button that, when clicked, resets the form's fields to their originally defined values. The glyph to be displayed on this button can be specified in the same way as for the SUBMIT button.
SUBMIT VALUE	A button that, when clicked, activates the form. You can use the attribute to have a non-editable glyph to be displayed on the button. The default glyph is application-specific. If the SUBMIT button is clicked in order to activate a form and has a certain NAME attribute, then this button produces a name/value pair in the posted data. Otherwise, the SUBMIT button contributes nothing to the posted data.
TEXT	Employed for single-line text input fields. Use this jointly with the SIZE and MAXLENGTH attributes.

The default control type is TEXT.

VALUE=***value***

Specifies the default value of the control (for text/numeric controls). If the control is of Boolean type, determines the value to be returned when such a control is activated.

Example

```
<FORM ACTION=»http://intranet/survey» METHOD=POST>
<P>Name
<BR><INPUT NAME=»CONTROL1" TYPE=TEXT VALUE=»Your Name»>
<P>Password
<BR><INPUT TYPE=»PASSWORD» NAME=»CONTROL2">
<P>Color
<BR><INPUT TYPE=»RADIO» NAME=»CONTROL3" VALUE=»0" CHECKED>Red
<INPUT TYPE=»RADIO» NAME=»CONTROL3" VALUE=»1">Green
<INPUT TYPE=»RADIO» NAME=»CONTROL3" VALUE=»2">Blue
<P>Comments
<BR><INPUT TYPE=»TEXTAREA» NAME=»CONTROL4" SIZE=»20,5"
MAXLENGTH=»250">
<P><INPUT NAME=»CONTROL5" TYPE=CHECKBOX CHECKED>Send receipt
<P><INPUT TYPE=»SUBMIT» VALUE=»OK»><INPUT TYPE=»RESET»
VALUE=»Reset»>
</FORM>
```

Components

This tag generates such components as HTMLButton (this control will have the type

SUBMIT if the ButtonType property is set to btSubmit, or the type RESET otherwise), HTMLCheckBox (CHECKBOX), HTMLRadio(RADIO), HTMLImageButton (IMAGE), HTMLEdit (this control will have the type TEXT if the Password property is set to False, and the type PASSWORD otherwise), HTMLHidden (HIDDEN), HTMLDBEdit (see HTMLEdit), and HTMLDBCheckBox (see HTMLCheckBox).

<ISINDEX ACTION=*url* PROMPT=*prompt-text*>

Description

Indicates whether a search index is specified.
ACTION=***url***
Determines the CGI (Common Gateway Interface) program which the line in the text input field should be passed to.
PROMPT=***prompt-text***
Determines the custom prompt to be used instead of the default prompt.
If PROMPT=*attributes* is omitted, the element outputs the following message followed by a text input field: «You can start the index search. Please enter the search keyword(s)». Once the user enters a text and presses ENTER, this text will be sent back to the page's URL as a request.

Example
<ISINDEX ACTION=»http://intranet/search» PROMPT=»Enter key words here…»>

Component
Not implemented in this version of the library.

<KBD>

Description
The text to be entered from keyboard. The text is printed in fixed-width bold-face font.
Example
<KBD>This user must enter this text.</KBD>
Component
No direct analog. Programmers can use the HTMLTag component.

<LI TYPE=*order-type* VALUE=*n*>

Description

Denotes a list item. In the DIR, MENU, OL, or UL blocks of text denotes a new item of the list.

TYPE=*order-type*
Changes style of an ordered list. Valid values for *order-type* are as follows:

A	Use capital (upper-case) letters.
a	Use lower-case letters.
I	Use Roman numerals.
i	Use lower-case Roman numerals.
1	Use numbers.

VALUE=***n***
Changes the order hierarchy of the list items.
Example
<DIR>
<LI>Art
<LI>Hystory
<LI>Literature
<LI>Sports
<LI>Intertainment
<LI>Science </DIR>
Component
See description of the <UL> tag.

<LINK HREF=*URL*>

Description

The LINK element establishes the hierarchy of navigation through various documents. LINK should be part of the HEAD element. The HEAD element can encapsulate more than one LINK elements.
HREF=***URL***
Determines the URL referenced to the current document.

Example
<LINK HREF=»http://www.microsoft.com/newdocnewdoc.htm»>

Component
Usually needs not be used. Programmers can use the AddHeader property of the HTMLPage component to place text into the header.

<LISTING>

Description

Presents the text in fixed-width font.

Example
<LISTING>Here's some plain text.</LISTING>

Component
No direct analog. Programmers can use the HTMLTag component.

<MAP NAME=*name*>

Description

Maps the active areas within a picture.
NAME=***name***
Gives the MAP object a name by which it could be subsequently referenced.

Example
```
<MAP NAME=»map1">
    <AREA ... >
    <AREA ... >
</MAP>
```

Component
No direct analog. Programmers can use the HTMLTag component.

<MARQUEE ALIGN=*align-type* BEHAVIOR=*type* BGCOLOR=*color* DIRECTION=*direction* HEIGHT=*n* HSPACE=*n* LOOP=*n* SCROLLAMOUNT=*n* SCROLLDELAY=*n* VSPACE=*n* WIDTH=*n*>

Description

Creates a marquee (“scrolling-text”) window.
ALIGN=***align-type***
Determines how the surrounding text is to be aligned relative to the marquee window. Valid value for *align-type* can be one of the following:

TOP	The surrounding text is aligned to the top of the window.
MIDDLE	The surrounding text aligned to the middle of the window.
BOTTOM	The surrounding text aligned to the bottom of the window.

BEHAVIOR=***type***

Determines the mode in which the object will be operating. Valid value for *type* is one of the following:

SCROLL	The text begins entirely outside the marquee window on one side and scrolls through completely until it is fully outside the window on the opposite side, after which the cycle is repeated. This is the default mode.
SLIDE	The text begins entirely outside the marquee window on one side and slides across the window until it touches the opposite margin.
ALTERNATE	The text bounces back and forth within the marquee window.

BGCOLOR=***color***

Determines the background color of the marquee window. The value of *color* can be specified either as a hexadecimal number (optionally prefixed with the symbol #), in the RGB format, or as a redefined color name. See description of Color.

DIRECTION=***direction***

Determines the direction in which the text will be scrolling. Valid value for *direction* is either LEFT or RIGHT. The default value is LEFT (text scrolls from right to left).

HEIGHT=***n***

Determines the height of the marquee window, either in pixels or as percent of the screen height. In the latter case, the *n* value should be followed with the percent character (%).

HSPACE=***n***

Determines the size of the marquee window‘s right- and left-hand margins in pixels.

LOOP=***n***

Determines the number of scroll cycles through the window after scrolling is activated. If *n*=-1 or if LOOP=INFINITE is specified, text scrolling will loop indefinitely.

SCROLLAMOUNT=***n***

Determines the amount of scroll, in pixels, between each successive output of text in the marquee window.

SCROLLDELAY=***n***

Determines the scroll delay, in milliseconds, between each successive output of text in the marquee window

VSPACE=***n***

Determines the size, in pixels, of the margins above and below the marquee window.

WIDTH=***n***

Determines the width of the marquee window, either in pixels or in percent of the screen width. In the latter case, the *n* value should be followed with the percent character (%).

Example

```
<MARQUEE DIRECTION=RIGHT BEHAVIOR=SCROLL SCROLLAMOUNT=10
SCROLLDELAY=200>This is a line of «scrolling» text.</MARQUEE>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<MENU>

Description

Denotes a menu. Implies that the following block of text consists of individual items each of which begins with the LI element.

Example
<MENU><LI>This is menu item #1.<LI>And this is menu item #2.
</MENU>

Component
No direct analog. See description of the <UL> tag.

<META HTTP-EQUIV=*response* CONTENT=*description* NAME=*description* URL=*url*>

Description

Presents information about HTML documents.
HTTP-EQUIV=***description***
Links item to the HTTP header. This information is subsequently used by the application when it reads the header. See examples below.
CONTENT=***description***
Determines the contents of the meta information that must be linked to the specified name or HTTP header. Can be used together with *URL*= and date/time specification to load the document repeatedly after expiration of a certain time period.
NAME=***description***
Describes the name of the document.
URL=***description***
Describes the URL of the document.

Examples
If the document contains the text:
<META HTTP-EQUIV=»Expires»
   CONTENT=»Tue, 04 Dec 1996 21:29:02 GMT»>
<meta http-equiv=»Keywords» CONTENT=»HTML, Reference»>
<META HTTP-EQUIV=»Reply-to»   content=»anybody@microsoft.com»>
<Meta Http-equiv=»Keywords» CONTENT=»HTML Reference Guide»>
the server may include header fields
Expires: Tue, 04 Dec 1996 21:29:02 GMT
Keywords: HTML, Reference
Reply-to: anybody@microsoft.com
as part of the HTTP response to the “GET” or “HEAD” queries about this document.

```
<ex><HTML>
<HEAD>
<META HTTP-EQUIV=»REFRESH» CONTENT=2>
<TITLE>Re-load the document</TITLE>
</HEAD>
<BODY>
<P>This document will be re-loaded every 2 seconds.
</BODY>
</HTML>

<HTML>
<HEAD>
<META HTTP-EQUIV=»REFRESH» CONTENT=»5; URL=http://www.sample.com/
next.htm»>
<TITLE>Load next document.</TITLE>
</HEAD>
<BODY>
<P>After expiration of five seconds the document «http://www.sample.com/next.htm»
will be loaded.
</BODY>
</HTML>
```

Component
Programmers can use the AddHeader property of the HTMLPage component to add the necessary information to the header.

<NOBR>

Description

Disables line breaks. Presents the text without line breaks.

Example

```
<NOBR>Here is a line of text which I don’t want to break . . . This is the end of the
line.
</NOBR>
```

Component
No direct analog. Programmers can use the HTMLTag component.

<NOFRAMES>

Description

Contents that can be viewed only by browsers which have no frame-support feature. Browsers that do support frames will not output anything contained between the opening and closing tags of NOFRAMES. Using NOFRAMES, you can create a page compatible with browsers of both types.

Example
<FRAMESET>
<NOFRAMES>You will need a browser supporting HTML 3.2 to view frames!
</NOFRAMES>
</FRAMESET>

Component
Not implemented in this version of the library.

<OBJECT ALIGN=*align-type* BORDER=*n* CLASSID=*url* CODEBASE=*url* CODETYPE=*codetype* DATA=*url* DECLARE HEIGHT=*n* HSPACE=*n* NAME=*url* SHAPES STANDBY=*message* TYPE=*type* USEMAP=*url* VSPACE=*n* WIDTH=*n*>

Description

Inserts a specific object (such as an image, a document, an applet or a control) into the HTML document.
ALIGN=***align-type***
Sets the object's alignment type. Valid values for *align-type* are as follows:

BASELINE	The object's bottom is aligned to the base line of the surrounding text.
CENTER	The object is centered between the left- and right-hand margins; subsequent text begins on the very first line following the object.
LEFT	The object is aligned to the left-hand margin; subsequent text is wrapped along the object's right-hand edge.
MIDDLE	The object's middle is aligned to the surrounding text's base line.
RIGHT	The object is aligned to the right-hand margin; subsequent text is

	wrapped along the object’s left-hand edge.
TEXTBOTTOM	The object’s bottom is aligned to the bottom of the surrounding text.
TEXTMIDDLE	The object’s middle is aligned to a middle point between the base line and x-height of the surrounding text.
TEXTTOP	The object’s top is aligned to the top of the surrounding text.

BORDER=***n***

Determines the width of the object’s border, if the object has been defined as a hyperlink pointer.

CLASSID=***url***

Identifies the object’s implementation. The *url* syntax depends on the object type. For example, for ActiveX registered controls the syntax is as follow: CLSID:*class-identifier*.

CODEBASE=***url***

Identifies the object’s code base. The *url* syntax is object-dependent.

CODETYPE=***codetype***

Determines the code media type.

DATA=***url***

Identifies the data for the object. The *url* syntax is object-dependent.

DECLARE

Declares the object without implementing it. Use DECLARE further in the document to create cross references to the object or when you use the object as a parameter in another object.

HEIGHT=***n***

Specifies the suggested height of the object.

HSPACE=***n***

Determines the horizontal spacing (an additional unfilled space between the object and any text or images to the right or left of it).

NAME=***url***

Establishes the name of the object when the latter is posted as part of a form.

SHAPES

Indicates that the object has a hyperlink pointer.

STANDBY=***message***

Describes the message to be displayed while the object is being loaded.

TYPE=***type***

Determines the data media type.

USEMAP=***url***

Determines the picture to be used jointly with the object.

VSPACE=***n***

Determines the vertical spacing (an additional unfilled space between the object and any text or images above or below of it).

WIDTH=***n***

Specifies the suggested width of the object.

Use of closing tag is mandatory.

The object can encapsulate any elements normally used within the body of an HTML document, including section headers, paragraphs, lists, forms, and even embedded objects.

Component
No direct analog. Programmers can use the HTMLTag component.

<OL START=*n* TYPE=*order-type*>

Description

Outputs lines of text in the form of an ordered list. Indicates that the following block of text is made up of individual items, each of which begins with the LI tag. The list's items are numbered.
START=***n***
Specifies the starting number in the list.
TYPE=***order-type***
Changes style of an ordered list. Valid values for *order-type* are as follows:

A	Use capital (upper-case) letters.
a	Use lower-case letters.
I	Use Roman numerals.
i	Use lower-case Roman numerals.
1	Use numbers.

Example
```
<OL>
<LI>This is item #1 on the list.
<LI>And this is item #2 on the list.
</OL>

<ex><OL START=3>
<LI>This is item #3.
</OL>

<OL TYPE=A>
<LI>This is item A.
</OL>
```

Component
See description of the <UL> tag.

<SELECT SELECTED VALUE=*value*>

Description

Denotes the selection of a list item.
SELECTED
Specifies the item selected by default. If nothing is specified, item #1 becomes the

default item.
VALUE=***value***
Specifies the value to be returned if the given element is selected.

<P ALIGN=*align-type*>

Description

Denotes a paragraph. Inserts an end-of-paragraph symbol and signifies the beginning of a next paragraph.
ALIGN=***align-type***
Establishes the alignment type for the paragraph. Valid value for *align-type* is CENTER, LEFT or RIGHT. The default alignment type is LEFT.
Use of closing tag is optional.

Example
<P>This is a paragraph.</P>

Component
No direct analog. Programmers can use the HTMLTag component.

<PARAM NAME=*name* VALUE=*value* VALUETYPE=*type* TYPE=*type*>

Description

Establishes property parameters for the specified object.
NAME=***name***
Sets name of the property.
VALUE=***value***
Sets the property's value. This value is passed to the object unchanged, except that any strings of alphabetic or numeric characters are replaced by their respective values.
VALUETYPE=***type***
Determines how the value is to be interpreted. Valid value for *type* can be one of the following:

DATA	The value type is data. This is the default value type.
REF	The value type is the URL.
OBJECT	The value type is the URL of an object in the same document.

TYPE=***type***
Specifies the Internet media type.
This element is valid only within the OBJECT element. Use of closing tag is optional.

<PLAINTEXT>

Description

Presents text in fixed-width font without tag processing. Also disables HTML parsing until the browser encounters the </PLAINTEXT> tag.

Example
<PLAINTEXT>Here is an HTML sample: <A HREF=»sample.url»>This is a shortcut to the sample.
</A></PLAINTEXT>

Component
No direct analog. Programmers can use the HTMLTag component.

<PRE>

Description
Presents text in fixed-width font.
Example
<PRE>Here is some plain text.</PRE>
Component
You should use the *Preformat* property for text components.

<S>

Description

Presents text in strike-out face.

Example
<S>This text is struck out.</S>

Component
You should use the *Font* property for text components.

<SAMP>

Description

Determines the text of the sample. Presents text in small-size font (if FONT FACE is not specified, the sample text is presented in fixed-width font).

Example
<SAMP>Here is some text in small-size fixed-width font.
</SAMP>

Component
No direct analog. Programmers can use the HTMLTag component.

<SCRIPT LANGUAGE=*scripting language*>

Indicates that a script is included. Scripts are executed and activated in exactly the same order in which they appear in the HTML document. Named objects can be referenced only in the order these objects appear in the document.
LANGUAGE=***scripting language***
Specifies language in which the script is created. Examples of such languages are «VBScript» and «JavaScript».

Example
<SCRIPT>
<SCRIPT language=»VBScript»>
</SCRIPT>

Component
In this version of the library, it is represented by the TJavaScript component exclusively to ensure support of Java scripts.

<SELECT MULTIPLE NAME=*name* SIZE=*n*>

Description

Denotes a view list or a drop-down list.
MULTIPLE
Indicates that more than one item can be selected at a time.
NAME=***name***
Assigns a name to the list.
SIZE=***n***
Determines the height of the list.

Example
```
<SELECT NAME=»Cars» MULTIPLE SIZE=»1">
   <OPTION VALUE=»1">BMW
  <OPTION VALUE=»2">PORSCHE
  <OPTION VALUE=»3" SELECTED>MERCEDES
</SELECT>
```

Components
HTMLListBox, HTMLComboBox.

<SMALL>

Description

Makes the text one size smaller.

Example
<SMALL>This text is of a smaller size. </SMALL>

Component
No direct analog. Programmers can use the HTMLTag component.

<SPAN STYLE= *spanstyle*>

SPAN is used to specify style information within the document. SPAN can be used for local text formatting with the use of STYLE as the attribute.

Example
<SPAN STYLE=»margin-left: 1.0in»>This paragraph is located 1.0 inch from the left-hand margin.
</SPAN>

Component
No direct analog. Programmers can use the HTMLTag component.

<STRIKE>

Description

Presents text in strike-out face.

Example
<STRIKE>This text is struck out.</STRIKE>

Component
No direct analog. Programmers can use the HTMLTag component.

<STRONG>

Description

Highlights the text, normally presenting it in bold face.

Example
<STRONG>This text will be printed in bold face.</STRONG>

Component

No direct analog. Programmers can use the HTMLTag component.

<SUB>

Presents the text in subscript.

Example
<SUB>This text will appear as subscript test.
</SUB>

Component
No direct analog. Programmers can use the HTMLTag component.

<SUP>

Description

Presents the text in superscript.

Example
<SUP>This text will appear as superscript.
</SUP>

Component
No direct analog. Programmers can use the HTMLTag component.

<TABLE ALIGN=*align-type* BACKGROUND=*url* BGCOLOR=*color* BORDER=*n* BORDERCOLOR=*color* BORDERCOLORDARK=*color* BORDERCOLORLIGHT=*color* CELLPADDING=*n* CELLSPACING=*n* COLS=*n* FRAME=*frame-type* RULES=*rules* WIDTH=*n* >

Description

Creates a table. The table is assumed to be empty unless you create rows and cells for it with the aid of the TR, TD and TH elements.
ALIGN=***align-type***
Determines the alignment type for the table. Valid value for *align-type* can be one of the following:

LEFT	The table is left-aligned. This is the default alignment type.
RIGHT	The table is right-aligned. If the table is narrower than the window, the text following the table will wrap along the table's left-hand edge.

BACKGROUND=***url***

Determines the background picture. The picture is arranged tile-style behind any text and graphics in the table, its header or cell.

BGCOLOR=***color***

Sets the background color. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDER=***n***

Specifies the size, in pixels, of the table's border. The default value of Border is zero.

BORDERCOLOR=***color***

Sets the color of the table's border. Should be used in conjunction with the BORDER attribute. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDERCOLORLIGHT=***color***

BORDERCOLORDARK=***color***

Set the drawing color for the table's 3D border.

CELLPADDING=***n***

The distance, in pixels, between the sides of a cell and its contents.

CELLSPACING=***n***

The distance, in pixels, between the border (outer part) of the table and its cells.

COLS=***n***

The number of columns in the table. If specified, this attribute can speed up the processing of tables (especially long ones).

WIDTH=***n***

Sets the width of the table in pixels or as percent of the window width. In the latter case, the *n* value should be followed with the percent character (%).

FRAME=***frame-type***

Determines which sides of the table's frame (outer borders) will be output. Valid value for *frame-type* can be one of the following:

VOID	Removes all outer borders of the table.
ABOVE	Outputs a border in the top part of the table frame.
BELOW	Outputs a border in the bottom part of the table frame.
HSIDES	Outputs a border in the top and bottom parts of the table frame.
LHS	Outputs a border on the left-hand side of the table frame.
RHS	Outputs a border on the right-hand side of the table frame.
VSIDES	Outputs a border on the left- and right-hand sides of the table frame.
BOX	Outputs a border on all sides of the table frame.
BORDER	Outputs a border on all sides of the table frame.

RULES=***rule-type***

Determines what rules (inner borders) of the table will be output. Valid value for *rule-type* can be one of the following:

NONE	Removes all inner borders of the table.
GROUPS	Outputs horizontal borders between all groups in the table. The groups are specified with the aid of the THEAD, TBODY, TFOOT and COLGROUP elements.
ROWS	Outputs horizontal borders between all rows of the table.
COLS	Outputs vertical borders between all columns of the table.
ALL	Outputs a border between all rows and columns of the table.

Use of closing tag is mandatory.
The THEAD, TBODY, TFOOT, COLGROUP and COL elements are optional, and can be used to format the table and change individual characteristics of its columns and column groups.

Example
```
<TABLE BORDER=1 WIDTH=80%>
<THEAD>
<TR>
  <TH>Heading 1</TH>
  <TH>Heading 2</TH>
</TR>
<TBODY>
<TR>
  <TD>Row 1, Column 1 text.</TD>
  <TD>Row 1, Column 2 text.</TD>
</TR>
<TR>
  <TD>Row 2, Column 1 text.</TD>
  <TD>Row 2, Column 2 text.</TD>
</TR>
</TABLE>
```

Component
In this version of the library, the HTMLTable component implements table formatting capabilities only to a limited extent (you cannot merge cells, for instance). Table width is determined by the *WidthOnPage* property. Border width is determined by the *Border* property.

<TBODY>

Description

If the table has no header or footnote (no THEAD or TFOOT element), then the TBODY element is optional. Use of closing tag is always optional.
You can use the TBODY element in a table more than once. This feature proves to be useful for breaking long tables into shorter parts as well as for controlling the arrangement of horizontal lines.

Example

```
<TABLE>
<THEAD>
<TR>
    ...
</TR>
<TBODY>
<TR>
    ...
</TR>
</TBODY>
</TABLE>
```

<TD ALIGN=*align-type* BACKGROUND=*url* BGCOLOR=*color* BORDERCOLOR=*color* BORDERCOLORLIGHT=*color* BORDERCOLORDARK=*color* COLSPAN=*n* NOWRAP=NOWRAP ROWSPAN=*n* VALIGN=*align-type*>

Description

Creates a table cell.

ALIGN=***align-type***

Determines the horizontal alignment type for the cell text. Valid value for *align-type* can be one of the following:

LEFT	The cell text is left-justified.
CENTER	The cell text is centered.
RIGHT	The cell text is right-justified.

By default, the text is centered.

BACKGROUND=***url***

Determines the background picture. The picture is arranged tile-style behind any text and graphics in the table, its header or cell.

BGCOLOR=***color***

Sets the background color. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDERCOLOR=***color***

Sets the color of the border. Should be used jointly with the BORDER attribute. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDERCOLORLIGHT=***color***

BORDERCOLORDARK=***color***

Set the drawing color for the 3D border.

BORDER=***n***

Specifies the size, in pixels, of the table border. The default value of Border is zero.
VALIGN=***align-type***
Determines the vertical alignment type for the cell text. Valid value for *align-type* can be one of the following:

TOP	The text is aligned to the top of each cell.
MIDDLE	The text is aligned to the middle if each cell.
BOTTOM	The text is aligned to the bottom of each cell.
BASELINE	Text in adjacent cells of the same row is aligned to a common base line.

By default, text is aligned to the middle of each cell.
This element is valid only within one row of the table. This implies that you must use the TR elements before using the TD element. All attributes are optional. Use of closing tag is likewise optional.

<TEXTAREA COLS=*n* NAME=*name* ROWS=*n*>

Description

Creates a control for a multiple-line input area in which the user can input and edit text.
COLS=***n***
Sets the width of the text input/edit area in characters.
NAME=***n***
Establishes the name of the text input/edit area. The name is used when this element is employed within a FORM element.
ROWS=***n***
Sets the height of the text input/edit area in characters.
Use of closing tag is mandatory. Any text enclosed between the opening and closing tags is used as the control's default value(s).

Component
The tag is represented by the HTMLMemo and HTMLDBMemo components. Dimensions are specified with the aid of the Cols and Rows properties.

<TFOOT>

Marks up the text as a table footnote. Use this element to separate lines of text in the footnote to the table form lines of text in the table header or lines of text in the table body. A table footnote is optional but, if specified, only one footnote is allowed. The TFOOT element is valid only within the relevant table. You must use the TABLE element before using this element. Use of closing tag is optional.

Example

```
<TABLE>
<TBODY>
   <TR>
    ...
   </TR>
<TFOOT>
   <TR>
    ...
   </TR>
</TABLE>
```

<TH ALIGN=*align-type* BACKGROUND=*url* BGCOLOR=*color* BORDERCOLOR=*color* BORDERCOLORLIGHT=*color* BORDERCOLORDARK=*color* COLSPAN=*n* NOWRAP=NOWRAP ROWSPAN=*n* VALIGN=*align-type*>

Description

Creates the header of a row or column in the table. This element is similar to the TD element, except that it highlights text in the TH cell to distinguish it from the text in the TD cells.

ALIGN=***align-type***

Determines the horizontal alignment type for the cell text. Valid value for *align-type* can be one of the following:

LEFT	The text is left-aligned.
CENTER	The text is centered.
RIGHT	The text is right-aligned.

By default, the text is centered.

BACKGROUND=***url***

Determines the background picture. The picture is arranged tile-style behind any text and graphics in the table, its header or cell.

BGCOLOR=***color***

Sets the background color. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDERCOLOR=***color***

Sets the color of the border. Should be used jointly with the BORDER attribute. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color

name. See description of Color.
BORDERCOLORLIGHT=***color***
BORDERCOLORDARK=***color***
Set the drawing color for the 3D border.
COLSPAN=***n***
Specifies the number of the table columns to be spanned by the given cell.
NOWRAP=**NOWRAP**
Disables word wrapping within the cell. Lines of text appear in the cell precisely as they are specified in the HTML document.
ROWSPAN=***n***
Specifies the number of the table rows to be spanned by the given cell.
VALIGN=***align-type***
Determines the vertical alignment type for the table text. Valid value for *align-type* can be one of the following:

TOP	The text is aligned to the top of each cell.
MIDDLE	The text is aligned to the middle of each cell.
BOTTOM	The text is aligned to the bottom of each cell.
BASELINE	The text in adjacent cells of the same row is aligned to the common base line.

By default, the text is aligned to the middle of each cell.
This element is effective only within one row of the table. This implies that you must use the TR element before using the TH element.

<THEAD>

Description

Marks up the text as a table header. Use this element to separate lines of text in the table's header form lines of text in the table footnote or lines of text in the table body. A table header is optional but, if specified, only a single header is allowed. The THEAD element is valid only within the relevant table. You must use the TABLE element before using this element. Use of closing tag is optional.

Example

```
<TABLE>
<THEAD>
   <TR>
   ...
   </TR>
<TBODY>
   <TR>
   ...
   </TR>
</TABLE>
```

<TITLE>

Description

Determines the title of the document. The browser uses this element as a window caption. This element is valid only within the HEAD element. Use of closing tag is mandatory.

Example
<HEAD>
<TITLE>»Welcome to Internet!»</TITLE>
</HEAD>

Component
The title is determined the *Title* property of the *HTMLPage* component.

<TR ALIGN=*align-type* BACKGROUND=*url* BGCOLOR=*color* BORDERCOLOR=*color* BORDERCOLORLIGHT=*color* BORDERCOLORDARK=*color* VALIGN=*align-type*>

Description

Creates a table row.
ALIGN=***align-type***
Determines the alignment type for text in the row's cells. Valid value for *align-type* can be one of the following:

LEFT	The text is left-aligned.
CENTER	The text is centered.
RIGHT	The text is right-aligned.

By default, the text is centered.
BACKGROUND=***url***
Determines the background picture. The picture is arranged tile-style behind any text and graphics in the table, its header or cell.
BGCOLOR=***color***
Sets the background color. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.
BORDERCOLOR=***color***
Sets the color of the border. Should be used jointly with the BORDER attribute. The value of *color* can be specified in hexadecimal or RGB format, or as a redefined color name. See description of Color.

BORDERCOLORLIGHT=***color***
BORDERCOLORDARK=***color***
Set the drawing color for the 3D border.
VALIGN=***align-type***
Determines the vertical alignment type for text in the row cells. Valid value for *align-type* can be one of the following:

TOP	The text is aligned to the top of each cell.
MIDDLE	The text is aligned to the middle of each cell.
BOTTOM	The text is aligned to the bottom of each cell.
BASELINE	The text in adjacent cells of the same row is aligned to the common base line.

By default, the text is aligned to the middle of each cell.

<TT>

Description

The Teletype mode. The text is presented in fixed-width font.

Example
<TT>Here is some plain text.</TT>

Component
No direct analog. Programmers can use the HTMLTag component.

<U>

Description

Text is presented is underlined face.

Example
<U>This text is underlined.</U>

Component
Use the Font property for text components.

<UL>

Description

Outputs lines of text along with their positional markers. Denotes that the next block of text is made up of separate items each of which begins with the LI tag. The list items are marked with positional markers.

Example

```
<UL>
   <LI>This is item #1 of the list, marked with its positional marker.
   <LI>And this is item #2 of the list, marked with its positional marker.
</UL>
```

Component

HTMLList. Embedded lists are not supported in this version of the library. For each individual item, you can specify a separate reference in the URL properties.

<VAR>

Description

Specifies the text to take the place of a variable. Outputs the text in small fixed width font.

Example

```
Enter <VAR>filename</VAR>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<WBR>

Description

Insert a «soft» line break character into a NOBR block of text.

Example

```
<NOBR>This line of text will not be broken no matter how narrow the window will be.
<WBR>And this line, however, <WBR>will be broken.</NOBR>
```

Component

No direct analog. Programmers can use the HTMLTag component.

<XMP>

Description

An example text. The text is output in fixed-width font.

Example

```
<XMP>Here is some plain text.</XMP>
```

Component

No direct analog. Programmers can use the HTMLTag component.