



Caché WebLink Developer Guide

Software Version: 4.2
Document Updated: 7 June 1999

Caché, Caché WebLink, and Caché WebLink Developer are trademarks of InterSystems Corporation. This documentation is the property of InterSystems Corporation. All rights reserved.

Copyright © 1999, InterSystems Corporation.

Chapters
1: Introduction
2: Design Concepts and Conventions
3: Run-time Value Substitution
4: Using Hyperlinks
5: Using Forms
6: Scripts
• Pre-page
• In-page
• Post-page
• Actions
7: Re-Usable Components
8: Running Applications
9: Application State Modes
10: Timing Out Sessions
11: Templates
12: Caché Objects and SQL
13: Using Frames
14: HTTP Headers and Cookies
15: Dynamic Pages
16: SSL
17: Internet Client Functions
18: Editing Notes
Appendixes
A: Installation
B: Restarting Hung Processes

Introduction

Caché WebLink Developer is a unique development environment, allowing the sophistication and productivity of modern HTML page development tools to be exploited for the rapid development of high performance, secure, interactive, database-linked Web-based applications. Such applications can be run over the Internet or locally across an intranet.

Caché WebLink Developer has three elements:

- A set of extensions to HTML that are compatible with most HTML page development tools.
- A compiler that compiles the HTML files into Caché ObjectScript routines.
- A run-time engine that invokes the routines that play back the pages from Caché and automatically provides the mechanisms for state, session, and security management. This run-time engine automatically provides a range of sophisticated security features that significantly reduce the risk of unauthorised access to applications.

All the complexity normally associated with developing CGI-based applications (such as the associated arcane URLs) are all automated for you. Designing an application is as simple as designing a static Web site.

The Caché WebLink Developer run-time engine runs within the Caché environment. Caché WebLink Developer does **not** use Microsoft's ASP engine, and the use of Caché WebLink Developer is **not** limited to particular Web servers or browsers. Caché WebLink Developer therefore will run with both Netscape and Microsoft Web servers and browsers - in fact any Web server supported by the Caché WebLink networking component. Caché WebLink Developer is, however, compatible with Microsoft's ASP in terms of the syntax used. It is, however, a quite separate technology.

Note: Caché WebLink Developer works with InterSystems Caché (NT, 95, 98, UNIX), in addition to DSM and MSM (NT only). Although it should be compatible with all HTML page development tools, the current version has been designed to work with and tested using Microsoft's FrontPage97 and FrontPage Express. FrontPage98 is not recommended because it has been found to restructure and modify some of the HTML constructs that WebLink Developer relies upon, although the patch recently issued by Microsoft for FrontPage98 is understood to alleviate these problems.



Design Concepts and Conventions

- [WebLink Developer Concept](#)
 - [WebLink Developer Conventions](#)
 - [Reserved Variable Names](#)
 - [Globals Used by WebLink Developer](#)
-

WebLink Developer Concept

Caché WebLink Developer allows the job of developing a Web-based application to be split into two tasks:

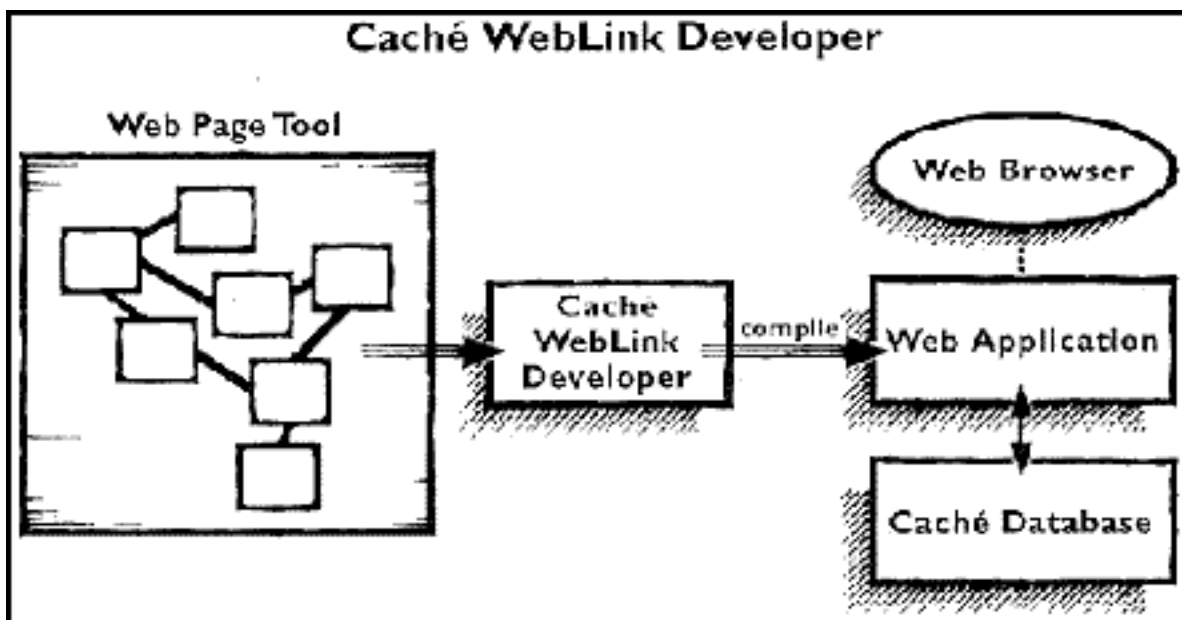
1. Designing the Web pages as if the entire application were a series of static Web pages. This job is arguably best done by designers who should not need to understand the technical complexities of Caché WebLink or Caché.
2. Linking in the back-end Caché database and business logic to those Web pages. This job is best handled by a programmer.

The goal has been to provide a common environment in which both can work comfortably — an environment that avoids working in the native Caché mode entirely; and an environment where the tasks can be transferred back and forth between page designer and programmer in a seamless way.

With Caché WebLink Developer, all tasks are carried out using a Web page development tool and a Web browser. Microsoft FrontPage97 or FrontPage Express are the Web development tools referred to in this document, but any tool should do (eg Macromedia DreamWeaver, Allaire Homesite, HoTMetal Pro, etc). In fact, because all an HTML page development tool does is to ease the creation and manipulation of HTML within ordinary text files, Caché WebLink Developer can even be used with simple text editors such as NotePad, and Developer provides its own simple browser-based page editing facility.

Although both the page designer and programmer work with HTML pages (with the programmer inserting Caché code into the HTML pages in the form of scripts), Caché WebLink Developer is used to "compile" the pages, automatically taking all the HTML pages that make up an application and pulling them into Caché as routines, globals, and objects. A run-time module "plays back" the pages in the right sequence, based on the actions selected by the user, with substitution of variables taking place automatically. The HTML page files are not actually physically used at run time, but neither the Web page developer nor programmer need ever be aware of the compiled Caché routines and globals.

The following illustration shows the development process.



Caché WebLink Developer hides all the complexities of the URLs and reserved name/value pairs that are needed to invoke WebLink from the user's Web browser. In fact, neither the page designer nor programmer needs to know anything about WebLink URLs — they are all automatically created based on the Web page designer's requirements. Development of an application is therefore in terms of a set of hyperlinked or form-linked static pages that are "brought to life" at run time.

If editing or modifications need to be made to the application, the HTML pages (not the compiled routines and globals) are edited with an HTML editor and the revised pages are recompiled using the Caché WebLink Developer configuration application. The Caché globals and routines that are created when an application's HTML pages are compiled are never actually touched by the programmer, because these Caché globals and routines are cleared and recreated at each compilation.

Application Development Cycle

After installing Caché WebLink Developer, you can develop, compile, and run applications using just a page development tool and a Web browser. You can use the Caché WebLink Developer Maintenance Suite to compile, configure, and, if required, run your applications. Indeed you can even do much of the HTML editing from your browser using Developer's Maintenance Suite if you wish. The Maintenance Suite also includes a Wizard to simplify the initial creation of an application, and can also be used to delete applications.

The normal development cycle is:

1. Create and/or edit Web pages that make up an application, using the Web page development tool of your choice.
2. If necessary, copy the files to an application directory on or addressable from the Caché server. This directory must be a sub-directory of your Global Application Directory.
3. Run the Caché WebLink Developer Maintenance Suite from a Web browser and use the hyperlink option on the main menu to "Compile an application". Select the application from the list provided.
4. To make further amendments to the application, go back to step 1.

Note that if you prefer to compile applications in Caché's programmer mode, you can do so by using the following call:

```
USER> DO ^%wldcomp(App_Name)
```

where *App_Name* is the case-insensitive name of the application to be compiled, as in this example:

```
USER> DO ^%wldcomp("Contacts")
```

In this example, all the pages in the application named "Contacts" will be compiled into a WebLink application.

You can also use the following top-level routine call to compile a single page from the command line:

```
DO ^%wldcomp(App_Name,Page_Name)
```

where *App_Name* is the case-insensitive name of the application and *Page_Name* is the case-insensitive page name of the application to be compiled; for example:

```
DO ^%wldcomp("Contacts","second")
```

The page named **Second.asp** will be recompiled. Note that you do not include the **.asp** file extension.

WebLink Developer Conventions

In order to build an application, you need to understand and adhere to the following fixed conventions.

State-Aware and State-less Modes

Caché WebLink Developer can run in both **State-Aware** and **State-less** modes of Caché WebLink. In fact, the State-less mode performs an emulation of State-Aware mode whereby local variables are stacked and recovered between pages, producing the effect of persistent local variables. This means that your approach to developing applications will be almost identical irrespective of the mode you choose to use.

If you use the State-Aware mode, you must have at least three licensed connections in order to develop and run applications. We recommend you have at least 5 connections.

Caché WebLink Developer will automatically create and look after all the URLs needed to start, maintain, and stop State Aware private sessions. You just need to think of an application as a set of hyperlinked Web pages, and build it accordingly.

Directories

An application is constructed as a set of static Web pages. The set of pages for any given application should be placed in its own directory on the Caché server. That is, each application should have its own directory, and all application directories normally have a common root directory, called the *global application directory*.

Say, for example, that you have three applications - Contacts, Accounts and Registration. You might have a global application directory of **C:\Apps\WebLinkApps**. The pages that make up each of these applications would reside, respectively, in:

- C:\Apps\WebLinkApps\Contacts
- C:\Apps\WebLinkApps\Accounts
- C:\Apps\WebLinkApps\Registration

Note that if your Web server runs on a different computer from the Caché computer, the global application directory must reside on your Caché server or must be addressable from the Caché environment as a mapped network drive/directory.

It is possible, through configuration parameters, to bypass the requirement to use a single Global Application Root Directory, and define the precise path for an application's source page directory.

Filenames

Pages within each application are assumed to be Cache WebLink Developer source files if they have a file extension of ".asp" (for example, acct_04.asp). Page names are NOT case sensitive. It is possible to reconfigure Developer to accept any file extension (though some, such as .htm, .html, .doc, etc should obviously be avoided).

An application will always have a start page (known as the First Page). By default, Caché WebLink Developer assumes that the First Page will be named **First.asp**, but this name can be changed as required in the application configuration facility. All other pages can be named as you wish, as long as they have the ".asp" file extension.

A number of file names are reserved for special purposes, acting as directives for the run-time engine. For example, Quit.asp, Back.asp, Return.asp, Do.asp and Close.asp (none of these are case sensitive). These are described elsewhere in the documentation.

Session Information

Caché WebLink Developer maintains state variables and session context for you automatically. A session is deemed to have started when the First Page (normally First.asp) is presented to the user, and finished when the [Quit.asp](#) [notional page](#) is selected (from either a hyperlink or form).

URLs in Hyperlinks

When defining a hyperlink between two Caché WebLink Developer pages, the URL should be defined using only the name of the

page to which you want to jump, including the ".asp" (or other configured) extension; for example:

First.asp
MainMenu.asp
LaunchPage.asp

A hyperlink might therefore be written as:

```
<a href=MainMenu.asp>Go to the main menu</a>
```

The important point is that the page designer does not need to worry about defining Caché WebLink URLs such as:

```
http://194.12.34.56/scripts/mgwms32.dll?MGWCHD=1p&guid=1jg32asrkjhkiuy
```

The Caché WebLink Developer run-time environment will produce these extended URLs automatically, deriving them from the easier-to-understand and maintain "pathname.asp" URLs.

Reserved Variable Names

Caché WebLink Developer's internal variables, where exposed to your scripts, start with the % symbol (for example, **%app**). Avoid using % variables in your scripts as these may clash with Caché WebLink Developer's internal variables and could cause unpredictable problems.

Developer exposes a number of other variables that you may use within your scripts. These variables provide either information on the internal status of Developer, or provide you with an interface to control certain aspects of Developer's behaviour at run-time. When you use these internal variables, treat them as reserved variable names and do not use these names for other purposes. A list of the Caché WebLink Developer's reserved variables follows.

Reserved Variable Names	
PREVPAGE	Read-only variable. Determines the previous page that led to the current one. Contains the name of the page in upper case without the .asp file extension (eg "FIRST", "MAINMENU")
BACK	A boolean that, when set to 1 in a pre-page processing script, forces the user back to the previous page (for example, when a user error is detected). BACK is automatically reset between pages to false (0).
JUMP	Used in pre-page processing scripts to force a jump to another page. Set JUMP to the name of the page (without the file extension) to jump to, and QUIT from the script. JUMP is reset automatically to a null value between pages.
BACKTRACK	A boolean read-only variable. If true (ie a value of 1), this indicates that the current page was returned to as a result of using the Back.asp pseudopage mechanism. Under many circumstances a page's pre-page script may need to detect this situation in order to correctly pre-fetch or initialise form field contents.
Error	Used to display all errors. Caché WebLink Developer detects a number of errors automatically and uses this variable to denote the cause of the error. You can generate error messages to the browser by setting the message into this variable.
Warning	Used to display warning messages. You can generate warning messages to the browser by setting the message into this variable. Whereas Error causes focus to be fixed on the form field that caused the error, Warning allows the focus to move to the next form field.
Focus	Used to define the field on which to move focus during field validation by the Event Broker.
PRESSED	Read-only variable. Contains the name of the Submit button that was pressed on the previous form. Used in applications whenever a form has multiple submit buttons.

MGWLPN	Read-only variable. The standard WebLink Login Profile Name. You normally do not need to refer to this.
MGWCHD	Read-only variable. The standard WebLink Connection Handle. You normally do not need to refer to this.
HomePage	Read-only variable. The URL of your (static) Home Page (typically, ../default.htm). The Home Page is set and modified with the Configure an Application option in the Developer Maintenance Suite.
CRLF	Read-only variable. Automatically set up to contain the standard end-of-line delimiter characters ASCII 13 (carriage return) and ASCII 10 (line feed).

Note on Reserved Words

You should avoid using a number of names as substitutable variable names within HTML pages, as the compiler may convert them to upper case as part of its parsing and processing. We have tried as much as possible to eliminate this potential side effect, but you may find circumstances where it occurs.

To be safe, avoid the key words the compiler is searching for. You can find an up-to-date list of these key words if you look in the **^%wldutg** entry-point section of the **wld.rou** routine file (in CacheSys\WebLink\scripts). Use a text editor (such as WordPad) to view the file.

Keep in mind that you can safely use a variable name that is partly made up of a reserved word. For example, the variable **MyName** is fine, but avoid using **Name** as a variable.

Globals Used by WebLink Developer

All the information that Caché WebLink Developer needs and uses is held in the following three globals.

Global Names	
^%WLDGASA	The "global ASA file" (in ASP parlance). This global contains the core Developer configuration information and is in the %SYS namespace.
^WLDROU	The routine/page name mapping global, that resides in the namespace where an application was compiled.
^WLDPARAM	Contains application-specific configuration parameters. This global resides in the namespace where the application was compiled.

Note that when you distribute an application to another machine, you only need to copy the compiled routines. Provided the target machine has a copy of WebLink Developer installed, it will create automatically any global references it requires.

Run-time Value Substitution

- [Simple Value Substitution](#)
 - [Variable URLs](#)
 - [Using \\$ Variables and Functions](#)
 - [Other Run-time Substitution Methods](#)
 - [Using MS ASP Syntax](#)
-

A key difference between a set of static pages and a dynamic, database-linked Web application is that certain fields and pieces of information on a page will vary in value, depending on the circumstances and data at run-time.

Caché WebLink Developer allows the application to be designed as a set of pages, using variable names that are automatically substituted at run-time with the actual values. The Web page designer can therefore determine and define the positioning, format, appearance, etc., of any variable item, without worrying about how the run-time value will be obtained. One of the main tasks of the programmer is to subsequently insert the logic into each page that will create the run-time values of each variable.

Simple Value Substitution

To substitute text at run-time onto a page, place a variable name in the page, with the name enclosed in vertical bars (the "|" character). A variable can be placed anywhere within the HTML that makes up a page (including any JavaScript), using the general form:

|VariableName|

VariableName can be any valid Caché ObjectScript variable name.

Suppose, for example, that you want to display a person's date of birth, using a run-time substitution of a variable named "DoB". At the required position in the page, simply use:

|DoB|

You may add any formatting you require. For example, you can display the value displayed in italics and/or as a heading. Simply apply such formatting to the variable name **and** the vertical bars that enclose it. You may use variables anywhere in a page, including URLs.

If the variable has a null value at run-time, no value is displayed. If, under such circumstances, the variable is the only text on a line on the Web page, the line will not be written to the browser; that is, you will not get a blank line, but instead the gap will be closed automatically.

If the variable is undefined, the variable name and the vertical bars will appear on the browser at run-time. This is because vertical bars are sometimes used as text delimiters on Web pages. If the text between vertical bars cannot be substituted, WebLink Developer assumes that the actual text must have been meant to be displayed.

Variables can be used in any HTML or JavaScript; for example:

```
<BODY OnLoad="HighlightRow( |CurrentRow| )" >
```

In this example, the run-time value of the Cache' variable "CurrentRow" will be substituted in as the parameter to the Event Handler function call.

This is an extremely powerful feature of Developer.

Variable URLs

A variable within "|" (or <% %>) characters may be defined as a hyperlink and associated with a URL. At run-time, the actual value for the variable is substituted and appears as the hyperlinked text. The actual HTML needed to create a hyperlink that is substituted at run time is of the form:

```
<a href="NextPage.asp">|VariableName|</a>
```

For example, if |PatientName| is defined as a hyperlink, the HTML code:

```
<a href="NextPage.asp">|PatientName|</a>
```

would create a hyperlink such as:

```
David Smith
```

You can also add values to URLs that are resolved at run-time. For example, you can have a hyperlink on a page that sends a person's date of birth back to the Caché server. However, the date of birth will not be known until run-time, and not necessarily until the page is actually written to the user's browser. This problem is easily solved by Caché WebLink Developer. Simply add the name/value pair to the URL, with the value defined as the variable; for example:

```
BirthDateProc.asp&DoB=|Dob|
```

This code adds the run-time value of the Date of Birth (as defined in the DoB variable) to the name/value pair list of the URL associated with the hyperlink that calls up the BirthDateProc page.

If you wanted a patient's name (as held in the variable PatientName) to appear as the hyperlinked text, the full HTML would be as follows:

```
<a href="BirthDateProc.asp&DoB=|Dob|">|PatientName|</a>
```

Caché would receive the variable DoB with the value defined in the variable DoB at the time when the hyperlink tag was written to the page.

Using \$ Variables and Functions

You can use \$ variables and \$ functions as variable fields within form element tags and as variables within the HTML, as in this example:

```
Current $H Date is |$H|
<Input type=text name=|$E(MyName,1,5)| value="|$P(String,Delim,PceNo)|">
```

Other Run-time Substitution Methods

You may also substitute global references, local array references, expressions, and functions within Web pages, as demonstrated by the following examples.

Run-time Substitution Examples	
^XYZ(123,4)	will substitute the value of the global reference ^XYZ(123,4)
<%=abc("Smith")%>	will substitute the value of the local array abc("Smith")
\$login^ABC()	will substitute the value of the expression login() in the routine ^ABC

In all these examples, the [...] and <%...%> syntaxes behave identically (provided the Language for ASP processing has been switched to Caché).

Using MS ASP Syntax

Caché WebLink Developer allows you to use Microsoft's Active Server Page conventions and syntax. In-page variables can therefore also be defined using the convention:

```
<%=VariableName%>
```

At run-time, this syntax is functionally identical to the |VariableName| syntax described above. Note, however, that FrontPage will display it differently within the FrontPage Editor.

If you are using MS ASP-style syntax, you must tell the Caché WebLink Developer compiler that your variable substitution is to be carried out by Caché WebLink Developer's run-time engine, rather than Microsoft's ASP engine (which is the default).

Therefore, in any page containing MS ASP-style notation, you must precede any ASP-style variable substitution with a single use of the directive:

```
<%@Language=Cache@%>
```

Caché WebLink Developer will then process any ASP-syntax statements for use by its own run-time engine.

You can revert to Microsoft's ASP processing at any point in the page by including the statement:

```
<%@Language=VBScript%>
```

Any subsequent ASP-syntax statements will then be ignored by the Caché WebLink Developer compiler, for processing by Microsoft's ASP engine instead.

You can therefore repeatedly switch between VBScript and Caché ASP notation within a page.



Using Hyperlinks

- [Invoking Functions](#)
 - [Using FrontPage to Define Hyperlinks](#)
 - [Other URL Substitutions](#)
-

An [earlier chapter](#) describes how to use hyperlinks to link pages within a Caché WebLink Developer application, by using the .asp pagename within the application's directory. You use the following syntax:

```
<a href="NextPage.asp">Go to the Next Page</a>
```

That chapter also shows how additional parameters can be passed to the next page by adding a name/value pair list to the page name, as follows:

```
<a href="NextPage.asp&param1=value1&param2=value2">  
Go to the Next Page</a>
```

or, if the values are unknown until run-time:

```
<a href="NextPage.asp&param1=|value1|&param2=|value2|">  
Go to the Next Page</a>
```

Note that you **cannot** use a variable page name. URL conversion is performed at **compile-time**, not run-time. Therefore, hyperlinks of this form will not work:

```
<a href="|PageName|.asp&param1=|value1|&param2=|value2|">  
Go to the Next Page</a>
```

When using run-time value substitution, keep in mind that any variables you create are persistent (even in Caché WebLink Developer's State-less Mode) and can be accessed in scripts on subsequent pages (unless you explicitly KILL or NEW them). The values do not need to be passed as URL parameters, unless a new value has to be passed to the back end.

Invoking Functions

You can invoke a Caché function when a hyperlink is clicked by including the parameter:

```
&action=FunctionName()
```

where *FunctionName* is the label of a function residing in your [Actions](#) script.

For example, you might have the following hyperlink in the page "EditDetails.asp":

```
<a href="Mainmenu.asp&action=file()">File the details</a>
```

Caché will expect to find a function with the starting label **file()** within the EditDetails.asp page, as in this example:

```
<script language=Cache>  
//type=actions  
file() ; validate and file details  
I Name="" S Error="You must enter a name" Q ; generate error alert window  
...  
; file the details  
...
```

```
Q
</script>
```

Using FrontPage to Define Hyperlinks

This section gives some tips on using Microsoft FrontPage (97 or Express) to create hyperlinks.

To define a WebLink-controlled hyperlink between pages First.asp and MainMenu.asp:

1. Call up First.asp into the FrontPage editor and type in the text for the hyperlink (for example, "Go to Main Menu").
2. Highlight the hyperlink text with the mouse and click the **Link** icon.
3. Enter "MainMenu.asp" in the **URL** field.
4. Click **OK** to accept.

To subsequently edit a hyperlink:

1. Place the mouse over the hyperlink and right-click it.
2. Select **Hyperlink Properties** from the menu.

Other URL Substitutions

You may refer to an .asp file in any HTML or JavaScript that would normally expect a URL. WebLink Developer will automatically expand the .asp file reference into the appropriate CGI WebLink URL.

In this JavaScript example, note that an additional, optional name/value pair is being sent (i.e., Var1 = value1) with the URL:

```
top.document.location = Second.asp&Var1=value1
```

In this second JavaScript example, a new browser window will be opened and loaded with the **MyPage** page:

```
MyWindow = window.open(MyPage.asp,WldWin1,features) ;
```



Using Forms

- [Ensuring WebLink Processes the Form](#)
 - [Form Contents](#)
 - [Run-time Substitution of Page Names](#)
 - [Submit Attributes](#)
-

HTML forms provide the means by which the user can interact with a Web application by typing information into fields and/or selecting from radio buttons, check-boxes, and drop-down lists.

Forms can be thought of as having three components that Caché WebLink Developer must be able to handle correctly:

- A means of ensuring that WebLink is used to process the form (i.e., the form's contents are passed, by WebLink, to the back-end Caché system).
- The contents that make up the form (i.e., the input elements between the <FORM> and </FORM> tags).
- The "Submit" button(s) within a form (typically at the end of the form). When clicked, these buttons cause the form's contents to be sent from the browser to the Web Server (which forwards it to Caché via WebLink).

The rest of this chapter examines each of these areas in detail.

Ensuring WebLink Processes the Form

The Web page must ensure that WebLink is used to process the form. That is, the form's contents must be passed by WebLink to the back-end Caché system. The <FORM> tag has two attributes that can control the processing of the form:

- The **METHOD** attribute. It is recommended that you use the POST method, because the alternative SEND method can lead to maximum string length limitations within the data stream to the web server.
- The **ACTION** attribute. Caché WebLink Developer will automatically set this attribute to ensure WebLink is invoked. Therefore, you **must** leave out this attribute. (For FrontPage, leave this attribute blank).

The resulting HTML that you should use in your page is as follows:

```
<form method=post>
```

Weblink Developer will automatically expand this out as required at compilation time.

Form Contents

This section describes the format and requirements for the contents of the form (that is, the "input elements" that make up the form between the <FORM> and </FORM> tags.)

Input Elements

You must provide a NAME attribute for every form element. Caché WebLink Developer maps this name automatically to the equivalently-named Caché local variable. Note this mapping is case sensitive. If you have DSM systems, remember that these systems only support local variables up to 8 characters long, so you must limit the length of your input field names accordingly.

To illustrate how the input elements are processed, assume that you specified:

```
Enter your name: <INPUT TYPE="Text" NAME="Username">
```

Then at run-time, on submission of the form, the Caché variable **Username** will contain the value of this field, as entered by the user.

Note that any extraneous punctuation characters etc will be stripped out when Developer constructs the appropriate variable name; for example:

```
Enter your name: <INPUT TYPE="Text" NAME="User_Name">
```

would generate the Caché variable **UserName**

A useful feature is that field names can be variables or include variables, the latter extremely useful if a loop is being used to create a repeated number of the same type of field; for example:

```
<INPUT TYPE="Text" Name=MyText | LoopNo | >
```

If LoopNo is a Caché variable indicating the row number in a dynamically generated table, then at run time, repeated Text fields would be created as follows:

```
<INPUT TYPE="Text" Name=MyText1>
<INPUT TYPE="Text" Name=MyText2>
<INPUT TYPE="Text" Name=MyText3>
etc...
```

Pre-populating Input Fields

To show an existing value in a text box in a form, set the VALUE attribute to contain an asterisk (*) and ensure that the page has a pre-page script or (less usually an) in-page script that creates the local variable of the same name as the text box.

Assume, for example, that the INPUT field is defined in the HTML page as:

```
<INPUT NAME="Username" TYPE="text" VALUE="*">
```

If the variable **Username** is then already set to the value "William", the word "William" will appear in the Username text box when the page is written to the browser.

Expressions in Value Parameter of Input Fields

You can use variables or an expression within the **VALUE** parameter of an <INPUT> text field by using the "|" syntax, as in this example:

```
<input type="text" size="20" name="T1" value="| $P(data,c,sep) | ">
```

Options in Selection Lists

When you use a selection list (such as pop-up menu), you can define the list of options by hard-coding the values for each list option. However, a more useful and flexible method is to let Caché WebLink Developer automatically map the option values at run-time.

To use this automatic mapping method, you create a local array **LIST** in one of your [in-page scripts](#). **LIST** has two subscripts:

- **Element name** - the name as specified in the Web page (such as "City" in the example below).
- **Menu option number** - an integer from 1 to n (where n is the maximum number of options in the selection list). The integer specifies the order of the option in the list.

The **value** of each array element contains the text to be displayed in the list, as illustrated by the city names in this example:

```
LIST("City",1)="Aberdeen"  
LIST("City",2)="Bristol"  
LIST("City",3)="Coventry"
```

This LIST contents will populate the selection list element whose NAME attribute is "City" with the three city names, in the order specified by the second (numerical) subscript.

Your HTML page needs only to provide the barest of details for the SELECT element, creating a placeholder for Caché WebLink Developer to automatically substitute your values as the corresponding <OPTION> tags at run-time. Therefore, in the above example, you need only define:

```
<SELECT NAME="City"></SELECT>
```

In other words, you do not need to specify any <OPTION> tags. In fact, WebLink Developer ignores any pre-defined <OPTION> tags if the **LIST** array is defined for the drop-down list. If there is no **LIST** array defined at run-time, then any <OPTION> tag contents are displayed.

Note that the **LIST** array is persistent, and is not modified or deleted by WebLink Developer at run-time. It is up to you to control its scope if necessary.

Returning Values From SELECT Items

The value(s) selected by the user from a drop-down list are returned by Developer to Caché in a local array called **SELECTED**. For example, if you chose "Chicago" from a list of cities (e.g., from a <SELECT> tag named "city"), you would get:

```
SELECTED("city","Chicago")=" "
```

However, you may instead want a matching **coded value** for the selected item, such as the value "23" for the "Chicago" selection. To do this, you must set up a **VALUE** array that matches the **LIST** array.

The syntax of the **VALUE** array is:

```
VALUE(field_name,i)=return_coded_value
```

where *field_name* is the name of the drop-down SELECT item (e.g., "city"), *i* is an integer from 1 to *n* (where *n* is the number of options in the selection list), and *return_coded_value* is the coded value for the *i*th SELECT option. An example would be:

```
VALUE("city",5)=23
```

The syntax of the two matching parallel arrays is:

```
LIST(field_name,i)=display_value  
VALUE(field_name,i)=return_coded_value
```

For example:

```
LIST("city",5)="Chicago"  
VALUE("city",5)=23
```

WebLink Developer then uses the **VALUE** array to return the correct code:

```
SELECTED("city",23)=" "
```

If you do not use the **VALUE** array, the **LIST** array is used (as described above) for the return value in SELECTED:

```
SELECTED("city","Chicago")=" "
```

Note that for single-choice drop-down lists, a variable of the same name is also returned; for example:

```
city = "Chicago"
```

If the **VALUE** array is available, the variable uses the appropriate coded value:

```
city = 23
```

An alternative mechanism is to use a second "piece" in the LIST array value, with a delimiter of "~", as in this example:

```
LIST("city",5)="Chicago~23"
```

This mechanism is functionally identical to using:

```
LIST("city",5)="Chicago"  
VALUE("city",5)=23
```

Multiple-Selection Lists

By using the MULTIPLE attribute with the <SELECT> tag, you can build a selection list that allows the user to select more than one option from the list; for example:

```
<SELECT NAME="City" MULTIPLE>\
```

When a form containing such a field is submitted, the multiple values are returned to Caché in the local array:

```
SELECTED(FieldName,value)=" "
```

where *FieldName* is the NAME attribute of the field element and *value* is the text value of a selected item.

For example, assume that you are using a multiple-selection list to select a city:

```
<SELECT NAME="City" MULTIPLE>  
<OPTION>New York  
<OPTION>Boston  
<OPTION>Chicago  
...  
</SELECT>
```

If New York and Boston were selected, the following array nodes would be set within Caché:

```
SELECTED("City","Boston")=" "  
SELECTED("City","New York")=" "
```

Note that the SELECTED array for a FieldName is flushed before returning the selected values. Any nodes denoting previously-selected values are deleted.

To pre-select check boxes, radio buttons or drop-down lists, simply pre-define the appropriate nodes of the SELECTED array. So in the above example, if these two nodes were set:


```
SELECTED("City","Chicago")=""  
SELECTED("City","New York")=""
```

then the selection list would be drawn with Chicago and New York pre-selected.

The use of the SELECTED array is therefore symmetrical on the input and output side, and applies to all single or multiple choice items (radio buttons, check-boxes and drop-down lists).

TextArea Boxes

You can pre-define the text to appear in a TEXTAREA box using the Caché array:

```
TEXTAREA(ElementName,LineNumber)=Text
```

Text may contain carriage return and linefeed characters. A carriage return and line feed are inserted automatically between each *LineNumber*.

For example, your HTML need only contain:

```
<TEXTAREA NAME="Notes" ROWS="5" COLS="20">  
*  
</TEXTAREA>
```

The asterisk (*) denotes that text held in the TEXTAREA array should be used as the default text to be used to pre-populate the box. Note that if you omit the asterisk, no text will appear in the box.

So, if, at run-time, your pre-page generates the local array:

```
TEXTAREA("Notes",1)="This is the first line of my notes"  
TEXTAREA("Notes",2)=""  
TEXTAREA("Notes",3)="A third line after a blank line"
```

Then these three lines of text (with the second line blank) would be automatically displayed in the TEXTAREA box as the default value.

Handling Long Paragraphs

Very long paragraphs are handled differently by WebLink Developer to ensure that they do not cause maximum string-lengths to be exceeded within Caché. Long paragraphs are split into lines of about 200-250 characters and placed in separate nodes of the TEXTAREA array. The second and subsequent nodes of such split lines have the tag <CONTINUE> inserted at the start of the text.

When retrieving text for subsequent re-display by WebLink Developer, you must not change the <CONTINUE> tag because it instructs WebLink Developer **not** to send a Carriage Return/Line Feed to the browser, but instead continue writing the text to the browser on the same TextArea line. If you are using Developer, this is handled transparently for you. However, if you are displaying the saved text using another environment (e.g., printing it), you must use the equivalent rules to process the <CONTINUE> directives. For example, the TEXTAREA nodes might look like:

```
TEXTAREA("Report",1)="This is the title of the Report"  
TEXTAREA("Report",2)=""  
TEXTAREA("Report",3)="Start of the first paragraph...etc. for about 200 characters"  
TEXTAREA("Report",4)="<CONTINUE>Paragraph 1 continues for about 200 chars"  
TEXTAREA("Report",5)="<CONTINUE>This is still the first paragraph"  
TEXTAREA("Report",6)=""  
TEXTAREA("Report",7)="This is the start of paragraph 2"
```

Processing the Edited Text

The TEXTAREA array is also used to hold the edited version of the text **after** the form containing it is submitted. Any previous contents of the TEXTAREA array for the input field concerned will be flushed and replaced with values corresponding to the user's input.

After form submission, the top level node of the TEXTAREA array will contain the number of lines of text saved in the array. For example, a TEXTAREA element called "Info" may contain, after submission of the form, the elements:

```
TEXTAREA("Info")=2
TEXTAREA("Info",1)="This is line 1 of the information"
TEXTAREA("Info",2)="This is line 2 of the information"
```

Note that Caché WebLink Developer removes any Carriage Return and Line Feed characters from the text returned by WebLink. Therefore, you do not need to do any further processing (but see the [previous section](#) on how long paragraphs are handled).

Note also that Caché WebLink Developer does not automatically delete or flush the TEXTAREA array between pages. Therefore, you must control its scope within your application if necessary.

Run-time Substitution of Page Names

WebLink Developer can substitute **.asp** page names at run time via the use of variables. For example, the **NextPage** extended attribute of a Submit button can be a variable:

```
<INPUT TYPE=SUBMIT NAME=Login NextPage=|MyPage|>
```

In this example, if the variable MyPage contained the value "MainMenu", then submitting the form will take the user to the page MainMenu.asp. Note the value of MyPage is case insensitive. NextPage can also use a function or expression to return the value.

```
<INPUT TYPE=SUBMIT NAME=Login NextPage=$$jump()>
```

In this example, a subroutine labelled **jump()** would be required in the page's Actions script, and should evaluate to a valid page name (case insensitive); for example:

```
jump() ; jump to a page depending on run-time conditions
I Username="Rob" Q "MainMenu"
Q "OtherMenu"
```

Another way of providing page navigation that is controlled at run-time is to use a *jump* page, i.e., a fixed page with a pre-page script that determines where to go by using the [JUMP](#) variable on the basis of some variables that you set.

However, you can use a further mechanism within forms. If you have the following HTML and in-page script, run-time substitution of the page name will take effect:

```
<%
; SubName = name of the submit button
; NextPage = name of page to go when submit button pressed
; Action = label of function to run when button pressed
D RTSUB^%wlduta(SubName,NextPage,Action)
%>
<INPUT TYPE="submit" Name="|SubName|" Value="Login">
```

Submit Attributes

The form's Submit button causes the form's contents to be sent from the browser to the Web Server, which in turn forwards it to Caché (via WebLink).

Caché WebLink Developer has three optional attributes for submit buttons that you can use to control the behavior of submit buttons and the forms to which they relate.

Submit Button Attributes	
DisplayIf	Displays the Submit button if the associated Caché expression or function evaluates to true (or 1). If it evaluates to false, the button is not included in the HTML page at run-time. For example, if you used DisplayIf=\$test() in the submit button, a subroutine labeled test() must be in the page's Actions script.
Action	Denotes the Caché function or expression to be executed when the button is clicked. The function will be expected to be found in the page containing the Submit button. Note that you can have an Actions script within any page, even if the page does not have a pre-page script.
NextPage	Defines the name of the page to which Caché WebLink Developer will jump (provided Error is not set by the Action) when the button is clicked. The .asp file extension does not need to be specified for the next page. A variable, function or expression that evaluates to a page name can be used if required.

Your Action function or expression can perform any actions you wish within Caché. Normally it will be used to validate the data contained in the submitted form. If the special Caché WebLink Developer variable **Error** is set within your Action to a non-null value, the **NextPage** will be ignored. Instead, the current page will be re-displayed and an Alert window will be generated with the text held in the **Error** variable. For example, if your pre-page script contained:

```
validate() ; validate the submitted data
if UserName="" set Error="You must enter a valid username" quit
...
```

then a JavaScript alert window containing the words "You must enter a valid username" is displayed if the form was submitted with a null value in the UserName text box.

You should also note the following:

- If **Error** is left undefined, Caché WebLink Developer will take the user to the page defined by the **NextPage** attribute. Caché WebLink Developer automatically flushes the **Error** variable between pages.
- The **DisplayIf** function is optional. If omitted, the button is always displayed.
- Your **DisplayIf** function can return either "true" or 1 to cause the button to be displayed. Returning a value of "false" or 0 causes the button to be omitted from the HTML generated at run-time.
- These Caché WebLink Developer attributes are processed by the Caché WebLink Developer compiler and run-time engine and do **not** appear in the HTML sent to the user at run-time.

As an example, assume your page contains the following HTML:

```
<INPUT NAME=Submit TYPE=submit VALUE="Test Data" DisplayIf=disp() Action=validate()
NextPage=MainMenu>
```

Then the "Test Data" button will:

- be displayed on the form if the function **disp()** evaluates to true.
- invoke the function **validate()** when pressed.
- take the user to the MainMenu page, provided **validate()** does not set the Caché local variable "Error" to a non-null value.

The functions **disp()** and **validate()** will be expected to be found by the Caché WebLink Developer compiler in this page's Actions

script.

Variables in the Submit Button

You can use a variable for the **NAME** parameter of a Submit button; for example:

```
<INPUT TYPE=SUBMIT NAME=|MyButton| VALUE=Login>
```

The value of the **MyButton** variable will be substituted.

Note that you may have fixed text before and/or after the variable part of the name:

```
<INPUT TYPE=SUBMIT NAME=abc|MyName|ghi VALUE=Login>
```

If the value of **MyName** was "def" at run time, the Submit button's name would be "abcdefghi".

Adding More Name/Value Pairs to the NextPage Parameter

Normally the desired action within a form is that the values to be sent to the Caché back-end are all defined as form elements or hidden fields within the form. It is actually possible to add further name/value pairs by adding them to the NextPage parameter after the page name, as this example shows:

```
<INPUT TYPE=SUBMIT NAME=Submit VALUE=Login NextPage=MyPage2.asp&Var1=Value1>
```

or

```
<INPUT TYPE=SUBMIT NAME=Submit VALUE=Login NextPage=MyPage2.asp&Var1=|MyValue|>
```

In the latter example, the run-time value of the Caché variable **MyValue** is substituted into the parameter list.

Using FrontPage to Configure Forms

To define a form in First.asp that, on submission, takes the user to MainMenu.asp:

1. Create the form by adding form elements into the page.
2. Place the mouse within the dotted area denoting the scope of the form and right-click it.
3. Select **Form Properties** from the menu.
4. Select **Settings**.
5. Make sure that the form METHOD is set to POST. Note that Caché WebLink Developer will automatically insert the correct ACTION for WebLink (i.e., /scripts/mgwms32.dll) at compile time, so you should leave ACTION blank.
6. Create a Submit button in the form (not a "Normal" button), and double click on it. Give the button a suitable name and Value/Label (e.g., "Login" - Will make the button read Login when displayed on the user's browser). Click on the **Extended..** option for the submit button.
7. Use the **Add** option to add two Caché WebLink Developer properties:

In the Name field, type **NextPage**.
In the Value field, type **MainMenu**.

In the Name field, type **action**.
In the Value field, type **login()**.

8. Create an (or edit the page's existing) Actions script, and add a function called **login**:

```
login() ; test for valid login  
I Username="" S Error="You must enter a Username" Q
```

...
Q

If **Error** is set by the **login** function, Caché WebLink Developer automatically repeats the page and displays the error message in a JavaScript Alert box. If **Error** is not defined, the action assigned to the button is assumed to have been successfully run and Caché WebLink Developer will jump the user to the page defined by the **NextPage** property.

Catering for a Variable NextPage

Clearly, the example above only allows one page to lead to one other; i.e., **First** goes to **MainMenu**. However, there will be situations where you will want to jump to different pages, depending on values of variables at run-time. We can also jump to a specific page by setting, within a page's pre-page script:

```
JUMP=page
```

where *page* is the name of the required page (without the .asp extension). For example:

```
I Button=">" S JUMP="NEXT"
```

This example will jump to a page called **Next.asp** (notice that JUMP must use the page name in upper case) if the value of the variable **Button** is ">".

Another, more elegant, way to jump to variable pages is to use an *expression* rather than an absolute page name in the **NextPage** property; for example:

```
NextPage=$$jump()
```

Caché WebLink Developer will expect to find an expression called **jump** in the Actions script which should Quit with the next page name; for example:

```
jump() ; jump to a page depending on run-time conditions  
I Username="Rob" Q "MainMenu"  
Q "OtherMenu"
```

Using Multiple Submit Buttons

If a form has multiple Submit buttons, WebLink Developer can identify which Submit button was pressed. The pre-page script in the next page has access to the [PRESSED](#) variable, which holds the name of the Submit button that was pressed.

Setting Focus in a Form

When an HTML page containing a form is loaded into a browser, focus is not set by default onto any of the form elements. If an initial focus is desired, this has to be done using a JavaScript function. Developer will do this for you, however — simply add, to the field on which you want initial focus to be set, the Developer-specific parameter **Focus=true** (case insensitive), as in this example:

```
<INPUT TYPE=TEXT NAME=Username FOCUS=true>
```

Only one field within the page should contain this parameter. Should more than one field include it, then focus will be set on the last of such fields in the page.

Submitting Forms Using Clickable Images

Forms are normally submitted by using one or more "submit" buttons, defined with the `<INPUT TYPE=SUBMIT>` tag. However, you have no control over the appearance of submit buttons, apart from defining their text.

It is possible to design your own buttons as images (e.g., GIF files) and embed these within the form. You can then use JavaScript to enable these custom buttons to submit the form.

You must use a variant of the anchor (<A>) tag that just defines the **OnClick** event handler, but note the reference using the "javascript:" call. Embed the IMAGE tag within the anchor, as in this example:

```
<a href="javascript:onclick=asub()">

</a>
```

Then include the **asub()** JavaScript function in the page:

```
<script language="JavaScript"> function asub() {
    document.forms[0].submit() ;
}
</script>
```

This will force submission of the first form on the page when the image is clicked.

Note that this approach does not demand that the image is actually inside the <FORM> and </FORM> tags, because the JavaScript function defines which form to submit.

Submitting Forms Using the <Input Type=Image> Tag

A feature of HTML is the ability to use an image as a substitute for a Submit button. Developer supports its use, and it can take the same extended parameters available for Submit buttons; for example:

```
<INPUT TYPE=IMAGE NAME="Image1" img src="images/next.jpg" Border=0 Action=login()
NextPage="Second" Width=66 Height=30>
```

Note that the NAME= parameter MUST be specified to ensure the correct behaviour at run-time. If the NAME= parameter is omitted, Developer will almost certainly make the wrong assumption about which button and/or image was pressed. The NAME= parameter will accept a variable name, and the Action= and NextPage= parameters are used identically to Submit buttons.

Submitting Forms Automatically With JavaScript

Normally, submission of a form occurs explicitly, because a Submit button has been pressed. However, JavaScript allows you to submit a form under automatic, program control. You do this by including a JavaScript function in your source HTML file. The function invokes the submit method for the named form.

For example, if you have a form of the type:

```
<form method=post name=MyForm>
... </form>
```

Then you can include a JavaScript function such as:

```
<script language="JavaScript">
function sendit() {
    document.MyForm.submit() ;
}
</script>
```

You then need to include an event handler (such as **OnChange**) that will invoke the **sendit()** function.

With a Developer application, however, you must tell the run-time engine the name of the next page when a form is submitted.

Normally, you do this with the Developer-specific **NextPage=** parameter within the Submit button tag. When using an automatic form submission, you must include a hidden field that will define the variable **NextPage** (the name is not case sensitive), as in this example:

```
<form method=post name=MyForm>
<input type=Hidden name=NextPage value="MyNextPage">
...
</form>
```

This is equivalent to:

```
<form method=post name=MyForm>
...
<input type=Submit name=Submit value="Send It" NextPage="MyNextPage">
</form>
```

A form can contain a submit button *and* be capable of being submitted automatically, as shown in this example:

```
<form method=post name=MyForm>
<input type="Hidden" name="NextPage" value="MyNextPage"><BR>
...
<input type=Submit name=Submit value="Send It" NextPage="MyOtherPage">
</form>
```

In this case, Developer will jump to **MyNextPage** if the form is submitted automatically, or to **MyOtherPage** if the Submit button is clicked.

If a form is submitted automatically, there is no equivalent mechanism for assigning an action to a Submit button. However, any field validation, etc. can be included in the Pre-Page script of the NextPage (e.g., in **MyNextPage.asp** in our example).

The one difference is that if an error condition is identified within the script, you must use the BACK variable to force the redisplay of the form:

```
<script language=Cache>
//type=pre-page
If MyCompany="" Set Error="You must enter a company name",BACK=1 Quit
...
</script>
```



In-page Scripts

- [Rules for Pre-page Scripts](#)
 - [Using Pre-page Scripts](#)
 - [Tables with Repeating Rows](#)
 - [Conditional Page Sections](#)
 - [Using ASP Syntax with FrontPage](#)
-

There are some situations where pre-page processing scripts are insufficient, and it is desirable to insert scripts within a page, at a specific location in a page. For example, in the middle of a page you may want to recall repeating records from the database, such as retrieve all the notes of visits to a client. In-page scripts can also be used to include or exclude portions of HTML depending on conditions at run-time.

In-page scripts can be defined to allow you to perform this type of action inside a page. The procedure involves two general steps:

1. The page designer defines how a single record should be laid out, either directly on the page or within a table.
2. The programmer creates a loop around this area of the page, fetching the data items for substitution at run-time.

The run-time module of Caché WebLink Developer then does the rest, amalgamating the programmed loop with the HTML in the middle, substituting variable values on every loop. Therefore, an in-page processing script usually, but not necessarily, creates a loop. The following sections describe two uses of in-page scripts.

Creating Simple Repeating Lists

To create simple repeating sequential lines, the Web page designer defines the Web page layout for one loop, using variables (i.e., delimited using the "|" character) wherever any run-time substitution needs to occur. The resulting HTML might look like:

```
<hr>
Date: |VisitDate|<br>
Our contact: |Contact|<br>
Notes of visit: |Notes|<br>
<br>
```

The programmer then inserts a loop around this HTML. Simple looping constructs based on GoTos are advisable - the use of "dot" syntax should be avoided. For example, a simple \$ORDER loop through the contact details global could be used.

Two alternative syntaxes are available: SCRIPT tags and ASP notation.

Syntax 1: Using <SCRIPT> Tags:

```
<SCRIPT Language=Cache>
//type=in-page
Set date=""
For Set date=$Order(^Contact(date)) Quit:date="" D List
G End ; jump past the HTML that lists the contacts
;
List ; get the details
Set d=^Contact(date)
Set VisitDate=date
Set Contact=$P(d,"#",1)
Set Notes=^Contact(date,"notes")
; now display the list
</SCRIPT>
```

```
<hr>
Date: |VisitDate|<br>
Our contact: |Contact|<br>
Notes of visit: |Notes|<br>
```



```
<SCRIPT Language=Cache>
//type=in-page
Quit
End ;
</SCRIPT>
```

continue with the rest of the HTML page.

Syntax 2: Using ASP Notation:

```
<%@Language=Cache%>
<%
Set date=""
For Set date=$Order(^Contact(date)) Quit:date="" D List
G End ; jump past the HTML that lists the contacts
;
List ; get the details
Set d=^Contact(date)
Set VisitDate=date
Set Contact=$P(d,"#",1)
Set Notes=^Contact(date,"notes")
; now display the list
%>

<hr>
Date: |VisitDate|<br>
Our contact: |Contact|<br>
Notes of visit: |Notes|<br>
<br>

<%
Quit
End ;
%>
```

continue with the rest of the HTML page.

Note1: In-page scripts must **not** Quit. If they do, the page will be prematurely finished (of course, subroutine blocks *within* an in-page script can Quit, as in the preceding example).

Note2: Scripts defined using the ASP syntax are implicitly treated as in-page scripts.

Tables with Repeating Rows

To create a table with repeated rows, the Web page designer creates a table (using FrontPage, for example) as normal. Any repeated rows are only defined once.

The programmer must insert a looping script within the table's tags. Note that if FrontPage is being used, this must be done in HTML view mode.

So the designer might create a table that looked as follows:

Date	Our contact	Notes
VisitDate	Contact	Notes
NB: 1999 visits only		

Note that in this example, the first and last rows will be written once only. Row two will be repeated on each loop, with **VisitDate**, **Contact**, and **Notes** being substituted at run-time with the appropriate values from the programmer's loop (see below).

The HTML for this table would be as follows:

```
<TABLE>
<TR>
<TD>Date</TD>
<TD>Our Contact</TD>
<TD>Notes</TD>
</TR>
<TR>
....<TD>|VisitDate|</TD>
<TD>|Contact|</TD>
<TD>|Notes|</TD>
</TR>
..<TR>
....<TD>NB: 1999 visits only</TD>
<TD></TD>
<TD>|</TD>
</TR>
</TABLE>
```

The programmer might therefore insert scripts as follows (in this example, ASP notation only is being used):

```
<%@Language=Cache@%>
<TABLE>
<TR>
<TD>Date</TD>
<TD>Our Contact</TD>
<TD>Notes</TD>
</TR>

<%
  Set date=""
  For Set date=$Order(^Contact(date)) Quit:date="" D List
  G End ; jump past the repeating row
  ;
List ; get the details
  Set d=^Contact(date)
  Set VisitDate=date
  Set Contact=$P(d,"#",1)
  Set Notes=^Contact(date,"notes")
  ; now display the row
%>

<TR>
....<TD>|VisitDate|</TD>
<TD>|Contact|</TD>
<TD>|Notes|</TD>
</TR>

<%
  Quit
End ;
%>

..<TR>
....<TD>NB: 1999 visits only</TD>
<TD></TD>
<TD>|</TD>
</TR>
</TABLE>
```

At run-time, this might be displayed as follows:

Date	Our contact	Notes
25 January 1999	Rob	Presented proposal
31 January 1999	Peter	Agreed to proceed
NB: 1999 visits only		

Note that it would be very straightforward to make any of the substituted values into [hyperlinks](#) that allowed "drilling-down" into the data; for example, by replacing |Contact| with:

```
<a href=DrillDown.asp&Contact=|Contact|>|Contact|</a>
```

When a particular contact was then clicked, the DrillDown page would be written out and the value of the specific contact would be passed to Caché in the variable **Contact** (for example, Contact="Peter" if Peter were clicked). This is a very powerful feature of Developer.

Conditional Page Sections

In-Page scripts can be used to make parts of a page conditional, i.e., they are only displayed if particular conditions hold true. In this way a page can be laid out that caters for all conditions - at run-time just those sections that apply are displayed.

To make a page section conditional, simply skip past the HTML using a series of fragments of scripts (for example, by using GoTo statements):

```
<%@Language=Cache%>
<%
  If condition1 goto Cond1
%>
<p>This is some HTML to display
...etc
<%
  Goto End
Cond1 ;
%>
<p> Here is the alternative HTML to display
...etc
<%
End ;
%>
<p>
continue with the rest of the page
```

Using ASP Syntax with FrontPage

Note that whenever you use ASP syntax, the FrontPage97 editor will show it as:

```
Language=VBScript, RUNAT=Server
```

This is simply an on-screen rendering issue. The earlier switch to the Caché WebLink Developer (Caché) ASP processing (i.e., <%@Language=Cache%>) overrides VBScript at run-time. If you change FrontPage's rendering by clicking **Language: Other** and entering "Cache", the generated HTML will revert to the <script language=Cache>...</script> syntax.

Pre-page Scripts

- [Rules for Pre-page Scripts](#)
 - [Using Pre-page Scripts](#)
-

Pre-page scripts are used to unconditionally perform an action before the page of HTML is generated for the user's browser. Their normal use is to fetch data for displaying within the page.

Rules for Pre-page Scripts

A pre-page script may actually be embedded anywhere within an HTML page. At compile-time it is extracted from the page. A page may contain **only one** pre-page script and is recognized by the compiler either:

- Explicitly, by virtue of an embedded directive; or
- Implicitly, because it is the first script found in the page (pages are compiled a line at a time, starting at the first line).
Note that an implicit pre-page script can be overridden by an explicit (embedded) directive.

A pre-page script is written as follows:

```
<SCRIPT language=Cache>
// type=pre-page
...
...lines of Caché Object Script
...
</SCRIPT>
```

Note that the language attribute must be set to "Cache" rather than JavaScript, JScript, or VBScript. Between the <SCRIPT> and </SCRIPT> tags (which can be in uppercase or lowercase), you can include any valid Caché ObjectScript.

The **type=pre-page** directive is optional; if used, place it at the start of the script. This directive ensures the script is recognized and processed as a pre-page script. It is good practice to always include this directive, particularly if a page includes many scripts.

Using Pre-page Scripts

When a Pre-page script is defined, it will be run unconditionally before a page is generated, regardless of which page preceded it. These scripts are therefore useful as a means to set or clear variables before the page begins.

Within a pre-page script, you may also set the reserved Caché variable **JUMP** to a different page name. This will cause Caché WebLink Developer to change its page pointer to the new page defined by the **JUMP** variable. Note that if this new page has its own pre-page script, it will then be executed. For example, assume the page "Process" contained the following code within its pre-page script:

```
Set JUMP="MainMenu" Quit
```

This **JUMP** setting would cause the "Process" page to be skipped, and control would pass to the "MainMenu" page. Note that if MainMenu had a pre-page script, it would now be executed.

You can also use the following reserved Caché WebLink Developer variables within pre-page scripts:

- **BACK** — When BACK is set to a value of 1, Caché WebLink Developer sets its page pointer back to the page that preceded the current page.

For example, assume the user had clicked on a hyperlink in the "Process" page to go to the "MainMenu" page. Within the pre-page script for MainMenu, suppose a condition had occurred that caused it to set **BACK=1** and **Quit**. MainMenu would

be skipped and instead, Caché WebLink Developer would cause the user to go back to the "Process" page. If the Process page had a pre-page script, the script would be executed.

Note that Caché WebLink Developer automatically clears **BACK** between each page.

- **Warning** — When the variable named **Warning** is set to a non-null value, Caché WebLink Developer sends a JavaScript alert window to the user along with the page's HTML. The action is identical to the use of the **Error** variable, except that when **Error** is set, Caché WebLink Developer automatically sets **BACK=1**, forcing the previous page to be repeated. You can use **Warning** to display warning messages while not disrupting the normal flow of pages. An example of **Warning** is:

```
Set Warning="You are about to delete this record!"
```

- **PRESSED** — For forms with multiple Submit buttons, the **PRESSED** variable holds the NAME of the Submit button that was pressed.
- **PREVPAGE** — You can detect the page where you have just been by using the reserved variable **PREVPAGE**. For example, if you have previously left the MainMenu page, **PREVPAGE** will be set to "MAINMENU". Note the page name is in upper case and does not include the ".asp" file extension. You should not change the value of **PREVPAGE** in your scripts; you should treat it as a read-only variable.



Scripts

Scripts are an important part of Caché WebLink Developer and provide the means by which your Web application can interact with the Caché environment.

The important thing to remember is that Caché WebLink Developer runs on the Caché server and that your application's HTML pages are generated within Caché. Only pure HTML is sent to the user's browser. Therefore, although Caché WebLink Developer pages are designed with Caché scripts embedded in them, these scripts only run on the Caché server, not on the user's browser.

Caché WebLink Developer supports four types of scripts:

WebLink Developer Scripts	
Pre-page scripts	Used to unconditionally perform an action before the page of HTML is generated. Typically used to pre-fetch data items to display within a page's form fields.
In-page scripts	Used to perform an action at a particular position in the HTML page. In-page scripts are mainly used for creating and controlling loops around parts of the page's HTML and for defining and/or controlling conditions under which parts of the page are displayed.
Post-page scripts	Used to perform an action after a page has been cleared from the browser (such as after a hyperlink or form Submit button has been processed), and immediately before the pre-page script for the next page is executed. Post-page scripts are only used infrequently.
Actions scripts	Contains functions/subroutines that are associated with specific events or actions, such as a form Submit button.

The following sections describe these scripts.



Post-page Scripts

Occasionally it is useful to clear up variables and perform actions at the end of a page as an event related to the page being exited, rather than as an event related to the start-up of the next page. In particular, when using Objects it is essential to close the instance of an object as soon as a page has been cleared and before the next one is displayed.

For these purposes, you may include a post-page script within a page. Note that a page can have only one post-page script. A page's post-page script is NOT run if the page's pre-page script set the JUMP variable to point to another page.

You can place a post-page script anywhere within the page. However, for visual clarity, it is recommended that you place the script at the end of the page.

You define a post-page script in exactly the same way as a pre-page script, except that you must use this WebLink Developer directive:

```
//type=post-page
```

The directive can be in upper- or lowercase.

A brief example of this directive is:

```
<SCRIPT Language=Cache>  
//type=post-page  
...  
...lines of Cache ObjectScript  
...  
</SCRIPT>
```



Actions Scripts

The **Actions** directive defines a script that contains functions and subroutines that are linked to events and actions such as submit buttons. This directive is typically used if a form Submit button includes an extended [action](#) attribute.

You can place an Actions script anywhere within the page. You define an Actions script with this WebLink Developer directive:

```
//type=actions
```

The directive can be in upper- or lowercase.

A brief example of this directive is:

```
<SCRIPT Language=Cache>
//type=actions
login() ;
  S Status=$$LOGIN^%wlduta(%App,Username,Password)
  I Status!="OK" S Error="Invalid Login Attempt"
  Q
</SCRIPT>
```

The matching Submit button would be:

```
<INPUT TYPE=SUBMIT NAME=Login VALUE=Login Action=login() NextPage=second>
```

You use an actions script block to contain any other functions you use, such as [DisplayIf](#) functions. You also use an actions script to hold functions referred to in hyperlinks of the form:

```
<a href=MyPage.asp&action=MyTest()>Click here for next page</a>
```



Re-Usable Components

- [Re-Useable Scripts](#)
 - [Re-Usable Pages](#)
 - [JUMPing to Re-Usable Components](#)
-

Re-Useable Scripts

Both in-page scripts and pre-page scripts have access to all the resources available on the Caché server, because they both run on that server. These scripts can therefore call stored routines.

If a script is likely to be used by several pages and/or Caché WebLink Developer applications, it can be written as a **standalone** Caché ObjectScript routine and called from a script, as in this example:

```
<SCRIPT language=Caché>
//type=pre-page
Do ^MyRoutine
</SCRIPT>
```

This example would execute the standalone routine **^MyRoutine** as a pre-page script.

An advantage of making a call to an external routine from a script is that changes can be made to the routine without requiring the page to be recompiled. The disadvantage is that the logic of the page is now in two places — the .asp file and the Caché routine. You must decide what your priorities are. At run-time there is unlikely to be any performance difference in either approach.

Re-Usable Pages

Each Caché WebLink Developer application consists of a set of pages within its own directory. However, there are times when it would be useful to be able to re-use some pages that you have already developed for another application.

Caché WebLink Developer provides a mechanism to develop and run re-usable pages. The procedure is as follows:

1. Create a new application directory for your component pages. For example, if **C:\InetPub\wwwroot** is your application root directory, create the directory:

```
C:\InetPub\wwwroot\Components
```

2. Construct the pages that make up your re-usable component and save them in this new directory.
3. Compile the pages in the new directory. Note that you may have many components or sets of component pages within the one directory.
4. Go to the page in your main application from which you want to call up the component pages. For example, your "Contacts" application has a page called "Process" in which you want to place a hyperlink to a re-usable page. This re-usable page is called "Mail" and has been saved in the "Components" application directory.

Create the hyperlink in the "Process" page; for example:

```
<a href="Components/Mail.asp">Click here to create an email</a><br>
```

The key thing to note is the inclusion of the application directory name "Components". If the Caché WebLink Developer compiler finds a reference to a ".asp" file which is preceded by a directory name, it creates a dynamic "component" link to the specified page within the specified component directory.

5. To create the return path from the component page back to the application that invoked it, you use the reserved Caché WebLink Developer page name **RETURN**.

For example, within the "Mail" page in the "Components" directory, include the hyperlink:

```
<a href="return.asp">Click here to go back</a><br>
```

Note that the page "return.asp" does not physically exist. In fact, Caché WebLink Developer treats this just like a RETURN command (for example, in a BASIC subroutine) and returns control back to the calling application - the page that contained the call to the component page is sent to the user's browser (and any pre-page script for that page is executed). In our example, clicking this hyperlink would return the user to the "Process" page in the "Contacts" application.

Note that Caché WebLink Developer internally stacks its application context on each call to a re-usable component, so that one component can call another - in theory to any level of stacking. However, each component can only return to the page that first called it - you can only unwind one level of the context stack at a time.

A re-usable component can consist of one or more pages. In our example, the page "Mail" could call another page "MailSend". Both pages would exist in the "Components" directory and "Mail" would simply refer to "MailSend" via the "MailSend.asp" URL; for example:

```
<a href="MailSend.asp">Click Here to send the email</a><br>
```

Because Caché WebLink Developer has reset its application context to "Components", any ".asp" page that is referenced is now assumed to exist in the "Components" directory. Any page within the "Components" directory can invoke a return to the calling directory ("Contacts" in our example). So the "MailSend" could include a hyperlink to "return.asp" which would return the user back to the "Process" page in the "Contacts" application.

JUMPing to Re-Usable Components

You can set the [JUMP](#) variable to invoke a call to a reusable component. For example, you can use the following in a pre-page script:

```
SET JUMP="MyComponents/MyPage.asp" QUIT
```

This will cause WebLink Developer to stack the current application and jump to **MyPage** in the **MyComponents** application. (Note neither the page nor the component application name is case sensitive).

Provided **MyPage.asp** quits using **Return.asp**, control will be returned to the calling application. However, control will be returned to the page that preceeded the one whose pre-page script set the **JUMP** variable.

As an example, in the Application "Contacts", a submit button in a form in **Page1.asp** passes control to **Page2.asp**. Suppose the pre-page script for **Page2** sets **JUMP** as shown in the example above. WebLink Developer now jumps immediately to **MyPage** in the "MyComponents" application. If the "Go Back" hyperlink in MyPage is set to **Return.asp**, then when this hyperlink is clicked, the user will be returned to **Page1.asp** in "Contacts". Visually, this is what the user will expect because **Page2.asp** was never actually seen - in fact if control returned to **Page2.asp**, the user would end up back in **MyPage**.



Running Applications

- [How Applications Run and How Security is Implemented](#)
 - [Quitting from Applications](#)
 - [Enabling the Browser's Back Button](#)
-

How Applications Run and How Security is Implemented

A Caché WebLink Developer generated application is started using a URL of the form:

```
http://www.myserver.com/scripts/mgwms32.dll?MGWLPN=My_LPN&wlap=Contacts
```

By default, Caché WebLink Developer applications run in WebLink's [Stateless Mode](#).

The WebLink routine **^%MGW3** is already configured for Developer and includes a line after the label HTML that passes control to **^%wld** (the run-time engine) when WebLink sees a non-null value for the **wlap** variable. The run-time engine:

1. Identifies the First page and namespace for the application.
2. Executes any pre-page script.
3. Writes out the HTML that was in First.asp (or whatever the First page has been called).

Any hyperlinks of the form:

```
page.asp
```

will already have been converted to WebLink URLs at compile time, but will now include a random 30-character token generated within the run-time module within Caché (The length of the tokens can be reconfigured if required).

The Caché back-end process retains pointers that indicate the action to be carried out when these tokens are received again. The tokens are flushed between pages and recreated from scratch when each page is written. Tokens are therefore unique to an individual transaction for an individual page for an individual user.

If an unrecognized token is received, the original page is re-displayed with [Error](#) substituted with "Invalid Action". This makes applications extremely secure and immune to the user pressing the Back and Forward browser buttons, because the application will always resynchronize itself to the page to which it is expecting a response.

The user's session is flagged as started when the First page is sent to the user's browser. For State-Aware applications, from this page onwards until shutdown of the private session, the background Caché session is running under the control of **^%wld** within its main READ loop.

Quitting from Applications

To exit from an application, you use a hyperlink of the form:

```
<a href="Quit.asp&GoToPage=../default.htm">Quit the application</a>
```

This will cause the current application to be shut down and the user to be sent to the Home Page.

The page **Quit.asp** does not physically exist. Instead, it is a reserved page name that tells Caché WebLink Developer to close the current application.

Note that the page defined by the **GoToPage** parameter is deemed to be relative to the Web server's /scripts directory. For example, on Microsoft's IIS Web server on NT4, the file "default.htm" would be expected to be found in the **<root> or <home>** directory, so must be referred to as ../default.htm.

You can also specify static pages in other directories; for example:

```
<a href="Quit.asp&GoToPage=../details/page1.htm">Quit the application</a>
```

Clicking on the hyperlink would quit the application and return the user to **page1.htm**, which would be found in the Web server's **/details** virtual directory.

You can also use a form submit button to exit the application; for example:

```
<INPUT TYPE=SUBMIT NAME=EXIT Value="Quit" NextPage="Quit&GoToPage=../default.htm">
```

When an application is exited, its session context flags and variables are cleared out. If it is a State-Aware application, the compiler will substitute the correct WebLink URL parameters needed to shut down the private session and return the user to the Home Page. To protect against a user leaving his browser for a long time and hogging a Caché process, [client pull](#) can be used.

Enabling the Browser's Back Button

By default, WebLink Developer prevents the user from using the browser's Back button. However, it is possible to configure Developer to allow the Back button to work. In fact, the implementation is quite sophisticated and it is important to understand the rules that Developer applies when working in this mode. It aims to create a sensible and unambiguous result for you, the programmer, when the Back key is pressed, so it should be clear what programming you need to perform to cater for the situation whenever it occurs.

Security Caution: Please be aware that if you enable the Back button, your application's security will be significantly compromised because you are allowing the re-use of previously-used tokens.

However, the additional security mechanisms applied by WebLink Developer **may** provide you with sufficient security for your application. The use of [SSL](#) will further help to protect your application. Before you use this feature, make sure that you understand all the security implications.

You enable the Back button on an application-by-application basis, as follows:

1. Open the Caché WebLink Developer Maintenance Suite page.
2. Select **Configure an Application**.
3. Select the **Security Model** option.
4. From the drop-down list, select **Token-based security, Back-button enabled with symbol table wind-back**.
5. Save the configuration parameters.
6. Recompile the application.

Keep the following notes in mind for applications with enabled Back buttons:

- When the browser's Back button is pressed, a cached version of a previously-sent page is recovered by the browser. If you click on any hyperlink or form on this page, Developer will reinstate the application's symbol table to the state it was in when the cached page was originally written (i.e., any subsequent changes to the symbol table are discarded).

Note: It is up to your application to unwind any changes subsequently made to globals, because WebLink Developer can only reinstate the local variable pool.

- WebLink Developer is intelligent enough to take care of multiple Back button presses. However if the user presses the Back button so often that a page appears for a logged-out or timed-out application, then if an attempt is made to use such a page, WebLink Developer sends an error page letting the user know that the page belonged to a session that has now been shut down.
- WebLink Developer can also cope with Forward button presses. However, remember that browsers flush their forward cache as soon as a hyperlink or form is clicked. For example, if you press the Back button three times, you now have three pages in the forward cache to which you can return. However, if you now click on a hyperlink in the cached page, the appropriate page will appear, but the three pages in the forward cache are lost and the browser will not let you access them again. This behavior is controlled by the browser, not by WebLink Developer.
- The behavior of the pseudo-page Back.asp also changes automatically to closely emulate the Back button behavior. That is, it winds back the symbol table to its status prior to the previous page being displayed, then re-runs the previous page (including its pre-page script). For example:

```
<a href=Back.asp>Go back to previous page</a>
```

Note that there *is* a difference here: the browser's Back button recovers the previous page from cache so its pre-page script will not have run. However, the Back.asp mechanism tries to recreate the situation when the previous page was originally written, so it reinstates the symbol table to its status just before the moment the previous page's pre-page script ran. It then re-runs the previous page's pre-page script and then draws the page. Under most circumstances there will be no difference, but if the pre-page script (or indeed any of the in-page scripts) has any time dependency, there will be a mismatch in behavior.

- To make this behavior possible, WebLink Developer has to save a snapshot of each application's symbol table before and after each page. As a result, it requires a considerable amount of global space, particularly if you have many users running simultaneously and the application writes many pages between logon and logout. However, the snapshot files are automatically cleared out whenever a Quit.asp is processed or the application times out.

Comparison with Developer's default behaviour:

Unless you specifically set the application configuration parameter, WebLink Developer will continue to resynchronize the browser with the back-end whenever the Back key is pressed. In this default mode, the Back.asp mechanism does NOT wind back the symbol table (though it does wind back the page stack). All variables are persistent irrespective of the direction of navigation.



Application State Modes

Applications can run in either Weblink's Stateless or State-Aware Modes. By default, applications run in Stateless Mode. In Stateless mode Developer simulates a stateful environment by saving and restoring the symbol table between each page. In stateless mode there is no guarantee that the same back-end process will be used for each subsequent page/transaction, so it is not possible to support true transaction processing with locking of resources. If you require such transactional integrity, you must use Weblink's state-aware mode.

To force an application to run in State-Aware mode, use the **Configure an Application** option in the Caché WebLink Developer Maintenance Suite. The resulting menu has a radio button to select Stateless or State-Aware mode. Set the mode, submit the form, quit, and recompile the application. It will now run in State-Aware mode. You do not have to make any other modifications to your application.

Starting WebLink Developer Applications

Caché WebLink Developer applications are started using the URL:

```
http://mysys.com/scripts/mgwms32.dll?MGWLPN=CacheServer&wlap=AppName
```

where:

mysys.com	is substituted with the domain name or IP address of your Web server.
/scripts/mgwms32.dll	invokes Weblink. Note that this call will differ if you are using the Apache web server and/or the standalone Weblink Network Service Daemon.
CacheServer	is substituted with the name by which Weblink refers to your Caché server (i.e., the Login Profile Name).
AppName	is substituted with the name of your Caché WebLink Developer application. This is case insensitive.

For example, if your Web server IP address is 192.12.34.56 and your Caché server's Login Profile Name is "LOCAL", and you create and compile an application called **Contacts**, you could use the following hyperlink to invoke the application from one of your Web pages:

```
<a href="http://192.12.34.56/scripts/mgwms32.dll?MGWLPN=LOCAL&wlap=contacts">Run the contacts database</a>
```

This hyperlink could be further reduced to:

```
<a href="/scripts/mgwms32.dll?MGWLPN=LOCAL&wlap=contacts">Run the contacts database</a>
```

This assumes that the Web server that served up the calling page was the same as that to be used for the application.

The page that is configured as the application's first page will be written to the browser. By default this will be the page named "First.asp", but this may be changed in the application's configuration settings.

Timing Out Sessions

- [Timing Out Private Sessions](#)
 - [Timing Out Stateless Applications](#)
 - [How Sessions Are Cleared](#)
-

Timing Out Private Sessions

A key problem with Web applications, in particular when using private State-Aware sessions, is how to shut down a background Caché private session if the user does not respond. This may happen because the user does not complete a transaction, or because the user's Internet connection has dropped. If the background Caché process is not closed down, it will continue to run, thereby needlessly using a Caché user license.

There is also the problem of how to unwind any partially-completed transactions — you may not be able to halt the background process.

Caché WebLink Developer allows you to use two techniques to shut down private sessions:

- Browser-level "client pull"
- WebLink-level client time-out

Browser-level Client Pull

Client pull is an ideal way of handling the situation where the user is still online with the browser displaying the last page, but for some reason has failed to respond within a maximum time limit. You can time a page out by using a **META** tag.

To specify a META tag in Microsoft FrontPage:

1. Right-click your mouse and select **Page properties** from the drop-down menu.
2. Select **Custom** and click the **Add** button beside the top box (titled "System Variables - HTTP-EQUIV").
3. In the Name field, type **Refresh**.
4. In the Value field, type something like:

```
300;URL=MainMenu.asp
```

If you are typing in the HTML, you should insert the following line between the <HEAD> and </HEAD> tags:

```
<META HTTP-EQUIV="Refresh" CONTENT="300;URL=MainMenu.asp">
```

This META tag means time-out the page after 300 seconds and jump to MainMenu. Because the .asp convention has been used, the compiler will convert this to a WebLink dynamic URL. In this example, the main menu would then be displayed.

You can also use the following for the URL:

```
URL=Quit.asp&GoToPage=default.htm
```

With this specification, when the browser timed out, the user's session would be closed down and the home page would be sent to the user's browser. This is a very useful security method.

WebLink-based Client Timeout

Client pull only works if the user's browser is running, the last page is being displayed, and the user's IP connection is in place. Remember, with a Web application, the Caché back-end can only send a page to a browser in response from the browser asking for it (i.e., the back-end cannot write to a browser when the back-end feels like it). Therefore, if no response is received from the browser, the only recourse is to halt the back-end Caché process.

In the current version of WebLink Developer, there is no facility to invoke a shut-down process to controllably sort out part-completed transactions. In addition, the current version is limited in its flexibility in terms of the client time-out period.

To activate WebLink-controlled client time-out, you must either set the **Client_Timeout** parameter (using the WebLink Systems Management page for configuring the Caché System parameters) or define a value for a reserved Caché variable **MGWCTO** within one of your pre-page processing scripts (probably First.asp). By setting one or other of these values to, say, 1800 (seconds), the user's back-end Caché private process will automatically shut down if no response is received within 30 minutes of the last page being sent to the user's browser.

Probably the best approach is to use both techniques together, with browser-based client pull activating first, and WebLink client time-out occurring later. Note that any remaining WebLink client time-out period is reset on each Web page refresh.

Timing Out Stateless Applications

Although there is no session to time out as such with stateless applications, Caché WebLink Developer creates its own session context during which local variables persist between pages. If user access security is included (i.e., the user is asked for a username and password), the user can continue to use successive pages during the session context. Unless the session context is closed, local variable stacks will continually accumulate and a user will continue to have access.

The **Configure an Application** option in the Caché WebLink Developer Maintenance Suite allows the session time-out to be set. If the user does not respond within this time period, the session becomes tagged for closure. The standard ASP default of 20 minutes is used unless over-ridden. Hence a user session is deemed to have expired if the user has not responded to any screen within 20 minutes. However, if, after 20 minutes, no other user has accessed a Caché WebLink Developer application, then the session's owner may be able to continue with his or her application.

How Sessions Are Cleared

Sessions that have timed out are cleaned up by a shut-down process that can be triggered in one of two ways:

- The default method. This method takes place whenever a new session is started. The first thing that happens is for any timed out sessions to have their local variable pool and stacks, token stacks, and session pointers deleted.
- Start the Session Shutdown Demon. To reduce the start-up overhead (particularly with busy systems), a Session Shutdown Demon can be started. If demon is running, it takes on the responsibility for clearing timed-out jobs. By default the demon sleeps for 5 minutes before rechecking for more jobs to be terminated. This sleep period can be reset if required.

To use the Session Shutdown Demon, select the **Configure WebLink Developer** menu option in the Caché WebLink Developer Maintenance Suite.

If you wish, you can ensure the ShutDown Demon is always started with Caché by putting the following command into the ^ZSTU startup routine:

```
JOB ^%wldssd
```

Note, however, that WebLink Developer will automatically try to ensure it is running at all times and will restart it automatically when the Developer run-time engine is run.

You can check the status of the ShutDown Demon in two ways:

- Use the **Configure WebLink Developer** menu option in the Maintenance Suite.

- Bring up the Caché system status (for example, using ^%SS) and look for the job running the ^%**wldssd** routine.



SSL

- [Configuring SSL](#)
 - [Configuring IIS to Support SSL with WebLink](#)
-

Caché WebLink Developer supports the use of SSL (Secure Sockets Layer) for secure, encrypted communication between the Web browser and Web server.

Configuring SSL

To use SSL, you must first set up Caché WebLink to support SSL:

1. If you are using Microsoft's IIS Web server, you must define a directory for SSL, and you must map this via the Registry to WebLink's DLL.
2. From WebLink Developer's "Configure WebLink Developer" option, select **WebLink SSL URL call**.
3. Enter the SSL URL that you have configured for WebLink.

For example, suppose you are using IIS and have set up an alias for your SSL directory of **/WebLinkSSL**. Also suppose you have used a Registry Script Map for the file extension ".wls" to invoke WebLink. At the **WebLink SSL URL call** field, you would enter **/WebLinkSSL/web.wls**. (The name "web" can be replaced by any name you like.)

4. Save your new configuration parameters.

The URL that you have specified will now be substituted by WebLink Developer's compiler into any SSL calls you make. You specify these in your application as follows:

- For hyperlinks: simply add "https://" (case insensitive) in front of the page reference, e.g.:

```
<A HREF=https://second.asp>Click here</a>
```

If you are using FrontPage97 or FrontPage Express, when you select **Hyperlink Properties**, enter the page name (e.g., second.asp) into the **URL:** field and select from the drop-down field **Hyperlink Type:** the option **https**. FrontPage will automatically add "https://" to the start of your specified URL.

- For forms: specify **Action=HTTPS://** within the <FORM> tag. For example:

```
<form action=https:// method=post>
...etc
</form>
```

If you are using FrontPage97 or FrontPage Express, select **Form Properties** and click on **Settings**. In the "Action:" field, enter **https://**.

- For other URLs (such as in <META> tags): add **https://** before the page name. For example:

```
<META HTTP-EQUIV=Refresh content="0;URL=https://first.asp">
```

To switch back to normal HTTP, simply leave out the **https://** from the URLs and Form tags.

Configuring IIS to Support SSL with WebLink

Under IIS, SSL is configured against directories. Therefore, the key to implementing these security requirements is to define an additional (virtual) directory for the WebLink applications requiring SSL. SSL is then configured for that directory. Applications not requiring SSL continue to use the conventional WebLink directory and file combination (i.e., /scripts/mgwms32.dll).

It is possible to set up a virtual directory for WebLink that can act as the focus of communications for the purpose of SSL. To do so, use this procedure:

1. Create your **SSL** directory under the Web server's root (**wwwroot**) directory. For example:

```
InetPub\wwwroot\WebLink_SSL
```

Although it may not be necessary to physically create this directory, it is recommended that you do so. The Web server will see this directory as virtual.

2. Create a new (unique) file extension. You can use any file extension that is not already in use by other applications (such as .doc and .txt). In this example, "wls" will be used as the WebLink-specific file extension.
3. Use the new extension to instruct the Web server to have WebLink process files of this type. To do this, use **RegEdit** (the NT Registry Editor) to add the following item to the registry:

Key name:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\Script Map]
```

Value:

```
".wls"="C:\Inetpub\scripts\mgwms32.dll"
```

This specifies that WebLink to be used to process all URLs that have the .wls extension.

4. Enter the main WWW configuration for IIS:
 - a. Enter the "Directories" section and **Add** a new entry. The directory name should be the full path to the physical directory set up in step 1; for example:

```
C:\InetPub\wwwroot\WebLink_SSL
```

- b. Make sure the 'Virtual Directory' radio button is checked.
- c. Set the alias. This will be the path you use in your URLs. For now, set it to:

```
/WebLink_SSL
```

Executebox is checked at the bottom of this form.

5. Notice also the SSL set-up at the bottom of the directories form. This will need attention when SSL is enabled for the Web server (i.e., when entering your digital certificate details); for details, refer to your Web server's documentation.

- Shut down and restart the computer.

You will now find that WebLink can be called by:

```
http://127.0.0.1/WebLink_SSL/secure_document.wls
```

as well as:

```
http://127.0.0.1/scripts/mgwms32.dll
```

Note that the file name *secure_document* can be any name of your choosing (it does not have to physically exist). The important part is the extension (.wls).

Now, all you need to do is set up and enable SSL-managed data streams for the **/WebLink_SSL** directory. "Non-secure" WebLink users can continue to use the conventional URL form:

/scripts/mgwms32.dll

Secure connections will use URLs of the form:

https://127.0.0.1/WebLinkSSL/web.wls?MGWLPN=local&...etc.



Templates

- [Purpose of Templates](#)
 - [Defining Templates](#)
 - [META Tags in Templates](#)
-

Templates are a very powerful feature of Caché WebLink Developer. WebLink Developer allows you to define one or more template files that determine the basic layout or look and feel of every page that makes up an application. The template will also be applied to any re-usable component pages used within an application.

An application may have more than one template. With multiple templates, different users are presented with a different look-and-feel for the same application.

Purpose of Templates

Template files define the background color or image used for every page, as defined in the template file's <BODY> tag. Any other content of the template file after the <BODY> tag and up to (but not including) a special <CONTENT> tag is also used in every page within the application.

The <CONTENT> tag in the template file denotes the point at which each application page contents (between the application page's <BODY> and </BODY> tags) are inserted. Anything else in the template file from (but not including) the template file's </CONTENT> tag to its </BODY> tag is also displayed on every application page. Thus a template file can define the background, headers, and footers for application pages.

For example, suppose a template file contained the following:

```
<HTML>
<HEAD><TITLE>Template file></TITLE></HEAD>
<BODY bgcolor="#AAFFD5">
<H1>This banner will appear on every page</H1>
<IMG SRC="cachelogo.gif">
<CONTENT>
This text is ignored, and substituted by each application page, between (but not including)
its BODY and /BODY tags
</CONTENT>
<H1>This footer will appear on every page</H1>
</BODY>
</HTML>
```

Every page in the application would have a pale blue background and the header and footer defined in the template file, in addition to the Caché logo. Note that the template file's title is not used — the title for each application page is used at run time. Also note that everything between the <CONTENT> and </CONTENT> tags in the template file is ignored.

Defining Templates

There are three methods of defining templates:

- If the file **TEMPLATE.ASP** exists within the application's directory, it is used by Caché WebLink Developer automatically and applied to every page within the application. Note that the name **TEMPLATE.ASP** is *not* case sensitive.
- If a template file of a different name is defined with the **Configure an Application** menu option in the WebLink Developer Maintenance Suite. This page is applied in preference to the **TEMPLATE.ASP** file if it exists.
- If the URL that initiates a Caché WebLink Developer application includes the parameter:

&template=FileName

In this case, the template file defined by *FileName* is applied to every page within the application. (You do not have to specify the ".asp" file extension.)

Note: This template is applied in preference to **TEMPLATE.ASP** (if it exists) and/or any other template file defined for the application.

For example, the URL:

```
/scripts/mgwms32.dll?MGWLPN=MyServer&wlap=Demo&template=tmp3
```

would result in the template file **tmp3.asp** being applied to every page in the Demo application. Tmp3.asp would be expected to be found in the Demo application directory.

META Tags in Templates

<META> tags can be used for a variety of usages. A frequent use is to provide information on the author of a page and to indicate any copyright information, as in this example:

```
<META Name="GENERATOR" Content="M/Gateway Developments Ltd, London, UK">  
<META Name="DATABASE" Content="InterSystems Cache">  
<META Name="Author" Content="Rob Tweed (rtweed@imtcons.demon.co.uk)">
```

<META> tags are placed in the <HEAD> section of a Web page.

WebLink Developer will copy to every page in an application any <META> tags that you place in a template page. This saves you manually copying the <META> tags into every page of an application.



Caché Objects and SQL

Caché WebLink Developer has been designed to work with SQL and the Caché object technology. SQL and Caché ObjectScript syntax can be used within the WebLink [scripts](#). However, the default compiler option must be re-set to ensure the scripts are properly compiled as MAC format rather than INT format. You do this from the Caché WebLink Developer Maintenance Suite, and must be done for each application.

If you are using Objects, you should map these, within your pre-page script(s) into simple variables that match any form element field names, and map the variables back into Objects within your Action script(s) that is/are associated with the form's submit button (s).



Using Frames and Windows

- [Initial Loading of Pages into Frames](#)
 - [Running an Application in its Own Browser Window](#)
 - [Opening and Closing Browser Windows](#)
-

Initial Loading of Pages into Frames

If your application uses frames, you need to know how to initially load the pages into the frames.

For example, suppose you had two frames (named "Upper" and "Lower") in a frameset document named **MyFS.asp**. If you want to initially load **Page1.asp** into "Upper" and **Page2.asp** into "Lower", you can use one of two methods:

- Standard loading method
- Cascade loading method

Standard Loading Method

The standard method is to use the <FRAME> tag to load each page into its target frame, using the **SRC=** parameter. For example, **MyFS.asp** would contain:

```
<FRAMESET ROWS="*,*">
<FRAME NAME="Upper" SRC="Page1.asp">
<FRAME NAME="Lower" SRC="Page2.asp">
</FRAMESET>
```

Under normal circumstances, Developer's default action is to destroy all a session's tokens whenever a page is unloaded. This would, of course, prevent frames from working. However, Developer can automatically recognize when you are using frames and keeps alive all tokens on all currently active frames.

Cascade Loading Method

If you have a limited number of WebLink connections, loading the pages in a cascade (using JavaScript) is an effective process. This procedure uses the above sample pages:

1. In the FrameSet document, set the **SRC=** parameter for "Lower" to be a blank static page (e.g., **blank.htm**), and the **SRC=** parameter for "Upper" to be **Page1.asp**.
2. In **Page1.asp**, include an event handler on the <BODY> tag to cause **Page2.asp** to be loaded into the "Lower" frame:

```
<BODY Onload=LoadPage2(>
```

3. Also in **Page1.asp**, include the following JavaScript function:

```
<SCRIPT Language=JavaScript>
function LoadPage2() {
    top.Lower.location = "Page2.asp" ;
} </SCRIPT>
```

In the JavaScript function, note the case sensitivity of the frame name in the object reference. You could alternatively use:

```
top.frames[1].location = "Page2.asp" ;
```

because in this example, "Upper" equates to **frames[0]** and "Lower" is **frames[1]**.

If required, you can pass name/value pairs into **Page2.asp** (for example, to ensure its [pre-page script](#) recognizes the first-time load of this page). To pass name/value pairs, add them as in this example:

```
top.Lower.location = "Page2.asp&firstload=true" ;
```

At run time in the pre-page script, the Caché variable **firstload** will be set to "true".

If you have three or more frames, load Frame3 from Page2, Frame4 from Page3, and so on. This cascading method is more efficient on WebLink connections, because a frameset calling multiple .asp pages would force multiple connections to be opened (because they hit the Web server in parallel). When done in a cascade, the one connection can support each successive call.

Running an Application in its Own Browser Window

You can configure an application to start up and run in its own browser window, and you can control the appearance, size, and position of the window.

To configure the application:

1. Open the Caché WebLink Developer Maintenance Suite page.
2. Select **Configure an Application**.
3. Towards the bottom of the form that appears are a series of questions relating to running the application in its own window. Select the appropriate parameters.
4. Save the new configuration parameters.
5. Recompile the application.

When you next run the application, it will run in its own window. When the application quits, the window will automatically close.

Opening and Closing Browser Windows

In addition to frames, an application can open new instances of browser windows, and it is possible to control their appearance, size and position on the screen. This normally requires a knowledge of JavaScript, but WebLink Developer makes this task easy by automatically generating the necessary JavaScript for you.

Once opened, there is no functional difference at the HTML and JavaScript level between a frame and a window, because they are each addressed via a target name.

Opening a Window

To open a new window within an application, you use a Developer-specific target name within a Hyperlink or form:

```
target=window(name=TargetName,{optional parameter list})
```

Optional parameters include all the standard JavaScript **window.open** feature list:

x=x-coordinate of top left corner (in pixels from the top),
y=y-coordinate of top left corner (in pixels from the left),
width=window width in pixels,
height=window height in pixels,
toolbar=yes/no, {display browser toolbar}
location=yes/no, {display browser location window}
resizable=yes/no, {make window resizable}
directories=yes/no, {display browser directory buttons}
status=yes/no, {display the browser status line}
menubar=yes/no, {display the browser menu bar}
scrollbars=yes/no, {add scrollbars if necessary}

The TargetName is what you use to subsequently target pages into this window from other windows/frames.

Optional parameters default to "no", except scrollbars and resizable which default to "yes", width and height default to 400 and 600 pixels respectively, and x and y which both default to 0.

- e.g., in a hyperlink:

```
<a href="MyPage2.asp" target="window(name=MyWindow,x=50,y=50,width=750,height=350)">Open Page  
2 in a new window</a></p>
```

- e.g., using a form's submit button:

```
<input type="submit" name="B2" value="Open page 2 in its own window" nextpage="page2"  
target="window(name=MyWindow2,width=750,height=350)">
```

Closing a Window:

To close a window, use the Developer pseudo-page "Close.asp".

- For example, in a hyperlink:

```
<a href=Close.asp>Close the current window</a>
```

- or in a form's submit button:

```
<input type="submit" name="B1" value="Close the window" nextpage="close">
```



HTTP Headers and Cookies

- [Inserting HTTP Header Code](#)
 - [Using Persistent Cookies](#)
-

This section describes how to insert HTTP header code and cookies in Developer-defined pages.

Inserting HTTP Header Code

With WebLink Developer, you have two methods of inserting HTTP header code into an HTML page generated by a WebLink application:

- using the `<HTTP>` tag
- using an HTTP script

Using the WebLink Developer HTTP Tag

You can include HTTP headers in Developer-defined pages by using the WebLink Developer `<HTTP>` tag. The syntax of this tag is:

```
<HTTP>
HTTP header info here
blank line here
</HTTP>
```

You can place the `<HTTP>` tag anywhere in the page; however, at compile time it is moved to the start of the page. You can also use variables within the HTTP header, as in this example:

```
<HTTP>
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: MyCookieName=MyCookie|number|; path=/; expires= Fri 08-Jan-1999 13:00:00 GMT

</HTTP>
<HTML>
```

The HTTP Header must be complete and properly formed with a mandatory blank line before the `</HTTP>` closing tag. Note that the WebLink Developer compiler ensures that there is an **HTTP/1.0 200 OK** entry at the start and a blank line at the end, even if you omit them.

Note that this example shows one of the uses of HTTP headers: [adding cookies](#).

Using HTTP Scripts

A second method of defining HTTP headers is to insert them as a script of the form:

```
<SCRIPT language=HTTP>
HTTP header info here
blank line here
</SCRIPT>
```

This script can be placed anywhere in the page. While the HTTP header content is case sensitive, the `</SCRIPT>` tags and

parameters are not. Note that if the final blank line is omitted, WebLink Developer will add it.

If you are using Microsoft FrontPage, you can use the **Insert/Script** menu option to create and edit the script.

Using Persistent Cookies

Persistent cookies are often used by Web developers as a means of maintaining state (which is the reason they were originally invented). However, many Web users disable cookies in their browsers because they distrust cookies. Because we do not want to force users to enable cookies in their browsers, WebLink Developer uses its own mechanism to maintain state (i.e., without using cookies). This mechanism uses *tokens* (generated by the back end) that are returned to the browser with every hyperlink or form.

However, cookies still can have an important role to play. For example, you can use cookies to recognize a user who has returned to your Web site and restore a particular user profile. Cookies also allow you to potentially dispense with a username and password. If you get a recognizable cookie back from a browser, you can obtain user information from it.

Before explaining how to use cookies with WebLink Developer, a few words of caution:

- Because users can disable cookies on their browsers, you must determine how to cope with a cookie-disabled browser when you are making decisions based on cookies.
- A cookie identifies the browser, but not necessarily the user. Therefore, you cannot depend upon cookies for unequivocal user authentication.
- Netscape Communicator and Microsoft Internet Explorer implement cookies differently. People who use both browsers may have a cookie filed for one browser but not the other.
- For persistence on the client, cookies are stored in a ordinary text file (**cookies.txt** in Netscape). There is nothing to prevent a user from editing a cookie file and/or copying a cookie file to another computer.

Programming Cookies with Developer

Cookies can be set on the user's browser by sending a **Set-Cookie** directive in the HTTP header of a page sent to the user's browser. Their use is best explained with this simple example:

1. In the first page of an application, add this HTTP script:

```
<script language="HTTP">
//type=HTTP
Set-Cookie: Testing=|DH|; path=/; expires=Mon, 20-Sep-99 11:00:00 GMT;
</script>
```

The format of the date and time must be expressed relative to GMT. Note that WebLink Developer will automatically create the other lines needed in an HTTP script (i.e., the "HTTP/1.0 200 OK" line and the blank line before the HTML stream). You must get the syntax of the **Set-Cookie** directive absolutely correct or it will not work.

2. Make sure the variable **DH** is set in the page's [pre-page script](#); for example:

```
<script language=Cache>
//type=pre-page
SET DH=$HOROLOG
QUIT
```

3. When this page is sent to the browser, it creates a cookie on the browser with whatever the value of \$HOROLOG was at the time the pre-page script executed; for example:

```
Testing=57596,37557
```

4. When you move to the next page of the application (via a hyperlink or form), the browser sends the cookie back to WebLink and Caché. The WebLink **%CGIEVAR** array should contain the above value in the CGI **HTTP_COOKIE** environment variable; for example:

```
%CGIEVAR( "HTTP_COOKIE" )="Testing=57596,37557"
```

The back-end Caché application can make decisions based on the cookie that was returned (particularly since it created and sent the cookie to the browser in the first place).

Because an expiry date was specified, the cookie is filed on the client and will be sent back to the browser on every hit until the expiry date is exceeded. **Note:** If an expiry date had not been specified, the cookie disappears when the browser is closed. This is a common mistake made with cookies.

If you specified a valid expiry date, try quitting the application and shutting down the browser. Then start the browser and the WebLink Developer Web interface, select the **Session Status** option and view the variables for the current session. You should find the browser still returns the original cookie as:

```
%CGIEVAR( "HTTP_COOKIE" )="Testing=57596,37557"
```

If you restart your application again, the cookie will be updated with the latest value of \$HOROLOG.

Note that because Netscape and Microsoft browsers implement cookies differently, each will recall only the cookie last sent to itself.

Obviously, you have complete control over the name and value of the cookies you wish to send to the browser, making them quite powerful. It is for this reason that most of the Web search engines and portal sites use cookies to recognize users.

Some Key Features of Cookies

Keep in mind the following features of cookies:

- A browser will only send a cookie back to the Web server that sent it in the first place. Therefore, your Caché application only has access to cookies it created.
- You can specify a path for a cookie. If you do, it will only be retrieved by requests using the same path or a subdirectory of it. Use the HTML **PATH=/** attribute if you always want to retrieve the cookie irrespective of the URL requested by the browser.
- Browsers have a limit to the number of cookies they can store, but it is large enough so that it is usually not a problem.
- When a WebLink Developer application creates a cookie (e.g., via **Testing=\$H**), a cookie named **Testing** is updated each time. Now assume that another Developer application creates another cookie; for example:

```
<script language="HTTP">
//type=HTTP
Set-Cookie: Cookie2=|XYZ|; path=/; expires=Mon, 14-Sep-98 11:00:00 GMT;
</script>
```

The second cookie (named **Cookie2**) is created on the browser with whatever the value of the variable **XYZ** was at run-time. Note that the **Testing** would still exist.

- On Netscape browsers, check the **cookies.txt** file to see what is happening on the browser.
- Your Caché application can delete a cookie on a browser by re-sending it with a expiry date that is in the past.

As you can see, cookies can be a powerful and useful tool if you want to implement some kind of state (or other user-related) persistence *between* sessions. Provided you are aware of the limitations, WebLink Developer provides an uncomplicated interface that allows their use in your applications.

Using the Event Broker for Dynamic Pages

- [Caché-side Field Validation](#)
 - [Handling Form Field Types](#)
-

WebLink Developer makes use of WebLink's Event Broker to provide a range of powerful functions, which are described below. WebLink's Event Broker provides a TCP socket connection between the browser and WebLink on the Web Server, and hence to/from the Caché server. When an event occurs on the browser (e.g., OnClick or OnChange), a request can be sent from the browser to the Caché server to carry out some processing and return results to the browser. The browser's page contents can then be updated (via JavaScript).

For a more detailed overview of the Event Broker, refer to the *Caché WebLink Guide*.

Caché-side Field Validation

WebLink Developer uses the Event Broker to allow Caché to control:

- field validation
- multiple form field value substitution
- refocusing

The mechanism provided uses WebLink Developer's token-based security mechanism and provides full read/write access to the Caché application's symbol table.

Implementing Field Validation

WebLink Developer provides a very simple but powerful interface to make use of the Event Broker for field validation and value substitution. The interface described below uses the **ActionOnChange** event handler and an [Actions script](#).

To illustrate this interface, suppose you had a form with four fields named:

- Username
- Password
- Info
- Submit (i.e., the submit button)

You want to make Caché check the Username field when the user moves focus to the Password field. To do this, you:

1. Add a Developer-specific **label** to the <INPUT> tag for the field to be verified. The label indicates an Actions script routine to be executed. The format to add the label is:

`ActionOnChange=labelname`

For example:

`<INPUT TYPE=TEXT NAME=Username ActionOnChange=Mytest(>`

Note that the variable for the **NAME** parameter is the field to be verified. In this example, the variable "Username" will contain the value typed in by the user. The current field's value is ***always*** passed to Caché.

2. Modify the the page's Actions script by adding the routine that will be executed by the label. In this example, the page's Actions script must have a routine named **Mytest**.

You can also pass the values of other form fields to Caché by adding the field names as a parameter list within the ActionOnChange parameter. For example:

```
<INPUT TYPE=TEXT NAME=Username ActionOnChange=Mytest(Info,Password)>
```

In this example, the values of the "Info" and "Password" field will be passed to Caché, in addition to the value of the "Username" field.

Note that the Mytest routine does not need to specify a matching formal parameter list. Just use a label of the form "Label()" and WebLink Developer will produce the appropriate results. For example, the routine in the Actions script will look like:

```
<SCRIPT Language=Cache>
//type=actions
Mytest(Info,Password) ; Test the username
...
...
Quit
</SCRIPT>
```

Setting Return Values with Field Validation

When setting up the field validation routine in the Actions script, you can return a string called **%return** which contains a list of Name/Value pairs. These pairs are of the form:

```
Name1=Value1&Name2=Value2&...etc
```

Note that the syntax (including the ampersand) is similar a URL. The order is not significant.

The names must be any combination of the following:

- A valid field name in the current page (e.g., Username, Password, Info).
- The reserved variable name [Error](#) (case sensitive), to define an error message for display to the user.
- The reserved variable name [Focus](#) (case insensitive), to define the field on which to move focus.

The values are used for the following purposes:

- For field names: the new value to appear on the Web page.
- For Error: the text of the error message to appear on the browser (within an alert box).
- For Focus: the name of the field on which to focus (e.g., Password).

The focus is also determined by these two situations:

- If Error is specified but Focus is not, focus returns to the original field.
- If %return is an empty string (i.e., null), focus moves to the next field.

Example1 of using %return:

```
%return="Error=Invalid username"
```

In this example, an alert box will appear containing the words "Invalid username" and focus will return to the Username field.

Example2 of using %return:

```
%return="Info=Try again&Error=Invalid username"
```

In this second example, the error alert box will be displayed, and the Info field will be updated to contain the text "Try again". Focus will return to the Username field.

Example3 of using %return:

```
%return="Info=Username OK&focus=Password"
```

In this third example, no error appears, the Info field changes to the text "Username OK", and focus switches to the Password field.

With the above rules, a sample Actions script could contain the following:

```
<SCRIPT Language=Cache>
//type=actions
Mytest(Username) ; Test the username
  If Username='ROB' SET %return="Error=Invalid Username&Info=Try Again"
  Quit
</SCRIPT>
```

In this sample Actions script, %return will be returned as null by default when Username="ROB".

Alternatively, the script could read as follows:

```
<SCRIPT Language=Cache>
//type=actions
Mytest(Username) ; Test the username
  If Username='ROB' S %return="Error=Invalid Username&Info=Try Again" Quit
  SET %return="Info=Username OK&Focus=Password"
  Quit
</SCRIPT>
```

In this case, when the user enters "ROB" as the username, the Info field is updated and focus moves to the Password field.

Of course, because the routine is run on the Caché server, it has access to the full resources of the Caché server, so it can make calls to existing routines; for example:

```
DO ^MyExistingUsernameTest(Username)
```

Handling Form Field Types

You may invoke the Event Broker interface from any form element by using a range of JavaScript event handlers. With them, you can modify the settings and/or contents of all form elements.

Event Handlers for Text and Password Elements

You may use one of two event handlers:

- **ActionOnChange:** During compilation, this is converted to the **OnChange** event handler parameter within the element. Use of the ActionOnChange parameter is described earlier in this chapter.
- **ActionOnBlur:** This is changed during compilation to the **OnBlur** event handler, which is triggered whenever focus is moved away from the form element, whether its value has changed or not. To use this Event Handler to trigger the Event Broker interface, use the special parameter:

```
ActionOnBlur=label({optional_parameter_list})
```

This is used identically to the ActionOnChange parameter.

Note that Netscape 2 browsers did not provide event handlers for the Password Element.

TextArea Element Event Handlers

Both the ActionOnChange and ActionOnBlur parameters can be used with TEXTAREA elements.

Event Handling for Radio Buttons and Checkboxes

Radio Buttons and Checkboxes only support the **OnClick** event handler, which is fired when a button or checkbox is selected using the mouse. WebLink Developer therefore provides the Event Broker interface for these elements via the special ActionOnClick parameter:

```
ActionOnClick=label({optional_parameter_list})
```

This is used identically to the ActionOnChange parameter.

Event Handling for Drop-down Lists

Select elements (using <Select> tags) support the OnChange, OnBlur, and OnClick event handlers, so you may use the ActionOnChange, ActionOnBlur, and ActionOnClick event handlers for Select elements. Note the difference each parameter provides in terms of triggering the Event Broker:

- ActionOnChange: triggers when you select a new or different item from the drop-down list
- ActionOnBlur: triggers when you move focus away from the drop-down list to a different element
- ActionOnClick: triggers when you click anywhere within the drop-down list

Dynamically Changing Drop-down Lists

You can modify the contents of a drop-down list and/or change the selected item from a Caché routine. A call to the Event Broker interface from any form element can be used to cause such a change.

In your Caché routine that is called via the ActionOnChange, ActionOnBlur, or ActionOnClick parameters, simply modify the LIST array for the element and/or the SELECTED array. This is the same as setting the drop-down list in your [pre-page script](#).

Then in the %return variable, add the name/value pair **SelectFieldName=New**. For example, if the Select tag was named "City", then you could do the following in the Caché routine called via the ActionOnChange parameter:

```
K LIST("City"),SELECTED("City") ; clear out current settings
S LIST("City",1)="London"
```

```

S LIST("City",2)="New York"
S LIST("City",3)="Los Angeles"
S LIST("City",4)="Paris"
S SELECTED("City","New York")=""
S %return="City=New&Focus=CustomerName"
Q

```

This would have the effect of replacing whatever was in the City drop-down list with the new list of four cities. New York would be selected automatically and focus would move to the CustomerName field.

The JavaScript generated by WebLink Developer works on both Netscape 4.x and Microsoft IE4 browsers, and can cope with you substituting a new list that is shorter, longer, or equal in length to the current list. The JavaScript needed to dynamically modify the browser's object model to cope with such changes is built for you automatically.

However, on Netscape 4 browsers, the width of the select-box window does not dynamically alter when you substitute the contents. You will therefore need to pre-determine the optimum width for the control.

On Microsoft IE4, the width is automatically adjusted to fit the widest option in the current list.

Dynamically Selecting Radio Button Options

You can select a Radio Button option via the %return variable when using the Event Broker interface. Simply set the field name to the value you wish to select.

For example, suppose you had this radio button pair for a Yes/No prompt:

```

Yes <input name=YesNo type=Radio value=1>
No  <input name=YesNo type=Radio value=0>

```

To automatically change the selection to "No", you would set in %return the name/value pair:

```
YesNo=0
```

The actual code would look like:

```
SET %return="City=New&YesNo=0&Focus=CustomerName"
```

Dynamically Selecting Checkbox Options

You can select one or more Checkbox options via the %return variable when using the Event Broker interface. Simply use the SELECTED array for the values you wish to select.

For example, suppose you had a checkbox set for a Color prompt:

```

Red <input name=Color type=Checkbox value="R">
Blue <input name=Color type=Checkbox value="B">
Green <input name=Color type=Checkbox value="G">
Orange <input name=Color type=Checkbox value="O">

```

To automatically change the selection to "Red" and "Green", you would use the SELECTED array as follows in your Caché routine:

```

KILL SELECTED("Color") ; flush out any existing selection
SET SELECTED("Color","R")=""
SET SELECTED("Color","G")=""

```

You then use the %return variable to invoke the change using the name/value pair:

```
FieldName=New
```

For example:

```
SET %return="City=New&YesNo=0&Color=New&Focus=CustomerName"
```

Any other options that were previously selected will be de-selected automatically.

Dynamically Changing TEXTAREA Contents

You can replace the contents of a TEXTAREA element using the TEXTAREA array (just as if you were setting it up in the pre-page script). To invoke the change within the %return variable, use the name/value pair:

```
TextFieldName=New
```

For example, if you had a TEXTAREA element called "MyText", your Caché routine might contain the following:

```
KILL TEXTAREA("MyText") ; flush out any existing text
SET TEXTAREA("MyText",1)="Line 1 of some new text"
SET TEXTAREA("MyText",2)="Line 2 of new text"
SET TEXTAREA("MyText",5)="This is line 5 - lines 3 and 4 are blank"
;
; now use the %return variable to invoke the change using FieldName=New
;
SET %return="City=New&YesNo=0&Color=New&MyText=New&Focus=CustomerName"
QUIT
```

Note: The TEXTAREA update mechanism cannot be used for very long sections of text.

Note on Reloading Pages

Remember that if you reload a page that you have modified via the Event Broker interface, the [pre-page script](#) will run. This may change back the contents of the page to their original settings.

As a result, you may need to test for a reload and bypass any initialization code. You can use the [PREVPAGE variable](#) for this test, as in this example:

```
If PREVPAGE'="MainMenu" Quit
...
... ;Pre-page script continues here, initializing values on the form
...
Quit
```

In this example, the pre-page script only initializes the page if it was loaded via the MainMenu page. Otherwise, the current settings in, for example, the LIST, SELECTED, and TEXTAREA arrays will be used to populate the form.

Internet Client Functions

- [Sending E-mail Via SMTP](#)
- [Receiving E-mail From POP3 Servers](#)
- [Using the FTP Facilities](#)
- [Retrieving Web Pages From Other Sites](#)

Caché WebLink Developer provides a set of functions that allow the Caché server to act as a range of Internet clients. Using these functions, the back-end Caché system can:

- send e-mails to an SMTP server
- receive e-mails from a POP3 server (in addition to other POP3 client functions)
- send and receive files to/from an FTP server (in addition to other FTP client functions)
- request and receive Web pages from a Web server

These functions can be used to enable simple messaging between otherwise disconnected Caché servers. For example, the HTTP function can be used to fetch and repackage Web content from another site or to monitor the availability and performance of a Web site. The e-mail send function may be used to send alerts to an email-enabled paging system to report failure or unavailability of a Web site.

Sending E-mail Via SMTP

To send an e-mail from Caché, use the following call in your script:

```
SET Status=$$SEND^%wldsmtp(smtpdomain,from,to,subject,.message,timeout,GTMOffset)
```

Parameters are as follows (all are mandatory except for *timeout* and *GTMOffset*).

Parameter	Meaning
smtpdomain	Domain name or IP address of SMTP server (e.g., mail.myisp.com or 192.12.34.56) Note: Your Caché system must be capable of making a direct TCP connection to the specified server.
from	E-mail address of the sender (e.g., cache@intersys.com)
to	E-mail address of recipient (e.g., yourserver@intersys.com) Note: These addresses may be personal addresses or addresses set up specifically for automatic e-mail delivery/receipt.
subject	Subject of the email - any text can be used. Note: For automated messaging applications, the subject field can be a useful identifier indicating the purpose/role of the received message, allowing the correct processing to be invoked.
.message	Name of a local array containing the contents/body of your e-mail message. This array should have a single subscript that denotes the line number; for example: <pre>message(1)="Line one of the message" message(2)="This is line 2" message(3)="Last line of the message"</pre>
timeout	Optional parameter. Maximum time the Caché server should wait to make a connection to the SMTP server. If omitted, a default of 20 seconds is used. The function aborts automatically if the timeout is exceeded.

GMTOffset	<p>Optional parameter. E-mails are date/timestamped using your Caché server's clock. If this parameter is omitted, the time stamp is shown as being based on GMT; for example:</p> <pre>25 May 1999 13:10:30 GMT</pre> <p>You can indicate your time zone with this parameter:</p> <pre>if GMTOffset = "+0500"</pre> <p>the date/time stamp will be 25 May 1999 13:10:30 +0500</p>
-----------	--

If the e-mail is successfully sent, the function returns a Status value of "1". A Status value of "0" indicates that a failure occurred. The reason for the failure is returned in the **FailureReason** variable.

Batch Sending of E-mails

You may choose to send e-mails immediately from your application by using the call above, or you can save them to a global for batch processing by a background e-mail sending daemon routine that is included in WebLink Developer.

To save an e-mail to the batch queue, use the following function call:

```
DO QMAIL^%wldemd(application,from,to,subject,.message,senddate)
```

Parameters are as follows (all are mandatory except for *senddate*).

Parameter	Meaning
application	Name of the WebLink Developer application within which the e-mail has been created. Normally, you can use the WebLink Developer internal variable %App .
from	E-mail address of the sender (e.g., cache@intersys.com)
to	E-mail address of recipient (e.g., yourserver@intersys.com) Note: These addresses may be personal addresses or addresses set up specifically for automatic e-mail delivery/receipt.
subject	Subject of the email - any text can be used. Note: For automated messaging applications, the subject field can be a useful identifier indicating the purpose/role of the received message, allowing the correct processing to be invoked.
.message	Name of a local array containing the contents/body of your e-mail message. This array should have a single subscript that denotes the line number; for example: <pre>message(1)="Line one of the message" message(2)="This is line 2" message(3)="Last line of the message"</pre>
senddate	Optional parameter. Specifies the date on which the e-mail is to be sent. The value should be in \$HOROLOG format. If this parameter is not specified, the current date (i.e., \$HOROLOG) is assumed.

To start and stop the batch e-mail daemon routine, use WebLink Developer's configuration page.

You can also start the daemon manually with the following command, which you can include in your system startup file:

JOB ^%wldemd(*timeout*,*GMTOffset*)

Both parameters are optional. Their meanings are as follows:

Parameter	Meaning
timeout	Optional parameter. Maximum time the Caché server should wait to make a connection to the SMTP server. If omitted, a default of 20 seconds is used. The function aborts automatically if the timeout is exceeded.
GMTOffset	<p>Optional parameter. E-mails are date/timestamped using your Caché server's clock. If this parameter is omitted, the time stamp is shown as being based on GMT; for example:</p> <p style="text-align: center;">25 May 1999 13:10:30 GMT</p> <p>You can indicate your time zone with this parameter:</p> <p style="text-align: center;">if GMTOffset = "+0500"</p> <p>the date/time stamp will be 25 May 1999 13:10:30 +0500</p>

Note that the batch e-mail daemon must first be configured using WebLink Developer's configuration page.

Receiving E-mail From POP3 Servers

E-mail can be picked up from a POP3 mailbox. The receive process is more complex than the transmission mechanism described above, because:

- you may be using the POP3 mailbox for a variety of purposes, and you may only want to select specific e-mails from your mailbox
- you may or may not want to delete the messages from your mailbox

WebLink Developer therefore includes a set of function calls that provide you with the most common and useful POP3 functions. The overall process is:

1. [Log in](#) to your POP3 mailbox.
2. [Find out](#) how many e-mails are waiting for you.
3. Optionally, [retrieve](#) the headers for these messages.
4. Select, [download](#), and process your waiting e-mails.
5. [Flag for deletion](#) any downloaded e-mails.
6. [Log out](#) from your POP3 mailbox.

Invoke these steps from your Caché system using the function calls described in the following seven sections.

Logging In to Your POP3 Mailbox

SET Status=\$\$LOGIN^%wldpop3(*pop3domain*,*username*,*password*,*timeout*)

Parameters are as follows (all are mandatory except for *timeout*).

Parameter	Meaning

pop3domain	Domain name or IP address of the POP3 server (e.g., pop3.myisp.com or 192.12.34.56) Note: Your Caché system must be capable of making a direct TCP connection to the specified server.
username	Your POP3 mailbox username (your ISP may allocate this for you)
password	Your POP3 mailbox password.
timeout	Optional parameter. Specifies the maximum time the Caché server should wait to make a connection to the POP3 server. If omitted, a default of 20 seconds is used. The function aborts automatically if the timeout is exceeded.

Successful login is indicated by a returned value of **1** in Status. If Status is **0**, a login error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the POP3 server is in the **%pop3dialog** array. This array is cleared automatically when the LOGIN function is invoked. The other POP3 functions append to the dialog array.

Returning the Number of Waiting E-mails

```
SET NumberOfEmails=$$HOWMANY^%wldpop3()
```

No input parameters are required for this function.

The function returns the number of e-mails that are waiting for you in your POP3 mailbox. The total size of the "maildrop" (in bytes) is in the **%pop3mdsize** variable.

A record of the dialogue that took place between the Caché system and the POP3 server is in the **%pop3dialog** array.

If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable.

Retrieving E-mail Headers

You can retrieve message headers only for the e-mails that are waiting for you, and optionally, download a fixed number of lines of the e-mail contents.

E-mails in your POP3 mailbox are selected by their **message number**. This is a simple integer with a range from 1 to the number of waiting emails. For example, if the HOWMANY() function returns a value of 5, your messages can be selected one by one using message numbers from 1 through 5.

You pick up the headers for a specified e-mail using the function:

```
SET NumberOfLinesRetrieved=$$TOP^%wldpop3(MessageNumber,NumberOfBodyLines)
```

Parameters are as follows:

Parameter	Meaning
MessageNumber	The specified message number.
NumberOfBodyLines	Optional parameter. Specifies the number of additional lines to retrieve from the message body (if that many lines are available). If <i>NumberOfBodyLines</i> is not specified, it defaults to 0 and just the headers are retrieved.

The total number of lines retrieved is returned by the function. If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable.

In addition, the following variables/arrays are returned.

%pop3message	This array contains the header (and optionally part of the body) of the e-mail. It has a single parameter, which is the line number. For example: %pop3message(1)="This is line 1 of the retrieved message" %pop3message(2)="This is line 2" %pop3message(3)="This is the last line"
%pop3Subject	The Subject of the e-mail
%pop3From	The e-mail address of the sender of the e-mail
%pop3To	The e-mail address of the recipient of the e-mail
%pop3Cc	The e-mail address(es) of any others to whom the e-mail was carbon-copied (Cc).
%pop3Date	The date/time stamp of the e-mail

Download the E-mail

You pick up a specific e-mail using the function:

```
SET NumberOfLinesInMessage=$$GETMESS^%wldpop3(MessageNumber)
```

where *MessageNumber* is the specific [message number](#). The number of lines retrieved is returned by the function. If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable.

In addition, the following variables/arrays are returned.

%pop3message	This array contains the body of the e-mail. It has a single parameter, which is the line number. For example: %pop3message(1)="This is line 1 of the retrieved message" %pop3message(2)="This is line 2" %pop3message(3)="This is the last line"
%pop3Subject	The Subject of the e-mail
%pop3From	The e-mail address of the sender of the e-mail
%pop3To	The e-mail address of the recipient of the e-mail
%pop3Cc	The e-mail address(es) of any others to whom the e-mail was carbon-copied (Cc).
%pop3Date	The date/time stamp of the e-mail

Flagging Downloaded E-mail for Deletion

If you wish to delete any messages from the POP3 server, use the function:

```
SET Status=$$DELMESS^%wldpop3(MessageNumber)
```

where *MessageNumber* is the specific [message number](#).

If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable. Successful deletion results in a Status of **1**.

Flagging Messages for Un-deletion

Messages are not actually deleted until you log out from the POP3 server. If you wrongly mark a message for deletion, the flag can be reset using the function:

```
SET Status=$$UNDELMESS^%wldpop3()
```

All messages flagged for deletion will be un-flagged.

This function has no input parameters. If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable. Successful completion of the function results in a Status of **1**.

Logging Out From Your POP3 Mailbox

You log out from your POP3 mailbox using the function:

```
SET Status=$$LOGOUT^%wldpop3()
```

This function has no input parameters. If the function fails, a value of **0** is returned. The reason for the failure is indicated in the **FailureReason** variable. Successful completion of the function results in a Status of **1**.

On successful logout, the connection to the POP3 server is terminated and any flagged messages are deleted from your mailbox.

Using the FTP Facilities

The FTP facilities allow you to carry out the following operations:

- Upload a file to an FTP server
- Upload the contents of a local array to an FTP server
- Download a file from an FTP server and save it as a file
- Download the contents of a file from an FTP server into a local array

Rather than pre-package the functionality, WebLink Developer provides you with a set of extrinsic functions that provide the main FTP protocol functions. You can therefore tailor the use of these facilities to meet your specific application requirements.

In general, the procedure for accessing an FTP server is:

1. [Log in](#) to the FTP server.
2. [Navigate](#) to the required directory on the FTP server.
3. [Check](#) the directory listing/contents on the FTP server.
4. [Download](#) or [upload](#) files.
5. [Log out](#) from the FTP server.

The functions provided within WebLink Developer to allow you to perform these (and other) tasks are described in the following sections.

Logging In to FTP Servers

To log in to an FTP server, use the following function call:

```
SET Status=$$LOGIN^%wldftp(ftpdomain,username,password,timeout)
```

Parameters are as follows (all are mandatory except for *timeout*).

Parameter	Meaning
ftpdomain	Domain name or IP address of the FTP server (e.g., ftp.myisp.com or 192.12.34.56). Note: Your Caché system must be capable of making a direct TCP connection to the specified server.
username	The username that allows you access to the FTP server. Notes: Anonymous ftp servers normally allow the username "anonymous". Your ftp username also normally dictates the level and range of access permissions.
password	The password for the username. Note: Anonymous ftp servers normally require the password "guest" or your e-mail address.
timeout	Optional parameter. Specifies the maximum time the Caché server should wait to make a connection to the FTP server. If omitted, a default of 20 seconds is used. The function aborts automatically if the timeout is exceeded. The specified timeout is used by all subsequent FTP functions.

Successful login is indicated by a returned value of **1** in Status. If Status is **0**, a login error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server is in the **%pop3dialog** array. This array is cleared automatically when the LOGIN function is invoked. The other FTP functions append to the dialog array.

Changing Directories on FTP Servers

You can change to a specific directory on the FTP server by using the function call:

```
SET Status=$$CHANGEDIR^%wldftp(DirectoryPath)
```

where *DirectoryPath* is the full or relative pathname of the directory to which you wish to point on the ftp server. Full pathnames are relative to the FTP server's logical root directory and must start with a "/" character. If a "/" is not the first character, a path relative to the current directory is assumed by the FTP server. Note that FTP servers normally adopt the UNIX syntax for directory paths.

Successful directory change is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Listing Directories on FTP Servers

You can obtain a listing of the files in the current directory on the FTP server by using the function call:

```
SET Status=$$DIRLIST^%wldftp()
```

This function has no input parameters.

The directory listing is returned in the **%ftpdirlist** local array. This array has a single subscript representing the line number; for example:

```
%ftpdirlist(1)="----- 1 owner group 72801 Apr 20 10:25 casp.rou"
%ftpdirlist(2)="----- 1 owner group 8491 Mar 3 11:14 Myfile.txt"
```

Successful directory listing is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Downloading Files From FTP Servers

You can download a file from the current directory of the FTP server and save it as a local file on (or in a directory addressable from) the Caché server. Use the function call:

```
SET Status=$$DOWNLOAD^%wldftp(filename,savepath)
```

Parameters are as follows.

Parameter	Meaning
filename	Name of the file to be downloaded (e.g., "casp.rou").
savepath	Optional parameter. Specifies the directory path on the Caché server's file system into which the downloaded file will be copied. If not specified, the Caché system's current working path will be assumed.

The contents of the file will be copied into a file of the same name in the specified (or default) path on the Caché server. The function will transfer both text and binary files.

Successful file transfer is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Downloading Files Into a Local Array

It is sometimes preferable to download the contents of a file into a local array on the Caché server (rather than create a copy of the file on the Caché server). To do this, use the function call:

```
SET Status=$$ADOWNLOAD^%wldftp(Filename,.LocalArray,Mode)
```

Parameters are as follows (all are mandatory except *Mode*).

Parameter	Meaning
Filename	Name of the file to be downloaded (e.g., "casp.rou").
.LocalArray	Name of the array to create on the Caché server and into which the contents of the specified file will be copied. Note the use of the initial dot (".") character in this parameter
Mode	Optional parameter. Sets the copy mode. Valid values are: R — record mode (default). Each line in the source file is copied into its own equivalent line in the target array. Note that if an invalid <i>Mode</i> value is specified, record mode will be used instead. S — stream (or binary) mode. Use this mode for binary files where no additional CRLF's are to be added to the data stream.

Successful file transfer is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Uploading Files To FTP Servers

To upload a file from the Caché server's file system to the FTP server, use the following function call:

```
SET Status=$$UPLOAD^%wldftp(Filename,LocalPath,ToFilename,ToPath)
```

Parameters are as follows (only *Filename* is mandatory).

Parameter	Meaning
Filename	Name of the file to be uploaded (e.g., "casp.rou"). The specified file must be capable of being found and opened by the Caché server.
LocalPath	Directory path on the Caché server in which the file to be uploaded resides. If omitted, the current working directory is assumed.
ToFilename	Name of the file to be created on the FTP server. If omitted, the filename of the source file will be used.
ToPath	Directory path on the FTP server in which the uploaded file will be created. If omitted, the current working directory on the FTP server will be used.

Successful file transfer is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Uploading From a Local Array to the FTP Server

You can also upload the contents of a local array, rather than a file on the Caché server's file system.

To upload a local array from the Caché server to the FTP server, use the following function call:

```
SET Status=$$AUPLOAD^%wldftp(.LocalArray,ToFilename,ToPath,Mode)
```

Parameters are as follows (all are mandatory parameters except for *Mode*).

Parameter	Meaning
<i>.LocalArray</i>	<p>Name of the local array whose contents is to be uploaded. Note the initial dot (".") at the start of the parameter.</p> <p>The structure of the array depends on the type of file (and therefore the Mode). If a record structured file is to be created, each line in the LocalArray should represent a line in the target file; for example:</p> <pre>LocalArray(1)="This is line 1 of the file" LocalArray(2)="This is line 2" ...etc</pre> <p>If a binary file is to be created, Stream mode should be selected and the function will not insert any additional CRLF's into the target file it creates on the FTP server.</p>
ToFilename	Name of the file to be created on the FTP server.

ToPath	Directory path on the FTP server in which the uploaded file will be created. If omitted, the current working directory on the FTP server will be used.
Mode	<p>Optional parameter. Sets the copy mode. Valid values are:</p> <p>R — record mode (default). Each line in the source array is copied into its own equivalent line in the target file. Note that if an invalid <i>Mode</i> value is specified, record mode will be used instead.</p> <p>S — stream (or binary) mode. Use this mode for binary files where no additional CRLF's are to be added to the data stream.</p>

Successful file transfer is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Logging Off From FTP Servers

You log off from the FTP server by using the function call:

```
SET Status=$$LOGOUT^%wldftp()
```

The function has no input parameters.

The connection to the FTP server will be terminated and the TCP device closed on the Caché server.

A successful logoff is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Other FTP functions

A number of other FTP functions can be executed while you are logged into the FTP server. These functions simply call the equivalent FTP command. For more information on FTP commands, see the relevant RFC for the FTP protocol. The WebLink Developer FTP functions are provided to allow you to build your own tailored FTP client access.

Note that a successful action is indicated by a returned value of **1** in Status. If Status is **0**, an error has occurred. The reason is indicated in the **FailureReason** variable.

A record of the dialogue that took place between the Caché system and the FTP server can be found in the **%ftpdialog** array.

Deleting Files From FTP Servers

To delete a file, use the following extrinsic function call:

```
SET status=$$DELETE^%wldftp(FileName)
```

The file specified by *FileName* will be deleted from the currently logged directory path on the FTP server, provided the logged-in username has the privileges to do so.

Note that this command takes immediate effect if the file exists, and the file will be deleted without any means of recovery. You must therefore take great care when using this function.

TYPE Command

You can set the type of file using the function:

```
SET Status=$$TYPE^%wldftp(FileType)
```

Valid values of *FileType* are **A** (ASCII), **E** (EBCDIC), **I** (Image), and **L** (Local).

Note that the upload and download functions provided within ^%wldftp set the file type to A.

MODE Command

You can set the mode of transmission to/from the FTP server using the function:

```
SET Status=$$MODE^%wldftp(ModeType)
```

Valid values of *ModeType* are **S** (Stream Mode), **B** (Block Mode), and **C** (Compressed Mode).

Note that the upload and download functions provided within ^%wldftp set the mode to S.

PWD Command

You can get the current working directory path by calling:

```
SET DirectoryPath=$$PWD^%wldftp()
```

This function has no input parameters.

The current working path on the FTP server is returned by this function if it successfully executes.

SYST Command

You can determine the operating system of the FTP server by calling:

```
SET OS=$$SYST^%wldftp()
```

This function has no input parameters.

The operating system of the FTP server is returned by this function if it successfully executes.

Renaming Files on FTP Servers

You can rename a file on the FTP server by calling:

```
SET Status=$$RENAME^%wldftp(FromFilename, ToFilename)
```

Parameters are as follows (both parameters are mandatory).

Parameter	Meaning
FromFilename	Name of the file on the FTP server to be renamed.
ToFilename	New name for the file on the FTP server.

This function assumes and works within the current working directory path on the FTP server. To change directory, use the [\\$CHANGEDIR](#) function.

Retrieving Web Pages From Other Sites

The Caché server can request and receive Web pages from other Web sites. The Caché server must be connected by a TCP/IP connection (e.g., the Internet) to the other site.

To retrieve a web page, use the extrinsic function call:

```
SET NumberOfLinesRetrieved=$$GET^%wldhttp(URL,timeout)
```

Parameters are as follows (*URL* is a mandatory parameter).

Parameter	Meaning
URL	<p>The URL to fetch (e.g., http://www.intersys.com/index.html. The http:// at the beginning of the URL is optional.</p> <p>You can specify a particular port by adding the port number using the normal URL convention. For example, port 8080 would be:</p> <pre>http://123.45.67.89:8080/MyPages/default.htm</pre>
timeout	Optional parameter. Specifies the maximum time the Caché server should wait to make a connection to the HTTP server. If omitted, a default of 20 seconds is used. The function aborts automatically if the timeout is exceeded.

If successful, the function returns the number of lines retrieved. The HTML retrieved is in the array **%html**. This array has a single subscript, which specifies the line number. For example:

```
%html(1)="HTTP/1.0 200 OK"
%html(2)="Server: Microsoft-IIS/3.0"
%html(3)="Date: Mon, 11 Jan 1999 17:20:37 GMT"
%html(4)="Content-Type: text/html"
%html(5)="Accept-Ranges: bytes"
%html(6)="Last-Modified: Mon, 26 Oct 1998 12:58:25 GMT"
%html(7)="Content-Length: 1011"
%html(8)=" "
%html(9)="<HTML>"
%html(10)=" "
%html(11)="<HEAD>"
...etc
```

Notice that the HTTP header will always be retrieved before the HTML. For this reason, the function can be used as an HTTP header debugging tool.

If the function fails, the reason will be reported in the **FailureReason** variable.

Editing Notes

- [WebLink Developer Text Editor](#)
 - [DisplayIf Functionality](#)
 - [Use of %KEY Array](#)
 - [Variables Within an Onload Event Handler](#)
 - [Automated Backtracking](#)
 - [Developing ObjectScript Modules](#)
 - [Setting Timeout Periods](#)
 - [Principal Device Protection](#)
 - [Configuring Internet Explorer](#)
-

This section contains miscellaneous editing notes and reminders.

WebLink Developer Text Editor

With WebLink Developer, you do not need a separate page development tool or text editor to build and maintain Web applications.

A WebLink Developer option lets you maintain an application's HTML (.asp) source pages from within the WebLink Developer browser session. You can view and edit existing pages, add new ones, delete pages, and compile individual pages and/or all pages in an application. You can also run the application within its own browser, leaving WebLink Developer running in the original browser.

Note that because the WebLink Developer editor provides basic text editing, it is not intended to be a complete replacement for a graphical Web-page development tool. However, you may find it useful for quick changes or as a replacement for a text editor.

DisplayIf Functionality

An [earlier chapter](#) describes how to use the **DisplayIf** attribute for form Submit buttons. You can also use this attribute:

- Within other HTML tags, including hyperlinks.
- As a WebLink Developer-specific tag.

These usages are described in the following two sections.

DisplayIf With Other Tags

The **DisplayIf** attribute can be used with other tags, including hyperlinks. An example of using **DisplayIf** with a hyperlink is:

```
<a href=second.asp displayif=$$disp()>Second page</a>
```

where **\$\$disp()** is the label of a function in your Action script block.

The one limitation is that when you use **DisplayIf** with an HTML tag that does not have a matching `</>` end tag, the WebLink Developer compiler cannot determine what else should be made conditional. Therefore, the condition is restricted only to the tag itself.

To illustrate this limitation, assume that you have this text box:

```
Username: <INPUT TYPE=Text Name=Username Displayif=$$disp()>
```

The compiler cannot assume that you also want to make the prompt conditional. Therefore, if **\$\$disp()** returns 0 or false, the input

box will not be displayed, but the "Username:" prompt will be. To solve this problem, use **DisplayIf** as a WebLink Developer-specific tag.

DisplayIf As a Specific Tag

You can use **DisplayIf** as a WebLink Developer-specific tag, with a matching terminating tag. The syntax is:

```
<DISPLAYIF CONDITION={ condition }>  
HTML and/or text  
</DISPLAYIF>
```

At run-time, all HTML and text between these tags will be conditional on the outcome of the specified *condition*. The tag name and the **CONDITION=** attribute name are case insensitive.

An example of this usage is:

```
<DisplayIf Condition=$$disp()>  
Username: <INPUT TYPE=Text Name=Username>  
Password: <INPUT TYPE=Password Name=Password>  
</DisplayIf>
```

If **\$\$disp()** returns false, the username and password prompts and form fields will not be displayed.

Note that you cannot nest <DisplayIf> tags. For this type of functionality, you must use an [in-page script](#).

Use of %KEY Array

Caché WebLink Developer automatically copies all the new values in the %KEY array (i.e., name/value pairs sent from the previous form or hyperlink) to local variables of the same name.

For example, if a form containing **UserName** and **Password** is submitted, the pre-page processing script in the subsequent page will have available to it variables called **UserName** and **Password** containing the values submitted by the user. %KEY still also exists (note that the %KEY name is case sensitive).

Note that %KEY is flushed automatically by WebLink between each page. However, the variables that Developer creates from %KEY are persistent between pages.

Variables Within an Onload Event Handler

WebLink Developer supports the use of variables within a call to the **OnLoad** event handler in the tag, as in this example:

```
<BODY OnLoad="HightlightRow(|CurrentRow|)">
```

or where multiple functions are to be invoked:

```
<BODY OnLoad="HightlightRow(|CurrentRow|) ; InitialisePage()">
```

Automated Backtracking

Users often want to backtrack through the pages they have navigated through. WebLink Developer automates and drastically simplifies backtracking, by allowing the page developer to use the pseudo-page name **Back.asp** (which is case insensitive) in hyperlinks and submit buttons.

Therefore, in all but your first page, you can provide a return to the previous page by using hyperlinks or submit forms such as:

```
<a href="back.asp">Go back to previous page</a>
```

and

```
<input type=submit name=GoBack value="Go Back" nextpage="Back">
```

When such a hyperlink or submit button is clicked, the user is taken to the previous page (i.e., the one that called the page they are currently viewing). The WebLink software does this by navigating up one level in the page stack and decrementing the current page stack level. The simple part is that you do not need to know at design time the actual page name to which the application will go.

If you use a "jump" page for run-time decisions on which page to jump to next, you must include a check for backtracking to ensure it is bypassed. In most cases, insert the following as the first line in the Jump page's pre-page script:

```
If BACKTRACK Set JUMP="BACK" Quit
```

WebLink Developer will always create the local run-time variable **BACKTRACK**, which is set to **true** (i.e., it has a value of 1) if the user has backtracked using the **Back.asp** pseudo-page. **BACKTRACK** always exists.

The **Back.asp** pseudo page makes use of Developer's internal page stack. You can see how it works by examining the local array **%pgstack**. The variable **%pageno** points to the current stack level in the page stack array.

There is one important consideration when using **Back.asp**: the pre-page script of the page you are back-tracking to will always be run, so it must be able to cater for such re-entry. Once again, testing the state of the **BACKTRACK** variable may be useful in such circumstances, since you may wish to the pre-page script to execute only when navigating forwards.

Developing ObjectScript Modules

Because the script editing facilities within FrontPage and other tools do not include built-in Caché ObjectScript syntax checking, you should be aware that this area will probably cause difficulties during application development.

If you are running your application in WebLink's State-Aware mode, you will find that some run-time syntax errors in your scripts can cause problems that require you to close down the WebLink connections. Sometimes you may have to manually shut down Caché processes using **^RESJOB** or equivalent utilities.

InterSystems recommends that you develop your Caché or Open M scripts separately in an environment that allows you to check your syntax. You should run each script as a standalone module to ensure that it will run properly. You can then cut and paste the script into the Web page.

InterSystems also recommends that you develop Caché WebLink Developer applications using the default [Stateless Mode](#). After you have fixed all scripting errors, you can change the application to the State-Aware Mode if you wish.

If a Caché WebLink Developer application crashes, do one of the following:

- In Stateless Mode, the session simply quits. Caché WebLink Developer will automatically trap and save the error, and also save status of variables. Use Caché WebLink Developer's built-in Error Trap examination facility, which you can access from the Caché WebLink Developer's configuration/ maintenance menu.
- In State-Aware Mode, use the WebLink System Management functions to shut down the session. Select **Close Down M Sessions** and click the **Close Sessions** button.

You should also use **^%SS** to see whether all the **^%MGW** jobs have stopped (other than the Process Control Demon, which will be running at the bottom IP Port you have specified). If one or more other **^%MGW** processes are running, you may have to use the **RESJOB** to terminate them. This procedure is required only when building the application. After the application is debugged, it will perform a graceful shutdown.

Setting Timeout Periods

The applications you create with Caché WebLink Developer will automatically halt if the browser does not send any response within a set time period, as determined by the [Client_Timeout parameter](#). It is recommend that you set the timeout period to at least 30 minutes.

If [client pull](#) is also being used on each page with a lower timeout period (such as 5 to 10 minutes), then under most circumstances client pull will properly shut down the user's session. However, it is possible for the user to break off communication (e.g., close the browser) so the Client_Timeout parameter will then come into force. Caché WebLink Developer does this all automatically for you, provided you set the Client_Timeout parameter. Note, however, that client pull-based shut-down is not automatically implemented and must be manually implemented in each appropriate page.

Principal Device Protection

You should not be making changes to the principal device. WebLink Developer will protect against accidental changes that your scripts may make when using files and other devices. Provided that you do not change the **%io** variable, WebLink Developer saves and restores **\$I** before and after your scripts and pages.

Configuring Internet Explorer

If you are using Internet Explorer, you may need to reconfigure its default caching algorithm. For IE 4.0:

1. Select **Internet Options** from the View menu.
2. In the **Temporary Internet files** section of the General tab, click the **Settings** button.
3. On the Settings screen, select **Every visit to the page** and click **OK**.

The Netscape Communicator browser caches properly by default, but you should check its caching parameters to be sure.



Installation Instructions

Caché WebLink Developer is automatically installed when you install the core WebLink product. For example, on a Caché 3.1 for Windows installation, both the core WebLink and the WebLink Developer are installed.

This appendix provides instructions for installing Caché WebLink Developer from the downloadable installation kit. Instructions for installing Beta Releases of Developer can be found further down the page.

Follow these steps to install the production version kit:

1. The installation kit is contained in a file named **wld41.zip**. Use WinZip to uncompress the file **wld41.rou**. The **wld41.rou** file must be extracted and copied to the directory associated with your Caché %SYS namespace (normally the \Cachesys\mgr directory).
2. Use the Caché routine restore utility (or ^%RI in programmer mode) to copy the routines from **wld41.rou** into your %SYS namespace. The routines all start with the characters **%wld**. Ignore any compilation errors that are reported for the routines **%wlddsm** if present - these will not be used on your system.
3. If you are installing Developer on an MSM or DSM system, use the ^%RR utility and the appropriate source file - wld41.msm or wld41.dsm respectively.
4. Provided you are using a recent copy of WebLink, the routine ^%MGW3 will already be configured to process Developer requests via WebLink. **If you are using a recent version of WebLink, you can skip to step 5 now.**

Older versions of WebLink are not pre-configured and must be set up as follows - you will need to do this if label HTML+1 in ^%MGW3 does not contain the line:

I \$D(%KEY("wlapp")) D ^%wld Q

If this is the first time you have installed WebLink Developer v4.1, you must use one of the following two methods to modify the WebLink daemon routine ^%MGW2 (or ^%MGW3 if you have previously been using this instead as the calling point for your WebLink applications):

- o **Method 1:** If this is the first time you have used WebLink or you have not made any significant changes to ^%MGW2 or ^%MGW3, go to Caché Programmer mode and, from the %SYS namespace, run the ^%wldcnf routine. For example:

```
%SYS> D ^%wldcnf
```

- o **Method 2:** If you have already made significant modifications to ^%MGW2 or ^%MGW3, manually edit in the following lines:

Insert the line:

IF \$G(%KEY("wlapp"))'="" DO ^%wld QUIT

after the label **A0** or, if your version of ^%MGW2 (or ^%MGW3) contains it, the label **HTML**.

Next, if your version of ^%MGW2 includes a label **JAVA**, insert the following line after it:

IF %REQUEST["wlapp." DO ^%wldja QUIT

5. If you have not already done so, it is now a good idea to patch the Caché WebLink Developer maintenance suite into one of your Web pages - perhaps into your home page. Incorporate the following lines of HTML into the page(s) of your choice:

```
<href="/scripts/mgwms32.dll?MGWLPN=yourLPN&wlapp=wlddev">  
Caché WebLink Developer Maintenance Suite</a><br>
```

where **"yourLPN"** should be substituted with your WebLink configuration Login Profile Name (see the main WebLink documentation). The Login Profile Name is the name that you assigned to the Caché server when you configured Caché WebLink.

6. The name/value pair *wlapp=wldev* denotes that a WebLink Developer application is to be run (**wlapp**) and that the application to be run is WebLink Developer itself (**wldev**). The WebLink Developer maintenance suite has, as this suggests, been developed using WebLink Developer and is a Caché-based Web application.
7. Ensure the WebLink networking component is properly installed and that communication is successfully taking place between your web server and Caché server. Start up a browser (on any workstation that can communicate with the Web server) and click the hyperlink you have just created in the previous step. The WebLink Developer application will start.

The Caché WebLink Developer maintenance suite application automatically detects whenever a new installation or upgrade is taking place. The first page you see will therefore be an advisory page indicating that it will now install and configure itself. Note that this process is entirely browser-based.

If you have used Version 3.4 (or earlier) of WebLink Developer, your old configuration will be converted to the new Version 4.1 format and structure. Applications that had been already compiled with the earlier versions of WebLink Developer are automatically recompiled into the new Version 4.1 format. All globals and routines related to earlier versions of WebLink Developer are automatically deleted from your Caché system. WebLink Developer will guide you through this process.

You should pay special attention to the form that asks you to confirm/define the namespaces (or UCIs) on your Caché (or MSM/DSM) system. It does this to ensure that it can find all your previously compiled applications in all namespaces (or UCIs) and automatically convert them. However, if you miss out or forget to add a namespace at this stage, do not worry - you can re-configure your WebLink Developer system and recompile applications later.

An important field to check, however, is the one that defines your System namespace (normally %SYS on a Caché system). This is used for a variety of functions throughout WebLink Developer.

You can give your Caché system a name - WebLink Developer will subsequently use this name to identify the Caché server with which it is communicating. This is particularly useful if you have more than one Caché server.

After the upgrade/install has completed, you will be sent a page with a button marked "Finish". Click this button to return to your Home Page. When you next click on the Hyperlink for the WebLink Developer maintenance suite, you will see the main Developer menu. Caché WebLink Developer is now ready for use.

When first installed, the Caché WebLink Developer maintenance suite does not perform any user authentication - anyone can run it. The first thing you should do is to define usernames and passwords for users authorized to run the Caché WebLink Developer maintenance suite. Do the following:

1. Start the Caché WebLink Developer Maintenance Suite.
2. Select the second menu option - "Change to a different Namespace/UCI". Select the %SYS namespace (or appropriate manager UCI).
3. Select the fifth main menu option - "Configure an application". You will be given a menu of the Caché WebLink Developer applications you have compiled in the %SYS namespace - you will normally only see one - the WLDEV application. This is the WebLink Developer Maintenance Suite. Select the WLDEV application.
4. Select the "Maintain User Details" hyperlink.
5. Select the "add a user" menu option - follow the instructions. Note that by default, the username is activated for 30 days. You may extend or reduce this period at any time by using the "Edit an existing user's details" option. You may add, edit, or delete users with this Web-based utility.
6. Quit from WebLink Developer and return to your home page. Select the hyperlink to run the Caché WebLink Developer maintenance suite again. You will now be asked to enter your username and password - from now on, only users authorised using the Caché WebLink Developer configuration application can use this application.

Note: to revert to having no user authentication, simply delete all users registered to use the WLDEV application - you must be in the %SYS namespace to do this.

Note that if somebody tries to hack into a Caché WebLink Developer application where the built-in user authentication has been used, repeated attempts at logging in to a particular username will result in that username being suspended - the expiration date is set to a date prior to the current date. As system manager, you may set the conditions under which users are suspended, and you may reinstate such usernames - in both cases use the "Configure an application" and "Maintain User Details" options in the Caché WebLink Developer configuration application.

Upgrading to a new (or Beta) version of Developer

The instructions above refer to the production version. If you are installing or upgrading to a Beta release, the filenames will differ: whereas the production version files are all named wld41.xxx (where xxx is a file extension), beta release file names will be of the form wldvvbb.xxx where vv is the version number and bb is the beta release number, eg wld4203.rou. Otherwise the installation process will be the same. The Beta Release zip file will contain any relevant Release Notes in a .txt file.

To install a new version of Developer, simply copy in the new versions of the %wld* routines, point your browser at the Developer application (wlapp=wldev) and it will re-configure itself to the new version.

MSM for NT Compatibility

WebLink Developer can run on the MSM for Windows NT platform. The ZIP file includes a file with a .msm extension (for example, **wld4142.msm**). Use this file if you use ^%RR to restore the WebLink Developer routines into your MGR UCI on MSM. Note there is no difference in the routines in the two save files - they have simply been saved differently.

Provided that WebLink is installed and running correctly on your MSM/NT system, WebLink Developer should run and start its configuration/installation process.

The configuration issues for MSM for NT are as follows:

- The default partition size should be 100K or greater.
- The default protection of % globals should be set to RWD for all UCIs, because WebLink-initiated processes in your user UCIs must have read/write/delete access to a number of % globals in the MGR UCI. The specific globals are:

^%WLDGASA
^%WLDSTAC
^%WLDERR
^%WLDGUID
^%WLDSESS

- When configuring WebLink Developer, remove the %SYS namespace from the drop-down list and in its place add your MSM UCIs; for example:

MGR ,MGW
USR ,MGW



Restarting Hung Processes

If a process gets stuck, the first thing to note is that you should **not** have to reboot. The problem is worse when using State-Aware Mode. For this reason, you should use develop your application in Stateless Mode, and then switch (if you so desire) when your application is bug-free.

The built-in \$I save and restore mechanism used by WebLink Developer should reduce the risk of problems, but runaway processes can still cause problems.

If a connection does get stuck, do the following:

1. From Programmer Mode, use **D ^%SS** or the equivalent GUI utility. All WebLink jobs should be running %MGW and will be connected to TCP ports incrementing from that used by the PCD.

If any WebLink job is stuck in another routine (ANY routine), it will need to be closed down. Do this first through WebLink:

- A. Bring up the **WebLink Systems Management** page and select **System Status**. Close down the stuck connection(s) by clicking the **Close** button.
 - B. Go back to **^%SS** and verify that the stuck jobs are gone. If not, use **^RESJOB** on them. They should now be working again.
2. If the previous step did not work and you still have problems, go back to WebLink and select **Close Down M Sessions**, and check the PCD to close it down also, to ensure everything is cleared. Check in **^%SS** that all %MGW jobs have gone and use **^RESJOB** on any WebLink connection jobs still running. Restart the PCD and all should be well again.
 3. If the above steps did not work, you must shut down and restart the Web server. You can then restart the PCD.



InterSystems Corporation

Copyright Notification

Copyright © InterSystems Corporation, 1999
All rights reserved.

These documents contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

M/SQL®, *M/PACT®*, and *M/NET®* are registered trademarks, and *InterSystems™*, *Caché™*, *Caché WebLink™*, *Caché WebLink Developer™*, *Caché SQL™*, *ISM™*, *DTM™*, *DT-MAX™*, *DT Windows™*, *DSM™*, and *DASL™* are trademarks of InterSystems Corporation.

Microsoft® is a registered trademark and *Windows™* and *Windows NT™* are trademarks of Microsoft Corporation.

For Support questions about any InterSystems products, contact the InterSystems Worldwide Support Center:

Phone: US: +1 617 621-0700 Europe: +44 (0) 1753 830-077

Fax: US: +1 617 374-9391 Europe: +44 (0) 1753 861-311

Internet Contact — support@intersys.com

[FTP Site](#)

[World Wide Web](#)

European BBS: +44 (0) 1753-853-534