4.900

ProWeb Server User Guide

by Rebecca Shalfield

ProWeb Server User Guide

The contents of this user guide describe the product ProWeb Server version 4.31, and are believed correct at the time of going to press. They do not embody a commitment on the part of Logic Programming Associates (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Logic Programming Associates Ltd.
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London
SW18 3SX
England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
web site: http://www.lpa.co.uk/

19 May, 2010

Contents

Chapter 1 - Introduction to the ProWeb Server

This chapter gives a brief overview of what ProWeb does and how it works.

Introduction to the World Wide Web

The World Wide Web (WWW) is a hypertext-based information system that supports the use of multimedia, including text, hyperlinks, graphics, video and sound.

The nature of the World Wide Web is that communications consist of discrete and discontiguous interactions. It's like speaking once on the telephone and then hanging up as soon as a reply from the recipient is received. Normally, there is only a short time period between each interaction, although such interactions could potentially be hours or even days apart.

Until fairly recently, the World Wide Web has been used mostly as an infrastructure for document distribution. There is, however, a growing need for the ability to distribute applications in addition to static documents so as to enable users to use the information they now have access to instead of just reading it. Products such as Sun's Java and Netscape's JavaScript address this issue by embedding the application inside the HTML document and having it interpreted on the client machine; this approach obviously requires that every browser on every client has the intelligence built-in in order to run such applications. This solution is unsatisfactory for a number of reasons. Firstly, an application expected to run on a variety of operating systems can make very few assumptions about the execution environment. Secondly, the code must be relatively small in size to accommodate an acceptable time for transmission over a network. Thirdly, some applications cannot assume the presence of an interpreter for their particular language, resulting in applications being written in the popular languages for which widespread browser support is known to exist. Fourthly, it cannot make assumptions about the amount of resources available (e.g. memory) on the client machine and must, therefore, keep its requirements low.

The definition of client implied within this user guide is a web browser that is programmed to communicate with and ask for information from a program running on a World Wide Web server. Server, on the other hand, means a World Wide Web server that is programmed to communicate with and provide information to a client (i.e. web browser).

What Does the LPA ProWeb Server Do?

The LPA ProWeb Server (hereinafter called ProWeb) addresses the problems outlined above by putting the intelligence on the server's side of the connection rather than the client's; bandwidth problems are, therefore, of little concern to the developer or the client and the application is always run in a known and controllable environment. This approach also makes applications available to a client that could not be run on their own machine.

ProWeb is a specialised extension to a Hypertext Transport Protocol (HTTP) server. ProWeb is written in Prolog and supports the development, testing and deployment of

intelligent, dynamic server-based applications on intranets and the Internet. ProWeb basically provides a link between an HTML page displayed on your client's web browser and your application (developed in ***WIN-PROLOG*** and associated toolkits) running on your server. When a client views HTML pages in the traditional way, there is normally no 'intelligent' interaction between the client and the server - the client simply asks for the next HTML page via its Uniform Resource Locator (URL) and it is dutifully returned by the server. With ProWeb, your clients can effectively 'talk' directly to your ProWeb-based application running on your server, using HTML pages as the means to pass back and forth the many interactions generated during such conversations.

The LPA ProWeb Server allows web sites to use the powerful reasoning capacity of ***WIN-PROLOG*** and associated toolkits completely in the background with HTML, Javascript, Java and other standard tools providing the user interface. Prolog is a very powerful technology, ideal for building intelligent solutions. The Web is saturated with information; low-level languages do not have the capacity to sift, sort and make sense of such a huge volume of data in a meaningful way. ProWeb brings the power of the industry-leading ***WIN-PROLOG*** to the Web, and so provides an ideal platform for building intelligent agents, smart search engines, diagnostic systems and data mining utilities.

What is CGI

CGI, or "Common Gateway Interface", is a specification which allows web users to run programs from their computer. CGI isn't a programming language in itself; rather, it is standard that allows programs or scripts written in other languages to be run over the Internet. CGI programs usually take input passed to it from a form on a web page, process the information, and then formats the results as a HTML document. The result is a web page that is generated dynamically.

ProWeb hides the complexity of CGI programming and HTML forms from the developer; ProWeb automatically generates all the required HTML and CGI code needed for your application to run smoothly on the Web. By handling all the communication between the forms and the application logic, ProWeb relieves the developer of all the tiresome book-keeping. This provides an intuitive and robust way for developers to build 'live', dynamic applications. The term 'dynamic application' is not to be confused with the term 'dynamic HTML', which is something completely different.

The features of ProWeb can be summarised as follows:

- The next HTML page to be sent to your client can be intelligently selected.

- Custom HTML pages can be generated 'on the fly' depending upon answers submitted on previous pages.

- Many clients may access your application simultaneously (limit being developer-defined), with each client being allowed a certain amount of time to complete their conversation (time being developer-defined).

- The power of Prolog's built-in backtracking can be harnessed to generate multiple solutions to a problem.

- The entire control mechanism for your web site can be contained in only one file.

- The asynchronous nature of the Web is 'hidden' from the programmer.

- Supports rapid prototyping of applications.

Backtracking on the Web

If you were to ask any Prolog programmer why they develop their applications in Prolog, they would at some point mention the backtracking capabilities of the language. It would therefore be largely unthinkable to develop a Prolog application without using such an ability. It is considered a strength too important to lose and the one reason why Prolog has proved a difficult language to interface to the Internet. Although such backtracking capabilities are preserved within ProWeb, a number of overheads will have to be incurred and a certain programming style is imposed on the developer.

ProWeb keeps track of the current execution state for each client and uses this information to support backtracking through the search space to provide alternative solutions where appropriate. Due to the web's asynchronous nature, getting to the correct continuation point in a conversation is especially hard. The Prolog language has conventionally dealt with such a situation through a mode of operation known as the "save state" model. This current release of ProWeb adopts a different mode of operation – the "recompute" model. With this model, your application is re-run from the beginning every time a response is received from a client. With the "recompute" model, an overhead in execution time is incurred as some of the code within your application is executed many times; substantial acceleration can be achieved however because ProWeb allows partial solutions, computed and stored in one run through, to be retrieved in the next. ProWeb uses this method itself to ascertain which pages have already been sent to a client and what answers were given to which questions.

The "save state" model may be incorporated into a later release of ProWeb, to be used instead of or alongside the "recompute" model; any code you develop will be unaffected by this change. Despite removing the overhead in execution time, the "save state" model possesses its own overhead, that of requiring the server to have a large amount of hard disk space.

Going Back in a Conversation

During a client's session on "the net", their web browser will most probably store previously loaded pages in a cache; by redisplaying such pages, a client is effectively given the opportunity to 'go back' in a conversation. If this is performed during a client's conversation with a ProWeb-based application, ProWeb will function as if the previous submission of the page and any subsequent pages had never taken place. ProWeb will, however, remember any submitted data for such pages and, upon re-sending a page to the client, any form it contains will be pre-filled with such data.

One further model that ProWeb could adopt in this situation is the "undo" model; although potentially very useful, it is very hard to execute and because it is currently not performed by ***WIN-PROLOG***, it can not, therefore, be utilised by ProWeb.

Serving Multiple Clients

The nature of the web allows many clients to view the same page on your server at the same time. This means that many clients could potentially send a response to your ProWeb-based application at any one time. By assigning a unique identification tag to each conversation, ProWeb is able to handle several conversations at the same time.

When a response is received from a client, it is executed immediately by your application unless busy, in which case the response is placed in a queue until your application becomes available. Assigning a unique identification tag to each conversation also ensures that no two conversations can interfere with one another (except for external side-effects).

Although ProWeb is able to cope with most internal side effects, thereby preventing the actions of one conversation from affecting another, external side effects are a different matter. If you, for example, allowed one client to delete files from your hard disk or modify the source code for the application itself, such actions will be 'visible' to all other clients; this will possibly be handled differently in later releases of ProWeb.

Control over Granularity of Conversations

ProWeb provides programmatic control over the number of forms/questions to be answered in a single interaction; you can let the client set this at the start of their conversation if you wish. Some clients will only want to answer one form at a time, others will want to answer many.

Support for Interactive Development

During development of more complex ProWeb application, the client-server communication channel can be simulated either by a copy-and-paste method between the client (any web browser) and the server or by an automated method for those using Microsoft Internet Explorer or Netscape Navigator. The above allows the ***WIN-PROLOG*** debugger to be utilised.

Error Handling

Effective error handling capabilities are essential for a server that executes user-defined code instead of just retrieving files. Errors can arise whilst the application is being developed or after it has been deployed due to incorrect input or unexpected changes in the state of the server. In all such cases, the server must be able to signal the error in an appropriate way and to resume normal operation. ProWeb will trap any runtime errors and report them back through to the client via their web browser.

What Does ProWeb Consist of?

In addition to supporting the standard HTTP protocol, ProWeb consists of a suite of high-level Prolog predicates that fully enable the development, testing and deployment of intelligent, dynamic server-based applications on intranets and the Internet. ProWeb automatically maintains the flow of a conversation by keeping a record of each interaction.

When an application has been deployed the communication channel between the client and server is automatically handled by the Common Gateway Interface (CGI). The Common Gateway Interface is a protocol that enables developers to write programs that create HTML in response to user requests. The CGI is a standard series of environment variables. It is used by ProWeb to take input from a form embedded in an HTML page. ProWeb then generates an appropriate HTML page in response to the inputs. Such a CGI lies at the heart of ProWeb and is seamlessly integrated with your ProWeb-based applications. From each client the CGI will accept a couple of lines of information that tells the server what it should be doing. The server can then send back data; in the case of ProWeb this would be an HTML document either retrieved from disk or created 'on the fly', or a combination of the two.

ProWeb's Execution Mechanism

To set the scene, you have ProWeb up and running on your server and one of your clients communicating with the server via their web browser. A typical 'live' World Wide Web ProWeb conversation will be as follows:

1. Client requests your ProWeb 'launch page', via its URL, from your server.

2. Your ProWeb 'launch page', being a disk-based HTML page, is returned to the client in the traditional way.

3. Upon receipt, your client completes the page's form

4. Client submits the form by clicking on the page's 'submit' button. The information entered into the form is sent to the CGI running on your server.

5. If ProWeb is not already running, it will be launched.

6. Your client's URL line is received by ProWeb, whereupon the relevant information is extracted and passed to ***WIN-PROLOG*** and your application. If ***WIN-PROLOG*** and your application are not already running, ProWeb will automatically launch them.

7. Your application restarts from the beginning.

8. A non-deterministic computation, based on the information just supplied, is performed.

9. The next HTML page is generated and sent to the client.

10. Your application terminates. Due to the large program size and relatively slow startup time, ***WIN-PROLOG*** will continue running for the time being.

11. Client receives next page. If the page contains a form requiring completion, flow of execution returns to, and continues from, step 3.

12. Conversation has come to a natural end.

13. After a predefined period of inactivity, the ProWeb and ***WIN-PROLOG*** executables will close themselves down.

Editing Your ProWeb Code

Probably the easiest way to develop your own ProWeb programs is to edit your Prolog code using your favourite editor, and then simply save the file and refresh the browser to force the affect of the update into play.

Generally, you will want to develop the logic for your application first using the standard Prolog IDE and get the benefit of dynamic syntax coloring and first class debugging.

Unicode Support

Proweb runs as a Unicode application.

Each HTML page generated by ProWeb is sent to a client with the following header:

Content-type: text/html; charset=utf-8

To see Unicode characters in your web browser, you may need to install a Unicode font, such as Arial Unicode (supplied on the Microsoft Office CD-ROM), and select this as your default font in the web browser itself.

ProWeb Help Pages

There are various examples and on-line help pages supplied with ProWeb.

Chapter 2 - Installing and Setting-up ProWeb Server

This chapter describes how to install the ProWeb Server toolkit and how to make your web server software ProWeb-aware.

Requirements

You can develop, test and deploy ProWeb-based applications on a machine running Microsoft Windows Windows 7, Vista, XP, ME, 2000, NT 4.0 or 98. You will also need to have installed some web server software such as IIS or Apache and a browser. You will, of course, also need a copy of ***WIN-PROLOG***.

IIS Web Server Software

IIS is not turned on by default when Windows is installed, but can be selected from the list of optional features.

If you are running Windows 7, the IIS 7.5 web server software is present on all editions.

If you are running Vista, the IIS 7 web server software is present on the Home Premium, Business, Enterprise and Ultimate Editions.

If you are running XP, the IIS 5.1 web server software is present on the Windows XP Professional CD-ROM. No web server software is supplied with the Home Edition of XP.

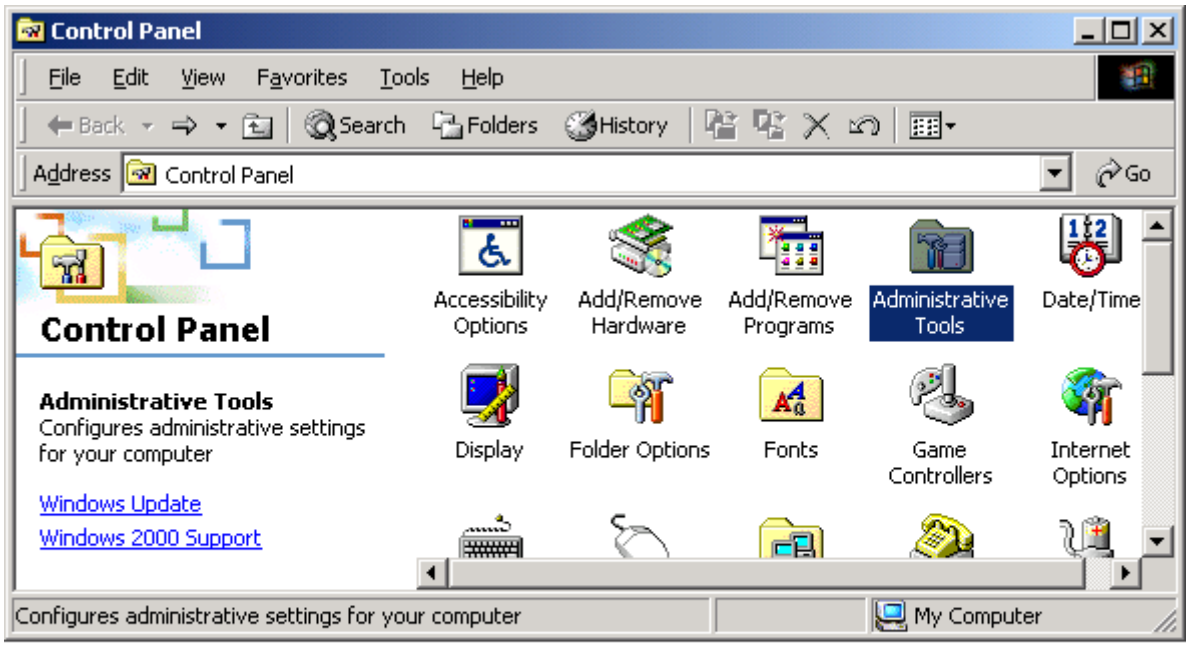
If you are running NT 4.0, IIS 4.0 is supplied on the Windows NT 4.0 Workstation CD-ROM and is downloadable from Microsoft as part of the Windows NT Option Pack. Equivalent products from other parties are also available, such as Netscape's FastTrack Server.

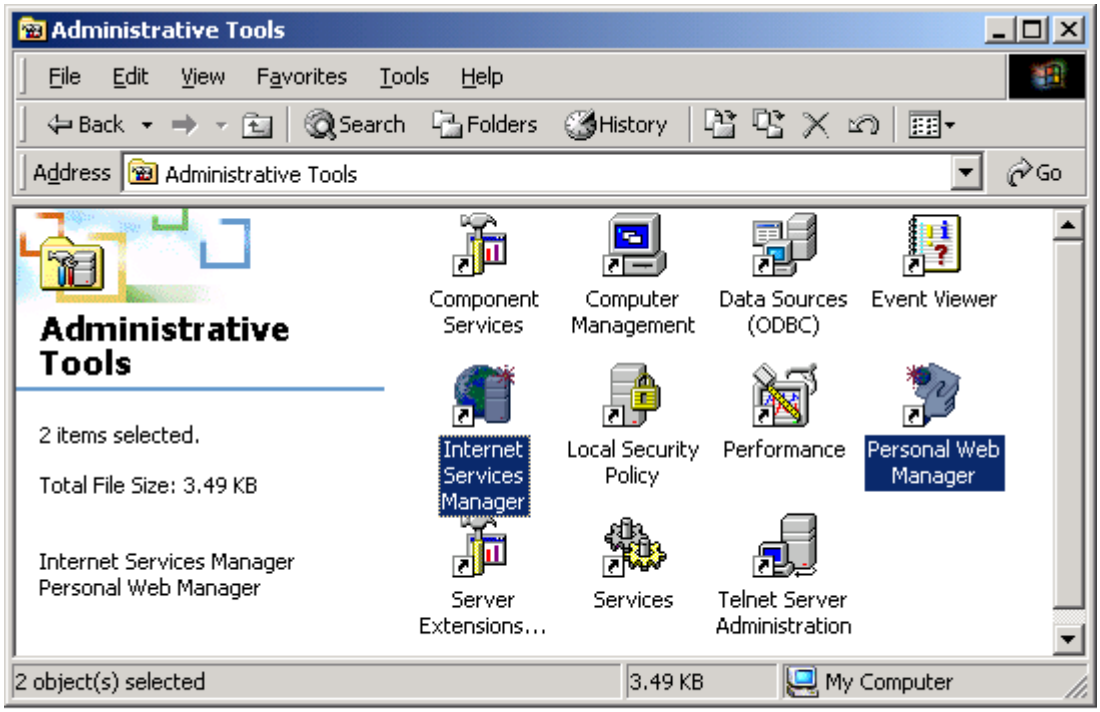
The Windows NT Option Pack can be downloaded free from the Microsoft Web Site at http://www.microsoft.com/. When executed on NT, the Windows NT Option Pack will install IIS; when executed on Windows 9x, the Windows NT Option Pack will detect that it is not running on NT and install PWS instead.

Much of the installation process is concerned with how to configure your web server to handle and allow CGI applications to work. So ProWeb Server is nothing particularly unusual, this is not as simple as it used to be as Microsoft tightens up on its security processes.

Apache Web Server Software

The Apache HTTP Server project is an effort to develop and maintain an Open Source HTTP server for modern operating system including Windows.





The rest of this chapter assumes that you have IIS.

Setting-Up IIS

If you are running Windows XP Professional on your computer you can install Microsoft's web server, Internet Information Server 5.1 (IIS) for free from the Windows XP Pro installation CD and configure it to run on your system by following the instructions below: -

1. Place the Windows XP Professional CD-Rom into your CD-Rom Drive.

2. Open 'Add/Remove Windows Components' found in 'Add/Remove Programs' in the 'Control Panel'.

3. Place a tick in the check box for 'Internet Information Services (IIS)' leaving all the default installation settings intact.

4. Once IIS is installed on your machine you can view your home page in a web browser by typing 'http://localhost' (you can substitute 'localhost' for the name of your computer) into the address bar of your web browser. If you have not placed your web site into the default directory you should now be looking at the IIS documentation.

5. If you are not sure of the name of your computer right-click on the 'My Computer' icon on your desktop, select 'Properties' from the shortcut menu, and click on the 'Computer Name' tab.

6. Your default web directory to place your web site in is 'C:\Inetpub\wwwroot', but if you don't want to over write the IIS documentation found in this directory you can set up your own virtual directory through the 'Internet Information Services' console.The Internet Public directory, C:\INETPUB, will now have been created.

7. The 'Internet Information Services' console can be found in the 'Administration Tools' in the 'Control Panel' under 'Performance and Maintenance', if you do not have the control panel in Classic View.

8. Once the 'Internet Information Services' console is open you will see any IIS web services you have running on your machine including the SMTP server and FTP server, if you chose to install them with IIS.

9. To add a new virtual directory right click on 'Default Web Site' and select 'New', followed by 'Virtual Directory', from the drop down list.

10. Next you will see a 'Browse...' button, click on this to select the directory your web site pages are in on your computer, after which click on the 'next' button to continue.

11. On the final part of the wizard you will see a series of boxes, if you are not worried about security then select them all, if you are and want to run ASP scripts then check the first two, followed by the 'next' button.

12. Once the virtual directory is created you can view the web pages in the folder by typing 'http://localhost/aliasName' (where 'aliasName' is, place the alias you called the virtual directory) into the address bar of your web browser (you can substitute 'localhost' for the name of your computer if you wish).

Setting-Up IIS 7 on Vista and Windows 7

1. To open the Windows Features dialog box, click Start, and then click Control Panel.

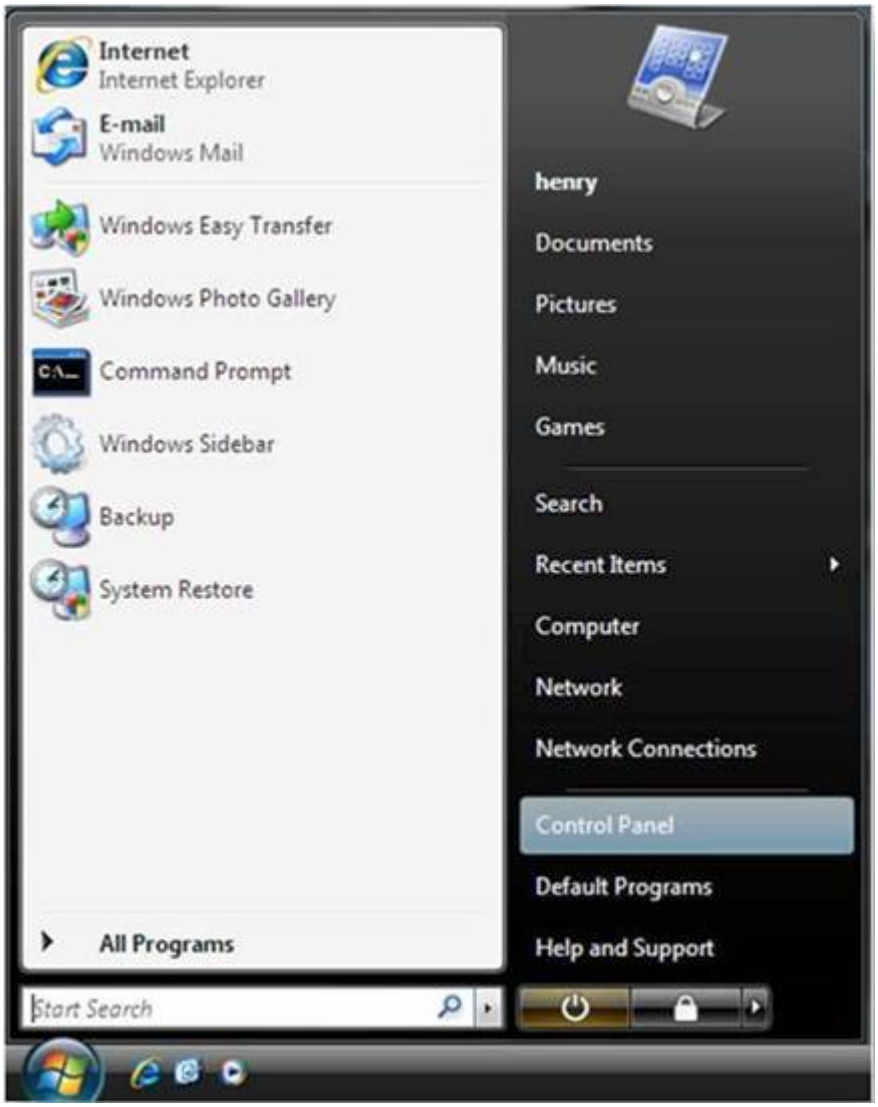


Figure 2: Windows Vista Start menu

2. In the Control Panel, click Programs.

Figure 3: Control Panel Home

3. Click Turn Windows features on or off.

Figure 4: Control Panel install options

4. You may receive the Windows Security warning. Click Allow to continue. The Windows Features dialog box is displayed.

Figure 5: Windows Security dialog box

5. Expand Internet Information Services. Additional categories of IIS features are displayed. Select Internet Information Services to choose the default features for installation.

Figure 6: Windows Features dialog box - IIS

6. Expand the additional categories displayed, and select any additional features you want to install, such as Web Management Tools.

Figure 7: Windows Features dialog box - IIS feature selection

7. If you are installing IIS 7 for evaluation purposes, you may want to select additional features to install. Select the check boxes for all IIS features you want to install, and then click OK to start installation.

Figure 8: Windows Features dialog box - Installation selection

8. The progress indicator appears.

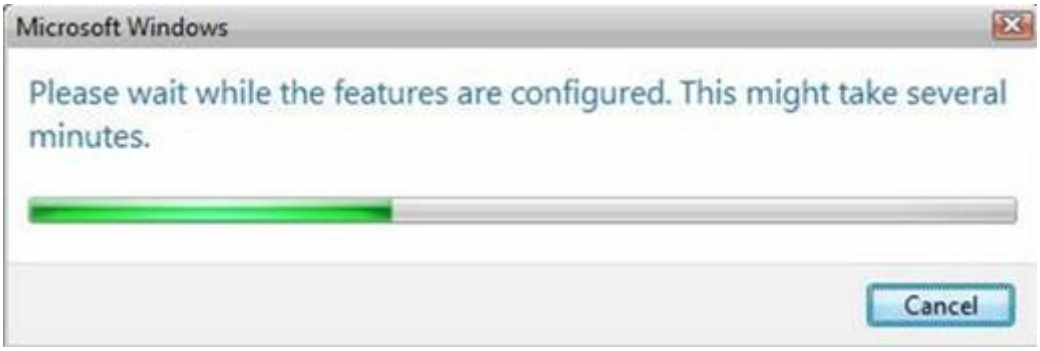


Figure 9: Progress indicator

9. When the installation completes, the Windows Features dialog box closes, and the Control Panel is displayed.

Figure 10: Control Panel Home page

10. IIS 7 is now installed with a default configuration on Windows Vista or Windows 7. To confirm that the installation succeeded, type the following URL into your browser, http://localhost.

Figure 11: Default Web site

11. Next, you can use the Internet Information Services Manager to manage and configure IIS. To open the IIS Manager, click Start, type inetmgr in the Search Programs and Files box, and then press ENTER.

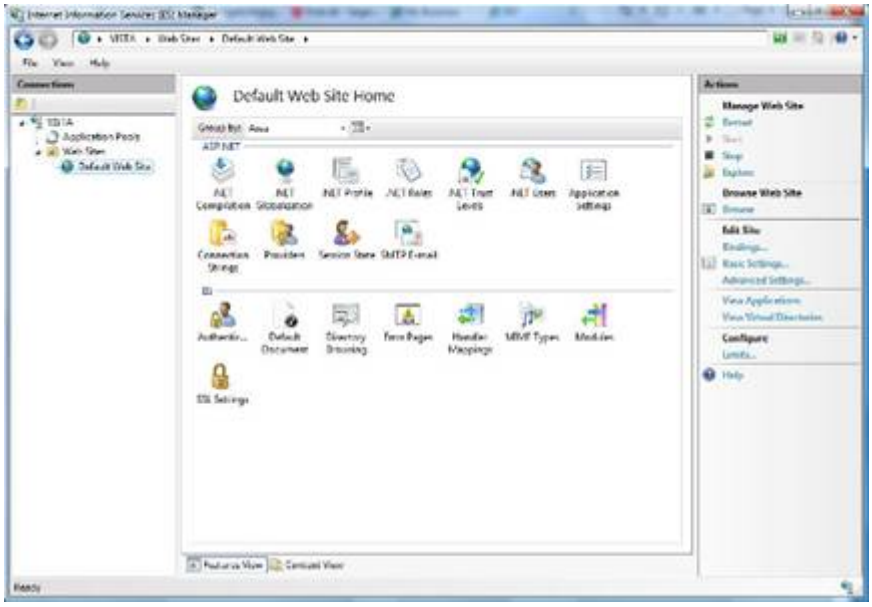


Figure 12: Default Web Site Home page

Installing CGI in IIS 6

CGI programs are executed when the Web server receives a URL that contains the CGI program name and any parameters that it requires. If your CGI program is compiled into an executable (.exe) file, you must give the directory that contains the program Execute permissions so that users can run the program.

You must be a member of the Administrators group on the local computer to perform the following procedure or procedures. As a security best practice, log on to your computer by using an account that is not in the Administrators group, and then use the runas command to run IIS Manager as an administrator. At a command prompt, type runas /User:Administrative_AccountName"mmc %systemroot%\system32\inetsrv\iis.msc".

Procedures to install and configure CGI applications

Set up a directory for your CGI programs. For extra security, separate your CGI programs from your content files. You do not need to name the directory Cgi-bin, although you can do so if you want. For more information, see Using Virtual Directories.

If your CGI programs are .exe files, give the directory Execute permissions. If your CGI programs are scripts, you can give the directory either Execute permissions or Script permissions.

If you choose to give the directory Script permissions, you must mark the script interpreter as a script engine in the directory properties. Only interpreters that are marked as script engines are allowed to execute in the directory. Executable files (.dll and .exe files) cannot be directly executed; that is, a browser request cannot launch an executable file on the Web server by including the program name in the URL. By enabling Scripts only permissions and selecting the Script engine check box, you can safely put content files (such as .htm or .gif files) in the same directory as your CGI scripts. Content files are displayed in the browser and scripts are executed, but no one can run an unauthorized program and script commands are not displayed in the browser.

For CGI scripts, create an application mapping between the file name extension of your script and the script interpreter.

IIS maps file name extensions to an interpreter.

Set NTFS permissions. If you are using NTFS access permissions, verify that all users who need to run the program have Execute permissions for the directory. If your Web site accepts anonymous users, make sure that the anonymous user (the IUSR_computername account) has Execute permissions.

PINGing Your Web Server

Once you think you have your web server software set up correctly, you could try pinging your server from a DOS box, with both its name and IP address:

C:\>PING <server_name><Return>
Pinging <server_name> [<IP_address>] with 32 bytes of data:
Reply from <IP_address>: bytes=32 time<10ms TTL=128

C:\>PING <IP_address><Return>
Pinging <server_name> [<IP_address>] with 32 bytes of data:
Reply from <IP_address>: bytes=32 time<10ms TTL=128

Assuming you get something similar to the above returned, your web server is set up correctly.

Installing *WIN-*PROLOG and the ProWeb Server Toolkit

The ***WIN-*PROLOG** system is installed onto your hard disk via use of the industry standard-based LPA Install Program (supplied with ***WIN-*PROLOG**). The ProWeb Server toolkit is only available on the ***WIN-*PROLOG** CD-ROM. If you are installing ***WIN-*PROLOG** for the first time, make sure that you select ProWeb, within the Select Components dialog, as one of the toolkits you wish to install. The ***WIN-*PROLOG** installation/setup procedure is documented in *Chapter 1 - Installing* ***WIN-*PROLOG** *and Toolkits* in the ***WIN-*PROLOG** User Guide.

Once you have ***WIN-*PROLOG** and the ProWeb Server toolkit installed, you will find that the files making-up the ProWeb Server toolkit have been placed into a PROWEB directory within the ***WIN-*PROLOG** home directory.

Creation of a ProWeb Home Directory

The directory of a typical web-based application will contain executable and initialisation files, HTML and image files, not to mention temporary files. To keep these in some sort of order for your ProWeb-based applications, it is quite important that ProWeb has its own structured 'home directory'; the PROWEB directory within the ***WIN-*PROLOG** home directory is such a directory. In order to run ProWeb, you will need to move the entire PROWEB directory into your Internet Public directory (i.e. C:\INETPUB). This user guide assumes that the PROWEB directory will be placed directly inside C:\INETPUB to become C:\INETPUB\PROWEB, although it does not have to have this name or reside at this level in the hierarchy. If the setup of your server requires that the Internet Public area be on a drive other than C:, ProWeb will be happy with this too.

Setting-Up The /PROWEB Alias

You next need to set-up an alias for the C:\INETPUB\PROWEB directory within your web server software. We recommend that this alias be called '/PROWEB'.

Register the '/PROWEB' alias with the Service Manager as both readable and executable. This task can be performed by following these steps (the first of which you may have just done):

- Display the Microsoft Internet Server (Common) folder (or the Microsoft Peer Web Services (Common) folder).

- Click on the Internet Service Manager application.

- Double click on your server's name.

- Select the Directories tab.

- Click on the Add button.

- Enter *c:\inetpub\proweb* in the Directory field (the case used may be significant).

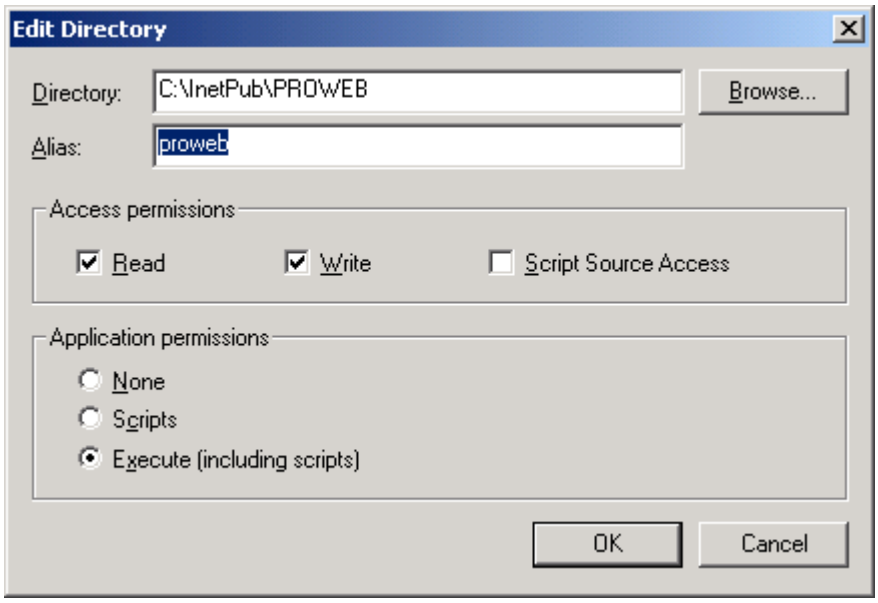
- Enter */proweb* in the Virtual Directory Alias field (the case used may be significant).

- Select the Access Read checkbox button

- Select the Access Execute checkbox button

- Click on the 'OK' button

Prior to clicking on the 'OK' button, the "Directory Properties" dialog will look something like (it may, however, appear as an HTML page in a browser in certain setups):



If you are using Netscape's FastTrack Server, the default Internet Public area is C:\NETSCAPE\SERVER instead of C:\INETPUB. All you need to do is to move the entire PROWEB directory into C:\NETSCAPE\SERVER and point the '/PROWEB' alias to C:\NETSCAPE\SERVER\PROWEB instead.

You will now need to reset your machine for the above to take effect.

Web Browser

In order to communicate with your ProWeb-based application, you are going to need to install a suitable web browser (e.g. Netscape Navigator or Internet Explorer). Because of the evolving nature of the web, it is best to install a couple of web browsers for testing purposes anyway.

Changing The PROWEB.INI Path/Alias Settings

The PROWEB.INI file is supplied with the following path and alias settings:

```
TEMP_PATH=c:\inetpub\proweb\temp\
HTML_PATH=c:\inetpub\proweb\html\
BASE_URL=http://localhost/proweb/
TEMP_URL=http://localhost/proweb/temp/
```

The settings assume that ProWeb has been installed into C:\INETPUB\PROWEB; if installed elsewhere, change the settings accordingly. You will also need to change 'server_name' to whatever your server is called.

Displaying The Supplied ProWeb Launch Page

Assuming the rest of your web server has been set up correctly, you should now be able to log onto your intranet and submit the URL of an HTML page, whereupon it should be displayed in your web browser.

A Uniform Resource Locator (URL) is an Internet address (i.e. an absolute link) composed of:

- the protocol type (i.e. http:).

- the name of the server to be contacted (i.e. www.lpa.co.uk).

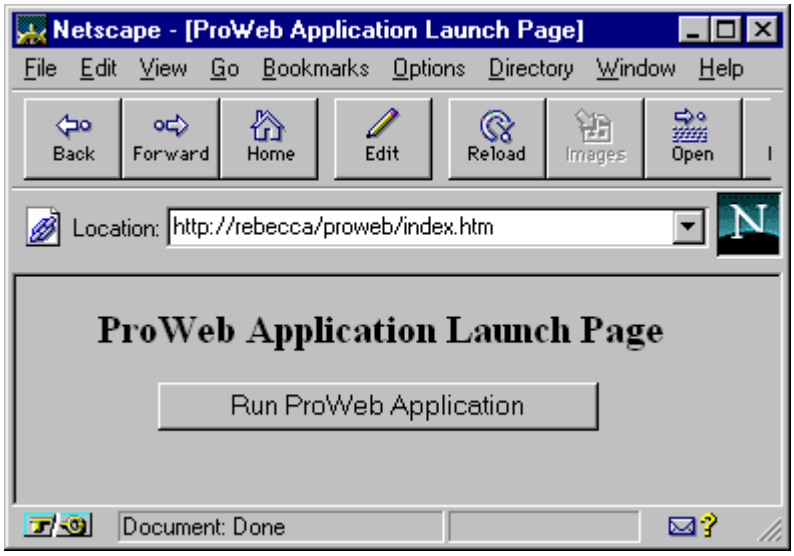
- the directories or folders (i.e. /proweb).

- the optional filename (i.e. index.htm).

A ProWeb 'launch page' has been set up for you in the supplied PROWEB directory, so let's see if we can now display it. From your web browser, execute the following URL:

http://<server_name>/proweb/index.htm

where <server_name> is the name given to your server. You should now see the following HTML page displayed in your browser:

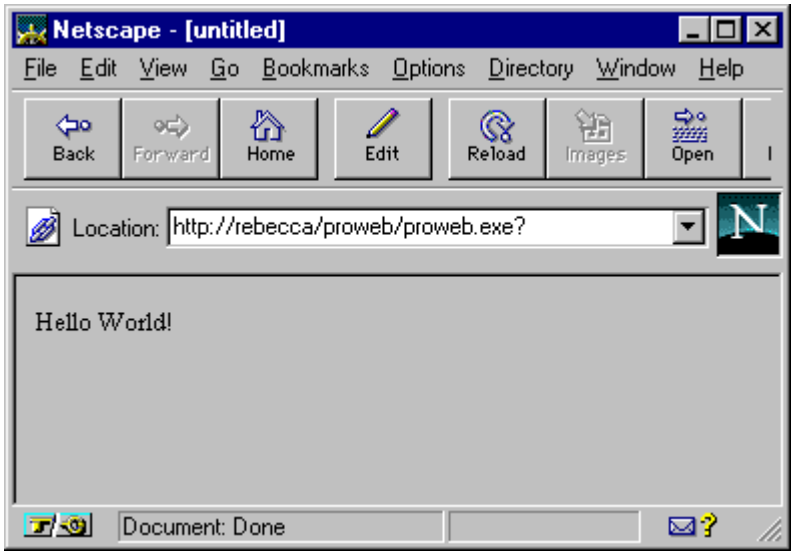


Getting The PROWEB.SYS File

The supplied ProWeb directory is an almost complete installation of ProWeb. Only one file is missing; this being PROWEB.SYS. The file, PROWEB.SYS, is a renamed copy of the ***WIN-PROLOG*** executable, PRO386W.EXE. Copy PRO386W.EXE, which you'll find in your ***WIN-PROLOG*** root directory, to your ProWeb home directory (i.e. C:\INETPUB\PROWEB) and rename it PROWEB.SYS.

Running Your First ProWeb-Based Application

If you click on the "Run ProWeb Application" button, you should see the following HTML page appear:



Congratulations, you have just run your first ProWeb application!

Trouble-Shooting

Problem Encountered	Explanation and Solution
I get an "HTTP 1.0 500 Server Error".	PROWEB.EXE is not where the ProWeb launch page (INDEX.HTM) says it is.
Whenever I click on a button within the ProWeb 'launch page', I get a "Netscape is unable to find the file or directory named: /proweb /proweb.exe Check the name and try again" error message appear.	You have loaded the page locally, perhaps by double clicking on the INDEX.HTM file directly. Load the file as: http://<server_name>/proweb/index.htm rather than as: file:///c|/inetpub/proweb/index.htm.
Whenever I click on the submit button of the ProWeb 'launch page', I get a "System Error (WEX:001:2)" error message page.	Ensure that you have copied the ***WIN-PROLOG*** executable file, PRO386W.EXE, to your ProWeb root directory (i.e. C:\INETPUB\PROWEB) and renamed it PROWEB.SYS. Ensure that it has only one extension and not PROWEB.SYS.EXE or some such.
Whenever I submit my ProWeb launch page, it wants to download PROWEB.EXE rather than execute it on the server.	You need to alter your server's settings to allow .EXE files to be executed on the server.

I keep getting a ProWeb error message	Does your computer have the two Microsoft Rich Edit DLLs; these are supplied with ***WIN-PROLOG*** for use on older versions of Windows.

Chapter 3 - HTML Forms

This chapter introduces the FORM-related elements of the HTML syntax.

Brief Introduction to HTML

It is assumed that you have an understanding of and have used the Hyper-Text Markup Language (HTML). It is beyond the scope of this user guide to go into any great depth about HTML except to mention those aspects applicable to ProWeb. Most web browsers, however, provide the user with the ability to view the HTML source of the displayed page; for beginners, this feature will prove extremely useful as it provides an opportunity to "learn by example" by matching the contents of a page with its associated HTML source code.

The Hyper-Text Markup Language is made up of elements and attributes. The FORM element, for example, consists of a <FORM> start tag and a </FORM> end tag; an associated attribute/value pair for this element might be METHOD=POST as in <FORM METHOD=POST>

HTML's Form-Related Elements

Forms are now one of the most popular features on the World Wide Web (WWW). A form allows you to interact with the text and graphics being displayed in your web browser. Forms can range from those which just ask you for your name and email address up to highly complex forms for, say, ordering goods over the Internet. Like Microsoft Windows itself, many input field types are available such as text, pop-up menu, password, checkbox and radio; all of these are available with ProWeb. Looking at the source code for such an HTML file you would see two elements of immense interest to us, namely <FORM> and <INPUT>, both of which we will use very shortly.

Fundamental Concepts

In the early days of computing, before the advent of the microcomputer, a program would run on a mainframe and be accessed via a number of dumb terminals; a situation that was to continue throughout the early days of the Internet. Although the many mainframe (i.e. server) computers still remain, all-be-it a lot smaller, the terminal (i.e. client) has been replaced with the powerful personal computer running a web browser.

The current trend on the Internet is for an application to be downloaded from the server and run on the client; such applications have their user interface inseparably linked with the rest of the program.

ProWeb allows a conversation to be conducted over an intranet or the Internet using the World Wide Web's conversational unit, the HTML page. Operating in a fashion more akin to the early days of the Internet, the ProWeb-based program is only run on the server with the client only getting the user interface component. Most of these HTML pages sent to the client will contain an HTML Form to be completed and returned; an HTML Form is very similar to a dialog box found under, say, Microsoft Windows, where

information is asked of the client (i.e. questions) and conveyed to the client (i.e. answers).

Chapter 4 - Launching and Stopping ProWeb

This chapter describes what a ProWeb launch page is and how to use it. It also documents how to stop ProWeb and why you would want to do so.

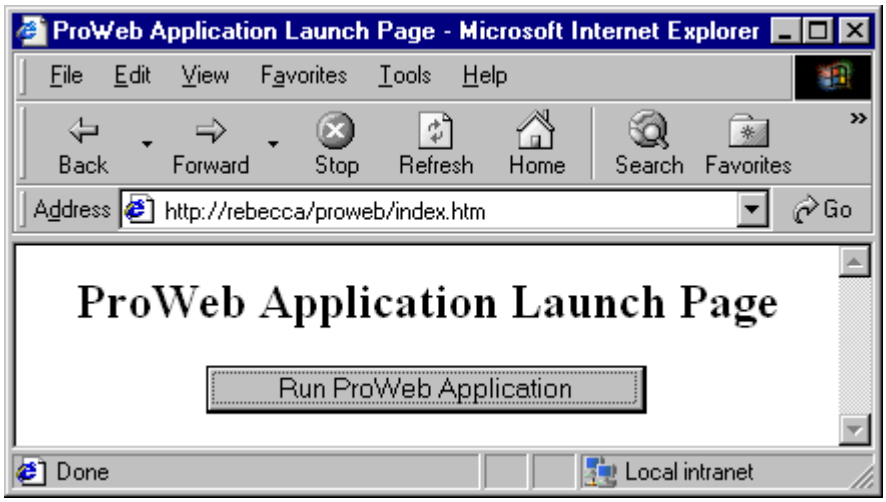
A ProWeb Launch Page

Like the majority of people starting to use ProWeb, it is highly likely that you already have an existing, carefully-designed, web site, which you do not wish to alter. ProWeb, being a web-based application, requires an HTML page from which to launch it. Adding a ProWeb-based application to an existing web site is simply a matter of providing a page from which the ProWeb-based application can be launched; in effect, a ProWeb 'launch page'. Such a page can be any page on your web site; in fact your ProWeb-based application could be launched from more than one page on your web site.

To get you up and running a little quicker, a ProWeb 'launch page' is included in the supplied PROWEB directory. In fact, you used this page at the end of the previous chapter to launch your first ProWeb-based application. From your web browser, execute the following URL:

http://<server_name>/proweb/index.htm

replacing <server_name> with your own server's name. This will load the file, C:\INETPUB\PROWEB\INDEX.HTM, into your browser.



The HTML source code of this page is:

```
<HTML>
 <HEAD>
  <TITLE>ProWeb Application Launch Page</TITLE>
  <META HTTP-EQUIV="content-type" CONTENT="text/html;charset=utf-8">
 </HEAD>
 <BODY>
  <FORM METHOD=POST ACTION="/proweb/proweb.exe">
   <DIV ALIGN="center">
    <H2>ProWeb Application Launch Page</H2>
```

```
    <INPUT TYPE=HIDDEN NAME="lpa" VALUE="examples">
    <INPUT TYPE=SUBMIT VALUE="Run ProWeb Application">
   </DIV>
  </FORM>
 </BODY>
</HTML>
```

Let's look at this page in detail: Within the <HTML> element (which every HTML file must have) we have the HEAD section followed by the BODY section. The BODY section itself contains a FORM. Every form must have a <FORM> tag at its beginning and a </FORM> tag at its end. The <FORM> element basically defines what script the form will use and how the information is to be sent to the script. The conventional meaning of a script would be a small utility written in a scripting language such as PERL; here, though, we will be passing the information to ProWeb. The METHOD attribute tells the form how to send its information to the server; in ProWeb's case, the POST method is being used. The ACTION attribute points the form at a URL that will accept the form's information and do something useful with it; 'ACTION="/proweb/proweb.exe?"' means send the information entered into the form to PROWEB.EXE in the PROWEB directory. The <INPUT> element is required for the gathering of information. For the ProWeb 'Launch Page', only one <INPUT> element has been defined, this being a submit button (having a TYPE="submit" attribute); this has a preset function to send the information in the form to whatever the ACTION attribute in the <FORM> tag is pointing to to be processed.

The EXAMPLES directory and the Default Example

The MAIN_GOAL setting in PROWEB.INI has been set up so that it will load a file named PW_DEMO.PL from the EXAMPLES directory; the '.PL' file extension indicates that this file is a Prolog source code file; it is unfortunate that the programming language Perl, with its close association with the Internet, shares the same extension. It will then execute the predicate *main_goal/0* as defined in PW_DEMO.PL; PW_DEMO.PL is simply a copy of PW_HW1.PL (which contains the code for the first 'Hello World!' example).

MAIN_GOAL=ensure_loaded(examples(pw_demo)),main_goal

When you want to run another example, be it one of the supplied examples or one of your own, simply delete the existing PW_DEMO.PL file and make a copy of the example file you want to run renaming it PW_DEMO.PL.

A more advanced method is to modify the MAIN_GOAL setting in PROWEB.INI…

MAIN_GOAL=ensure_loaded(examples(my_file)), my_main_goal

…but we'll come to that in a later chapter.

It is advisable not to develop an application that you might want to keep directly in PW_DEMO.PL as you may accidently delete or overwrite it.

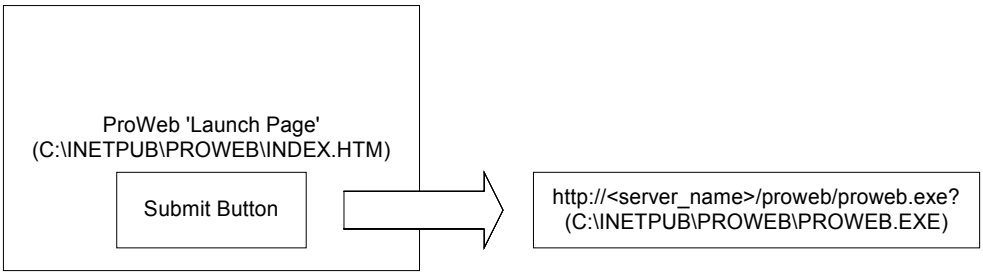
Shutting ProWeb Down

Before we can run any other example though, we need to ensure that ProWeb has shut itself down, otherwise it will use the version of PW_DEMO.PL already in its memory instead of loading and running the new one. Executing the following URL will shut ProWeb down immediately:

http://<server_name>/proweb/proweb.exe?!+001

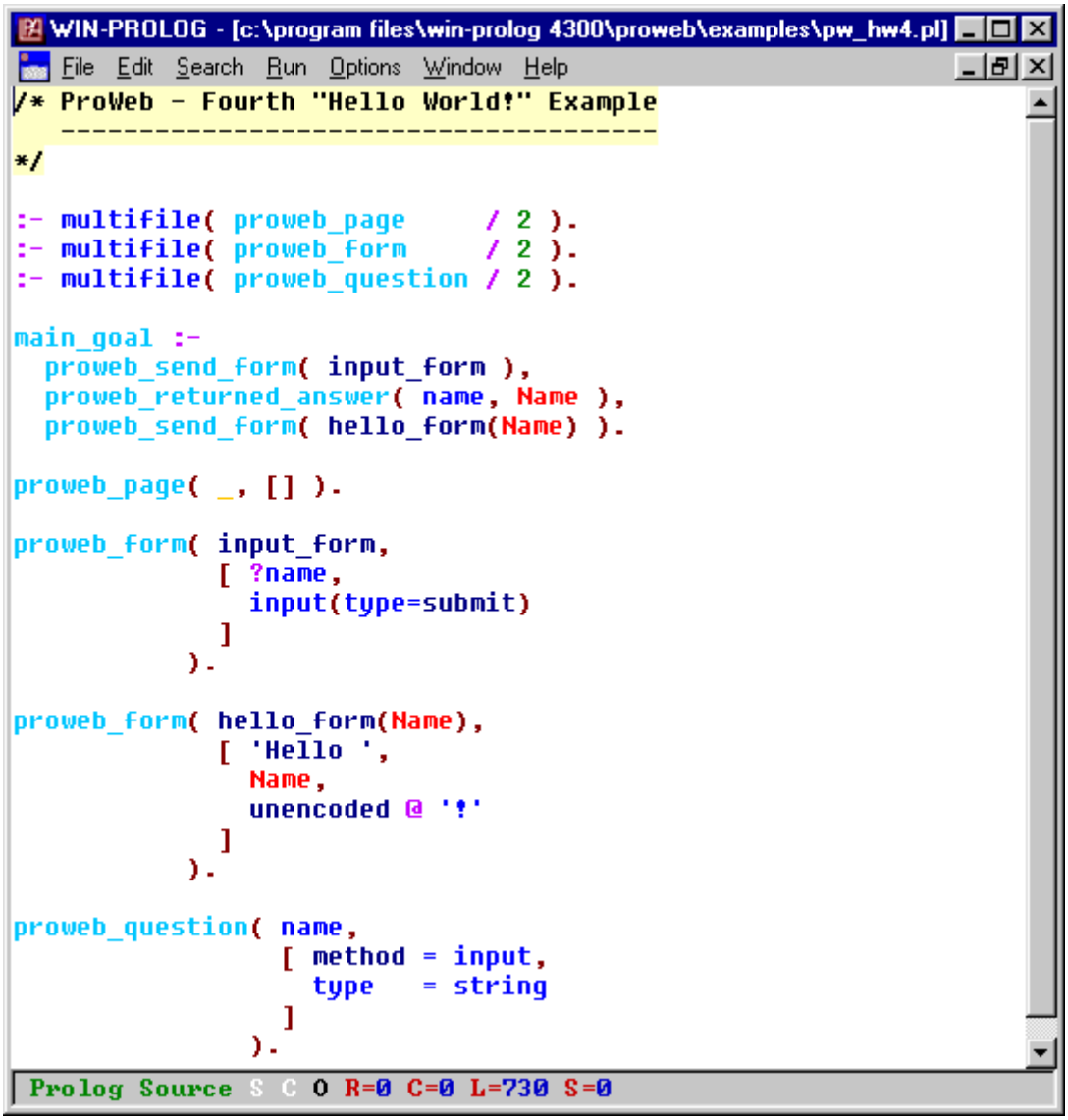
where <server_name> is the name of your server. If ProWeb is successfully shut down, a ProWeb System Error (REX:001:0) will be returned.

An easy way to redisplay the ProWeb 'launch page' in your web browser is to click on the back button one or more times; you will find this at the top-left directly underneath the file and edit menu options. Click on the web browser's back button now until the ProWeb 'launch page' is redisplayed.



Chapter 5 – Utilising The *WIN-*PROLOG Development Environment

The ***WIN-PROLOG*** development environment is the best place to create and edit your ProWeb programs as it provides automatic syntax colouring and syntax checking.



You can now create a new or load an existing ProWeb programs with automatic syntax colouring, but the ProWeb operators will not be present.

You can even create/edit your HTML files within the ***WIN-PROLOG*** development environment, although the syntax colouring will be incorrect. This is especially useful if you need to include Unicode characters in your HTML files and your usual text editor (e.g. Notepad) does not support Unicode.

Chapter 6 - First Steps For The Web Designer

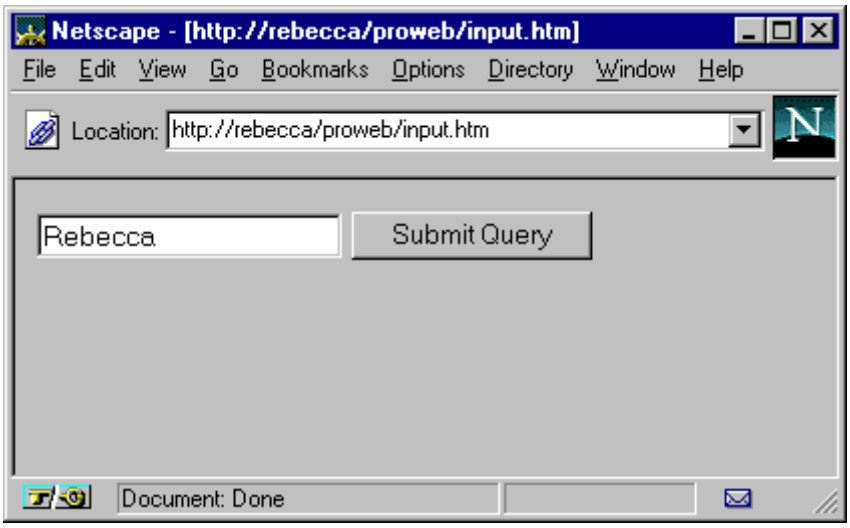
This chapter guides you through the creation of a "Hello World!" example from the point of view of a web designer.

The Non-ProWeb Approach

I am sure that as a web designer, you are use to writing HTML code and server side scripts. Let's take a look at a very simple HTML page containing a form:

```
<HTML>
 <HEAD></HEAD>
 <BODY>
  <FORM METHOD="post" ACTION="some_program_on_the_server">
   <INPUT TYPE="text" NAME="my_name">
   <INPUT TYPE="submit">
  </FORM>
 </BODY>
</HTML>
```

The dummy ACTION value of 'some_program_on_the_server' would be the actual name of a program which runs on the server, such as a Perl script. When submitted, the contents of the form would be passed to the script. Enter the above HTML code into your text editor or create via an HTML editor (such as FrontPage or PageMill) and save as the file, INPUT.HTM, in the C:\INETPUB\PROWEB directory. When displayed in a web browser, the page looks like the following (after 'Rebecca' was typed in):



When submitted, the browser tries to execute the generated URL, but fails to find 'some_program_on_the_server', resulting in an "HTTP/1.0 404 Object Not Found" page being returned:

What we now need to do is to modify INPUT.HTM so that the form is passed to ProWeb for processing instead.

Passing The Form To ProWeb

In order to pass the form to ProWeb, we just need to change ACTION="some_program_on_the_server" to ACTION="/proweb/proweb.exe":

```
<HTML>
 <HEAD></HEAD>
 <BODY>
  <FORM METHOD="post" ACTION="/proweb/proweb.exe">
   <INPUT TYPE="text" NAME="my_name">
   <INPUT TYPE="submit">
  </FORM>
 </BODY>
</HTML>
```

Re-save the file under the same name (i.e. INPUT.HTM).

Our First ProWeb Program

Open a new edit window in your text editor.

Now enter the following ProWeb program:

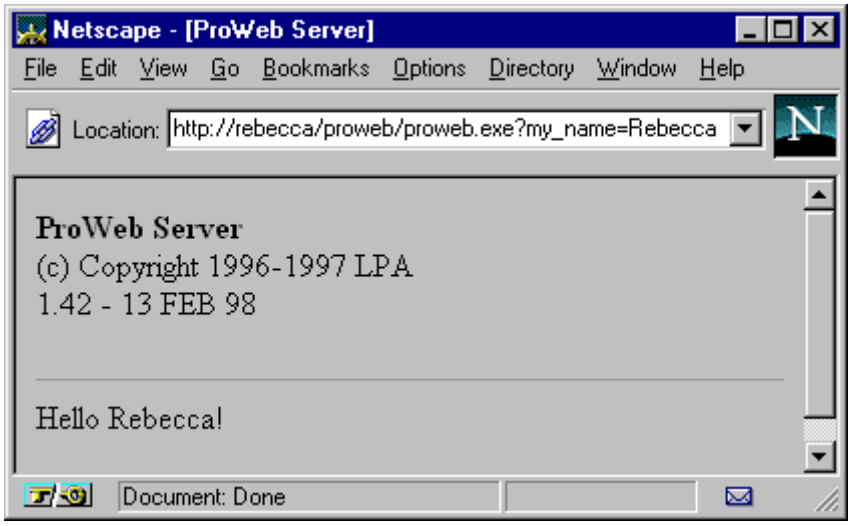
```
:- multifile proweb_form/2.

main_goal :-
  proweb_returned_input( my_name, MyName ),
  proweb_post_reply( your_name, MyName ),
  proweb_send_form( output_form ).

proweb_form( output_form, include('OUTPUT.HTM') ).
```

If you save this program as the file, PW_DEMO.PL, in the C:\INETPUB\PROWEB\EXAMPLES directory, you will be able to run it very shortly.

Before we can run the program, however, we need to create OUTPUT.HTM. Enter the following HTML code into a new text editor window or create via an HTML editor and save as OUTPUT.HTM in the C:\INETPUB\PROWEB directory:

```
<HTML>
 <BODY>
  <FORM>
   Hello <PROWEB REPLY="your_name">!
  </FORM>
 </BODY>
</HTML>
```

Go back in the browser and reload INPUT.HTM. When INPUT.HTM is submitted for a second time, the contents of the form will be passed to PROWEB.EXE whereupon the above ProWeb program will be executed and the following page returned:



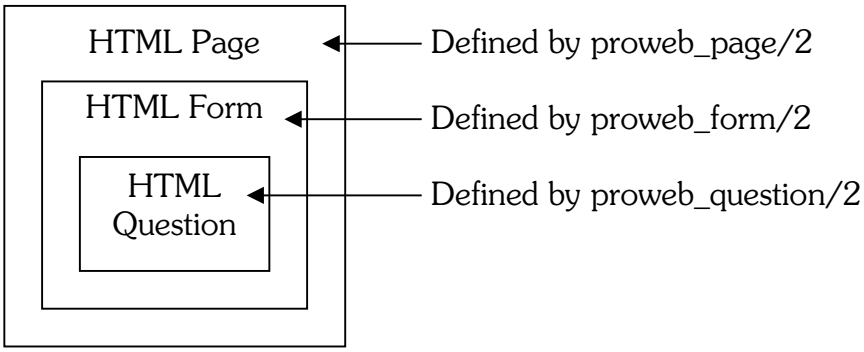
Let's look at this program in detail: In order for the above program to be executed, ProWeb needs to know its 'entry point'. Every ProWeb-based application has an 'entry point' or 'main goal'; a predicate that is executed each time an interaction occurs. In the supplied setup, the 'entry point' has been defined as the predicate *main_goal/0*.

The first line within *main_goal/0* is *proweb_returned_input(my_name, MyName);* this causes ProWeb to pick up the value of *my_name* from the URL line and assign it to the variable *MyName*.

The second line within *main_goal/0* is *proweb_post_reply(your_name,MyName)*; this stores the value of *MyName* so that it can be picked-up later when the output page is generated.

The third line within *main_goal/0* is *proweb_send_form(output_form)*; this is an instruction to ProWeb to wrap *output_form* up into a complete HTML page and send it to the client's browser. Before ProWeb can send *output_form* to the client's browser, it needs to assemble the required HTML code making up the page.

ProWeb is a form-based system. Proweb inserts one or more questions into a form and inserts one or more forms into a page.



ProWeb searches the program for a definition of *output_form*. Within a ProWeb-based application, there would be a user-defined *proweb_form/2* clause for every form, the first argument of which is the name of the form and the second, its definition.

The line, *proweb_form(output_form,include('OUTPUT.HTM')),* contains the definition for *output_form* – it states that *output_form* simply contains *Hello <PROWEB REPLY="your_name">!*. When ProWeb encounters the clause, *proweb_form(output_form, include('OUTPUT.HTM'))*, the *include/1* statement instructs ProWeb to extract the HTML Form part out of OUTPUT.HTM. So despite OUTPUT.HTM containing a complete HTML page, ProWeb is only interested in the contents of the HTML Form itself (i.e. everything between the <FORM> tags):

```
Hello<PROWEB REPLY="your_name">!
```

In fact, we could have written OUTPUT.HTM as such with the same result!

When ProWeb generates *output_form,* the *<PROWEB REPLY="your_name">* tag will get replaced with the name you entered on the input page.

ProWeb now has an HTML form containing the phrase, *Hello <the_name_you_entered>!*, but no HTML page in which to embed it. It then searches the code for a *proweb_page/2* clause which would provide a definition of the HTML page to contain the HTML form; the first argument is the name of the form the clause applies to and the second, the definition of the page. As it fails to find one, the default *proweb_page/2* clause is used.

For every ProWeb predicate that our program uses, we need to declare it as being defined in more than one file, hence the *multifile/1* inline command.

Making The Input Page ProWeb-Generated

In the above example, INPUT.HTM is working as a ProWeb launch page (just like INDEX.HTM) with only the output page being generated by ProWeb. What we will now do is utilise the supplied ProWeb launch page (INDEX.HTM) and have both the input and output pages generated by ProWeb:

```
:- multifile proweb_form/2.
```

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_input( my_name, MyName ),
  proweb_post_reply( your_name, MyName ),
  proweb_send_form( output_form ).

proweb_form( input_form, include('INPUT.HTM') ).

proweb_form( output_form, include('OUTPUT.HTM')).
```

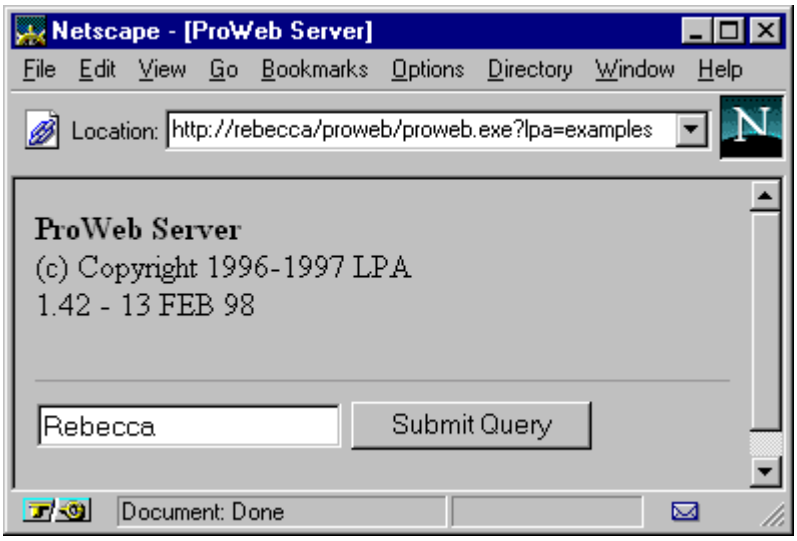
Extracting The Form

When ProWeb encounters the clause, *proweb_form(input_form, include('INPUT.HTM'))*, the *include/1* statement instructs ProWeb to extract the HTML Form part out of INPUT.HTM. So despite INPUT.HTM containing a complete HTML page, ProWeb is only interested in the contents of the HTML Form itself (i.e. everything between the <FORM> tags):

```
<INPUT TYPE="text" NAME="my_name">
<INPUT TYPE="submit">
```

In fact, we could have written INPUT.HTM as such with the same result!

Load the ProWeb launch page (e.g. INDEX.HTM) into your web browser and submit it. The following page should appear ('Rebecca' was entered afterwards):



It looks very similar to the standalone version of INPUT.HTM except that this one was generated by ProWeb. Entering a name and submitting the page will result in the same output page (i.e. 'Hello Rebecca!') as above.

Because the rest of INPUT.HTM outside of the <FORM> tags is discarded by the *proweb_form(input_form, include('INPUT.HTM'))* clause, ProWeb does not have any idea what the page the form is embedded into looks like; as a result, it uses its default page.

ProWeb automatically surrounds every form it generates with <FORM> tags, complete with METHOD=POST and ACTION="/ProWeb/proweb.exe" attributes.

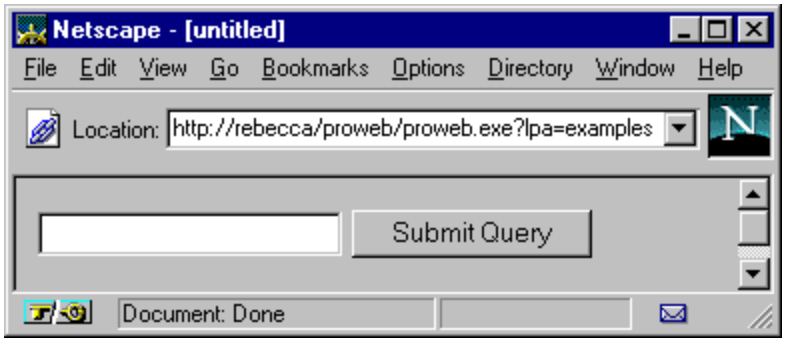
The page which will surround the form can be defined within a *proweb_page/2* clause; for now, we can just add a dummy one to our ProWeb program:

```
:- multifile proweb_page/2.
:- multifile proweb_form/2.

main_goal :-
 proweb_send_form( input_form ),
 proweb_returned_input( my_name, MyName ),
 proweb_post_reply( your_name, MyName ),
 proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, include('INPUT.HTM') ).

proweb_form( output_form, include('OUTPUT.HTM') ).
```

The *proweb_page(_, [])* clause states that it should, because of the underscore, be used by all forms and its definition for the HTML page, the empty list, is absolutely nothing! As a result, ProWeb still has to resort to using default values when generating the page. In the absence of a definition for the HTML page that the form is to be embedded into, ProWeb generates the minimalist amount of HTML code required:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY>
  ...
 </BODY>
</HTML>
```

The presence of this *proweb_page/2* clause does, however, cause both the input and output page to be visually enhanced:



The source code of this page is:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[416]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<INPUT TYPE=TEXT NAME="my_name">
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

which is pretty much what we started out with, plus a little ProWeb "housekeeping".

Adding a Page Definition as an HTML File

Like a form, the page can also be defined in an external HTML file. Enter the following into a text editor or create via an HTML editor and save as the file, PAGE.HTM, in the C:\INETPUB\PROWEB directory:

```
<HTML>
<HEAD>
<TITLE>ProWeb Hello World! Example</TITLE>
</HEAD>
<BODY BGCOLOR="yellow">
<H2>ProWeb Hello World! Example</H2>
<HR>
<FORM>
<PROWEB FORM>
</FORM>
</BODY>
</HTML>
```

In order for ProWeb to use PAGE.HTM, we need to modify the *proweb_page*/2 clause:

```
proweb_page( _, include('PAGE.HTM') ).
```

When the program is run again, the input form will look like the following:

The <PROWEB FORM> tag in the page definition instructs ProWeb to insert the form, be it *input_form* or *output_form*, at this point in the HTML code. The source code of this page is:

```
<HTML>
 <HEAD>
  <TITLE>ProWeb Hello World! Example</TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <H2>ProWeb Hello World! Example</H2>
  <HR>
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[419]">
   <INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
   <PROWEB FORM>
   <INPUT TYPE=TEXT NAME="my_name">
   <INPUT TYPE=SUBMIT>
  </FORM>
 </BODY>
</HTML>
```

You will notice that ProWeb has expanded the opening <FORM> tag to include the METHOD and ACTION attributes.

You will also notice that the form itself is confined to the lower part of the BODY due to the inclusion of the <FORM> tags within PAGE.HTM; i.e. the code, <H2>ProWeb Hello World! Example</H2><HR>, is not part of the form.

Chapter 7 - First Steps For The Prolog Programmer

This chapter guides you through the creation of a "Hello World!" example from the point of view of a Prolog programmer.

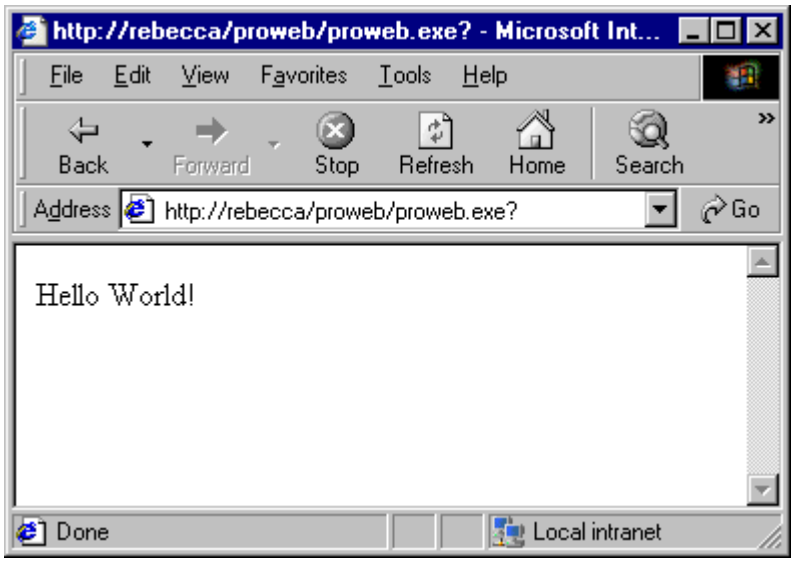
Introduction

Most computer books start with the infamous 'Hello World!' example, so we will follow suit. This chapter goes through a series of eight "Hello World" examples:

Example	Description	Filename
First	This example simply returns a page containing the text "Hello World!".	PW_HW1.PL
Second	This example builds on the first example and introduces an input field for you to enter your name and a submit button for you to submit the form.	PW_HW2.PL
Third	This example builds on the second example by introducing a second page for output; in this example, it outputs the text "Hello World!".	PW_HW3.PL
Fourth	This example builds on the third example by modifying the second page to output the name entered on the first page.	PW_HW4.PL
Fifth	This example provides an alternative way of writing the fourth example by using ProWeb's built-in information storage mechanism.	PW_HW5.PL

1: “Simple Hello World!” Example

By clicking on the "Run ProWeb Application" button of the ProWeb 'launch page', the following HTML page appears in the browser:



Rather than get tied up with detail at this stage of what PROWEB.EXE actually is and what it does, let's initially look at the ProWeb-based application that it runs. What has happens is that ProWeb (for now, think of this as PROWEB.EXE) has run the following ProWeb-based application (PW_HW1.PL copied to PW_DEMO.PL):

```
:- multifile proweb_page/2.
:- multifile proweb_form/2.

main_goal :-
  proweb_send_form( hello_form ).

proweb_page( hello_form, [] ).

proweb_form( hello_form, 'Hello World!' ).
```

Let's look at this program in detail: In order for the above program to be executed, ProWeb needs to know its 'entry point'. Every ProWeb-based application has an 'entry point' or 'main goal'; a predicate that is executed each time an interaction occurs. In the supplied setup, the 'entry point' has been defined as the predicate *main_goal/0*.

The only line within *main_goal/0* is *proweb_send_form(hello_form)*. The line, *proweb_send_form(hello_form),* is an instruction to ProWeb to wrap *hello_form* up into a complete HTML page and send it to the client's browser.

Before ProWeb can send *hello_form* to the client's browser, it needs to assemble the required HTML code making up the page.

ProWeb searches the program for a definition of *hello_form*. Within a ProWeb-based application, there would be a user-defined *proweb_form/2* clause for every form, the first argument of which is the name of the form and the second, its definition. The line, *proweb_form(hello_form, 'Hello World!'),* contains the definition for the *hello_form* form – it states that *hello_form* simply contains the phrase, 'Hello World!'.

ProWeb now has an HTML form containing the phrase "Hello World!" but no HTML page in which to embed it. It then searches the code for a *proweb_page/2* clause which

would provide a definition of the HTML page to contain the HTML form; the first argument is the name of the form the clause applies to and the second, the definition of the page. The line ProWeb finds, *proweb_page(hello_form, [])*, states that the page contains nothing, so ProWeb has to use certain default values to construct the page. ProWeb then returns an HTML page to the client and terminates.

For every ProWeb predicate that our program uses, we need to declare it as being defined in more than one file, hence the two *multifile/1* inline commands.

The HTML source code of the page returned to the client is:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY>
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   Hello World!
  </FORM>
 </BODY>
</HTML>
```

Let's look at this page in detail: ProWeb starts with the phrase 'Hello World!', this being the contents of the form as defined in the program. ProWeb wraps this up into an HTML form:

```
<FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
 Hello World!
</FORM>
```

You will notice that this form possesses a METHOD=POST attribute and an ACTION="/ProWeb/proweb.exe" attribute. ProWeb generates these attributes automatically for every form it produces so that the form can be submitted back to ProWeb. This form can't be submitted, however, because it does not have a submit button.

In the absence of any definition for the HTML page 'container' that the form is to be embedded into, ProWeb generates the minimalist amount of HTML code required:

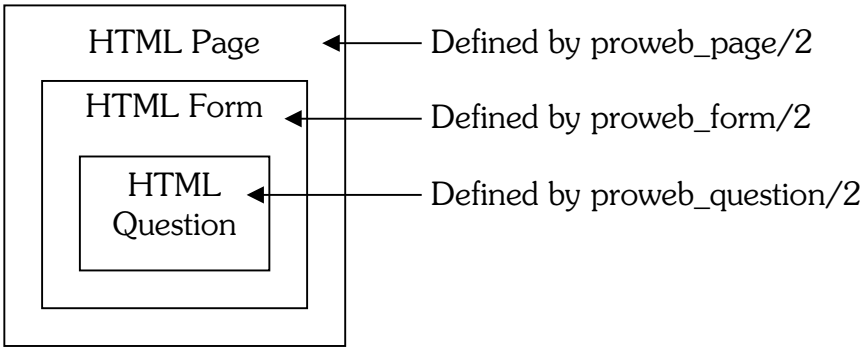
```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY>
  ...
 </BODY>
</HTML>
```

2 : “Enter Name and Submit” Example

We are now going to add an input field into our form (see PW_HW2.PL):

```
proweb_form( input_form, [?name, input(type=submit)] ).

proweb_question( name,
                 [ method = input,
                   type   = string
                 ]
               ).
```

The '?name' is ProWeb's notation for insert the question 'name' at this point in the form. The definition for the question itself is contained in the *proweb_question/2* clause; the first argument is the question name and the second, the question's definition. The ProWeb notation, 'input(type=submit)', states that a submit button is to be placed in the form following the *name* question.



3 : "Simple Output Page" Example

Now that we are able to submit a page, let's now modify our Hello World! example (see PW_HW3.PL) so that our name can be entered on one page and output on the next. For now though, we'll just turn our example into a multi-page one.

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_send_form( hello_form ).

proweb_page( _, [] ).

proweb_form( input_form, [?name,input(type=submit)] ).

proweb_form( hello_form, 'Hello World!' ).

proweb_question( name,
                 [ method = input,
                   type   = string
                 ]
               ).
```

The *input_form* is sent to the client. The client enters their name and submits the form back to ProWeb. ProWeb then sends *hello_form* back to the client.

4 : “Name Echoed on Output Page” Example

In the third example, the output page simply contained the text "Hello World!" regardless of the name entered on the first page. Let's now modify the second page so that the name entered on the first page can be output (see PW_HW4.PL).

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( name, Name ),
  proweb_send_form( hello_form(Name) ).

proweb_page( _, [] ).

proweb_form( input_form,
             [ ?name,
               input(type=submit)
             ]
           ).

proweb_form( hello_form(Name),
             [ 'Hello ',
               Name,
               '!'
             ]
           ).

proweb_question( name,
                 [ method = input,
                   type   = string
                 ]
               ).
```

The line, *proweb_returned_answer(name, Name)*, will only succeed if the ProWeb question, *name*, has been returned; this will only be true if the first page has been submitted back to ProWeb. Having got a value for *name*, the program now passes it as a parameter - *proweb_send_form(hello_form(Name))*; the value of *name* is then embedded into *hello_form* - "Hello <name>!".

5 : “Storing The Name Between Pages” Example

This fifth "Hello World!" example presents an alternative approach to storing and returning a question's value.

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( name, Name ),
  proweb_post_reply( name, Name ),
  proweb_send_form( hello_form ).
```

```
proweb_form( hello_form,
             [ 'Hello ',
               ??name,
               '!'
             ]
           ).
```

Rather than pass *name* as a parameter, we can store it in Prolog's internal memory using *proweb_post_reply(name, Name)*. Having stored *name*, it can then be picked-up within the definition of *hello_form* using the ProWeb notation, *??name*.

Chapter 8 - First Steps - Trouble-Shooting

This chapter gives explanations and solutions to any problems you may experienced in your first few steps with ProWeb.

Problem Encountered	Explanation and Solution
Netscape Navigator keeps telling me it can not find the ProWeb 'launch page'.	Check that the file extension is correct – the file *.htm can not be referred to as *.html or vice versa.
Whenever I click on a button within the ProWeb 'launch page', I get a "Netscape is unable to find the file or directory named: /proweb/proweb.exe Check the name and try again" error message appear.	You have loaded the page locally, perhaps by double clicking on the INDEX.HTM file directly. Load the file as http://<server_name>/proweb/index.htm rather than as file:///c|/inetpub/proweb/index.htm.
Nothing happens when I click on the submit button of the ProWeb 'launch page'.	Check that your ProWeb 'launch page' is pointing at PROWEB.EXE in the correct directory. Ensure that PROWEB.EXE and PROWEB.SYS are being run on the server machine; have the Task Manager visible whilst submitting the page. Ensure that the correct version of PROWEB.EXE is being used. If you have just created a new overlay, ensure that PROWEB.SYS is the same version of ***WIN-PROLOG*** as used to create the overlay file.
Whenever I click on the submit button of the ProWeb 'launch page', I get a "System Error (WEX:001:2)" error message page.	Ensure that you have copied the ***WIN-PROLOG*** executable file, PRO386W.EXE, to your ProWeb root directory (i.e. C:\INETPUB\PROWEB) and renamed it PROWEB.SYS. Ensure that it has only one extension and not PROWEB.SYS.EXE or some such.
After submitting the ProWeb 'launch page', I get the ***WIN-PROLOG*** error message "Main program terminated with an error! Error #20 Predicate Not Defined main_goal/0"	Ensure that the ProWeb setting, *main goal*, in the PROWEB.INI file is set as intended; you can point it to any zero arity predicate you wish to have as your application's main goal.
I am using Personal Web Server and Microsoft Internet Explorer together. I have to	Try using Netscape Navigator instead of Internet Explorer.

wait for ***WIN-PROLOG*** to timeout before the HTML page is completely drawn within the web browser window.	
ProWeb appears to be frozen, but I am unable to end it via the Task Manager.	Execute the URL line: http://<server_name>/proweb/proweb.exe?!+001 from your browser, where <server_name> is your own server name. Set the ProWeb *idle timeout* setting to a lower number of seconds (i.e. 30) to ensure that PROWEB.EXE and PROWEB.SYS time out on their own within a reasonable time; once your ProWeb application is working successfully, this setting can be increased again.

Chapter 9 - Variations on a Theme

In this chapter, we are going to take a very simple ProWeb application idea and write the application up in as many variations as ProWeb allows.

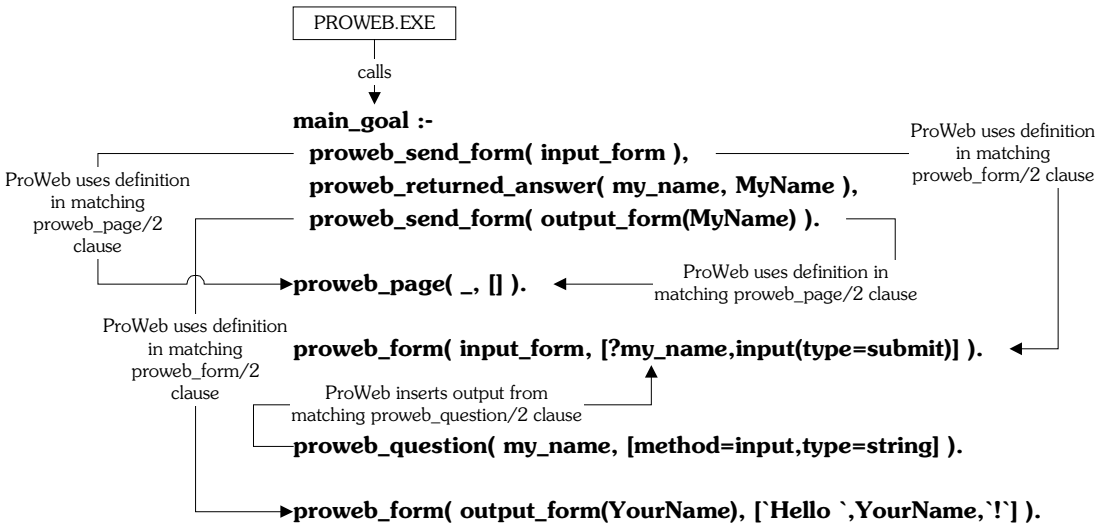
Introduction

All the variations have the same user interface - the first form allows a <name> to be entered and submitted, the second form simply output the phrase, 'Hello <name>!'.

Version One

This version introduces the main ProWeb building blocks - *proweb_page /2*, *proweb_form/2*, *proweb_question/2*, *proweb_send_form/1*, *proweb_returned_answer/2* and the single question mark notation for asking a question:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( my_name, MyName ),
  proweb_send_form( output_form(MyName) ).

proweb_page( _, [] ).

proweb_form( input_form, [?my_name,input(type=submit)] ).

proweb_form( output_form(YourName), [`Hello `,YourName,`!`]).

proweb_question( my_name, [method=input,type=string] ).
```



Version Two

This version moves *proweb_returned_answer/2* to the *proweb_form/2* clause of *output_form* and loses the passing of *MyName/YourName* as a parameter:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, [?my_name,input(type=submit)] ).

proweb_form( output_form, [`Hello `,YourName,`!`]) :-
  proweb_returned_answer( my_name, YourName ).

proweb_question( my_name, [method=input,type=string] ).
```

Version Three

This version introduces *proweb_post_reply/2* for storing a reply and the double question mark notation for inserting a reply posted earlier with *proweb_post_reply/2*:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( my_name, MyName ),
  proweb_post_reply( your_name, MyName ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, [?my_name,input(type=submit)] ).

proweb_form( output_form, [`Hello `,??your_name,`!`]).

proweb_question( my_name, [method=input,type=string] ).
```

Version Four

This version replaces the Prolog term version of the *input_form* with a real HTML version. The <PROWEB QUESTION="..."> element is introduced to replace the single question mark notation:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, include('input.htm') ).

proweb_form( output_form, [`Hello `,YourName,`!`]) :-
  proweb_returned_answer( my_name, YourName ).

proweb_question( my_name, [method=input,type=string] ).
```

The code within INPUT.HTM being:

```
<PROWEB QUESTION="my_name"><INPUT TYPE="SUBMIT">
```

Version Five

This version replaces the Prolog term version of the *output_form* with a real HTML version. The <PROWEB REPLY="..."> element is introduced to replace the double question mark notation:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( my_name, MyName ),
  proweb_post_reply( your_name, MyName ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, [?my_name,input(type=submit)] ).

proweb_form( output_form, include('output_form.htm') ).

proweb_question( my_name, [method=input,type=string] ).
```

The code within OUTPUT_FORM.HTM being:

```
Hello <PROWEB REPLY="your_name">!
```

Version Six

This version loses the *proweb_question/2* clause, defining the *my_name* question in pure HTML. As a result, the predicate, *proweb_returned_input/2*, has to be used instead of *proweb_returned_answer/2*. The value of *my_name* is asserted using *proweb_assert/1* and picked-up with *proweb_call/1*:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_input( my_name, MyName ),
  proweb_assert( your_name(MyName) ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form, include('input.htm') ).

proweb_form( output_form, [`Hello `,YourName,`!`]) :-
  proweb_call( your_name(YourName) ).
```

The code within INPUT.HTM being:

```
<INPUT TYPE="text" NAME="my_name"><INPUT TYPE="SUBMIT">
```

Chapter 10 - The Network Paths Example

During this chapter, we are going to look in detail at the supplied ProWeb-based network paths example.

The Network Paths Example

Although the two source code files for the Network Paths example have been supplied, you may like to assume they haven't and create them afresh.

This ProWeb-based program will be developed in three stages:

- Develop a simple network paths search program in ***WIN-PROLOG.***

- Develop the core ProWeb front end to the network paths search program.

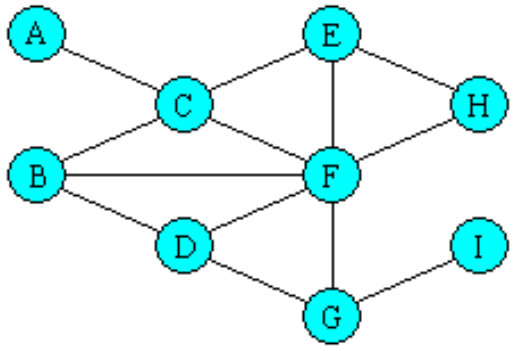
- Interface the core ProWeb front end to the network paths search program.

Start up ***WIN-PROLOG*** and open up a new edit window via the menu option, File/New. Although the program you are now going to develop is already present on your hard disk as the file, NETPATH.PL, in the C:\INETPUB\PROWEB\EXAMPLES directory, you may like to type in the code as you go, commenting it yourself as you feel necessary and saving it under a different name.

If you prefer, stages one and two can be done in reverse order, although both must be completed before stage three can be done.

Stage 1 - Develop a Network Paths Search Program in *WIN-PROLOG*

As Prolog is an ideal language for searching a network to give all the paths (via backtracking) from one node to another, this seems like a good basis for a ProWeb-based example.



So as not to dive immediately into developing a ProWeb-based program, let's get something running in 'pure' Prolog first. Enter the following program into your application's source file window (the one titled 'Untitled'):

```
search( StartNode, EndNode, Path ) :-
        search( StartNode, EndNode, [StartNode], Path ).
```

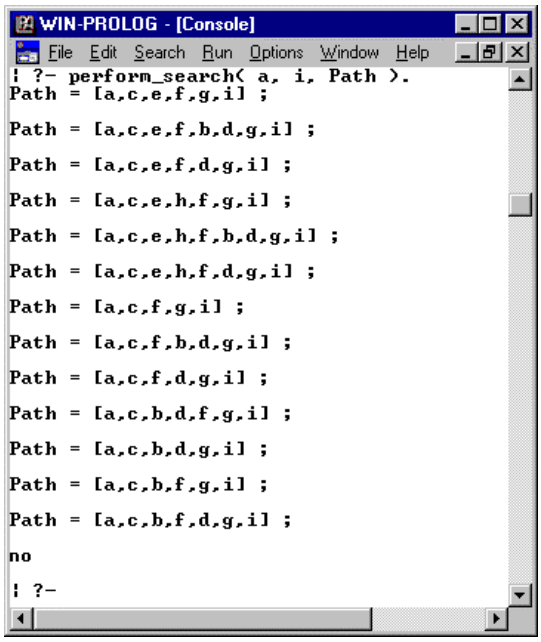
```
search( Node, Node, _, [Node] ).

search( ThisNode, EndNode, NodesVisited, [ThisNode|Path] ) :-
        linked( ThisNode, NextNode ),
        \+( member(NextNode,NodesVisited) ),
        search( NextNode, EndNode, [NextNode|NodesVisited], Path ).

linked( X, Y ) :- arc( X, Y ); arc( Y, X ).

arc( a, c ).
arc( b, c ).
arc( b, d ).
arc( b, f ).
arc( c, e ).
arc( c, f ).
arc( d, f ).
arc( d, g ).
arc( e, f ).
arc( e, h ).
arc( f, g ).
arc( f, h ).
arc( g, i ).
```

When you have finished entering the above, compile the code via the Run/Compile menu option and save under the filename NETPATH.PL in your ProWeb home directory's Examples folder.

The above code, as you can appreciate, is a self-contained Prolog program. When developing an application, of which a ProWeb-based one is no exception, it is always good practice to be able to test each part in isolation. Execute the following from the *WIN-PROLOG* command line:

```
?- search( a, i, Path ). <enter>
```

Its output will be as follows:



The program has returned each possible path through the network as a list of chars; try it for yourself using different start and end nodes. In stages 2 and 3, this will be expanded upon so that the same output can be received via a web browser.

Stage 2 – Develop the Core ProWeb Front End

The search program developed in stage one can be set aside for the time being as it is not required for this stage.

Although the program you are now going to develop is already present on your hard disk as the file, PW_NTPTH.PL, in the C:\INETPUB\PROWEB\EXAMPLES directory, you may like to type in the code as you go, commenting it yourself as you feel necessary and saving it under a different name. If you name your file, PW_DEMO.PL, you will be able to run it 'live' as you work through this chapter.

Creating the Node Selection Form

The first HTML page of this Network Paths program will need to ask the client what values the start and end nodes have. The first task with any page is to define its overall layout and the form you wish to embed within it. The form you are going to use for the Node Selection Form is the one defined within the file, TMPLATE1.HTM, in C:\INETPUB\PROWEB\HTML\NETPATHS.

The text of TMPLATE1.HTM is as follows:

```
<TABLE>
   <TD></TD>
   <TD>
      <H2>Network Paths Node Selection Form</H2>
      <IMG SRC="network.gif">
      <H3>Please select start and end nodes</H3>
```

```
    </TD>
</TABLE>
```

Additional HTML code could have been added to this file to make it loadable into a web browser; such code would be ignored by ProWeb as it is only interested in the contents of the form.

A user-defined *proweb_form/2* clause having the name of the form as its first argument must be present before ProWeb can create the Node Selection Form. In fact, such a user-defined clause will be required for every form you ask ProWeb to create. Enter the following into your source code file:

```
proweb_form( Form, [include(File),Footer] ) :-
  form_title_file( Form, _, File ),
  form_footer( Form, Footer ).

form_title_file( node_selection_form, `Node Selection Form`,
'html/netpaths/tmplate1.htm' ).
```

Although the format of the *proweb_form/2* clause above is a little complex it does allow you to easily define such a clause for subsequent forms simply by adding the necessary *form_title_file/2* and *form_footer/2* clauses. The *form_footer/2* clause for the *node_selection_form* will be discussed shortly.

The term, *include(File)*, is an instruction to ProWeb to read *File* and embed its contents at this point in the form; in the case of a form, only the HTML code within the <FORM> element (if present) is read in. You can have more than one *include/1* term within the same clause.

In addition to specifying the form to be embedded within an HTML page, the attributes of the page itself can also be specified via a user-defined *proweb_page/2* clause.

Enter the following into your program's source file:

```
proweb_page( [ Form ],
          [ include('html/netpaths/head.htm'),
            Title,
            include('html/netpaths/body.htm'),
            proweb(Form),
            include('html/netpaths/foot.htm')
          ]
        ) :- form_title_file( Form, Title, _ ).
```

The file HEAD.HTM contains the opening HTML tag and the beginning of the HEAD section:

```
<HTML>
   <HEAD>
      <TITLE>
```

The file BODY.HTM contains the end of the HEAD section and the beginning of the BODY section:

```
    </TITLE>
</HEAD>
<BODY BGCOLOR="white">
```

The file FOOT.HTM contains the end of the BODY section and the closing HTML tag:

```
    </BODY>
</HTML>
```

When an HTML page is put together by ProWeb, the value of the variable *Title* will be inserted between the contents of HEAD.HTM and BODY.HTM. The form itself will be inserted between the contents of BODY.HTM and FOOT.HTM.

At first glance, *include/1* above may appear to be an HTML element; it is actually a special Prolog term that, when encountered by ProWeb, causes ProWeb to insert, at that point in the page it is constructing, the text contained in the given HTML file. The *include/1* term simply sets the current input stream to the HTML file given, reads in the file and then sets the current input stream back to what it was before.

Like the *proweb_form/2* clause above, this *proweb_page/2* clause is generic and will be used for all HTML pages created for this Network Paths example. Note that in the absence of an appropriate user-defined *proweb_page/2* clause, ProWeb will use its default *proweb_page/2* clause; this features the ProWeb banner.

The Node Selection Form will also require a unique footer as defined below:

```
form_footer( node_selection_form,
        [       table('WIDTH=100%'),
                tr,
                td('WIDTH=50%',align=center),
                input(type=submit, value='Get Solutions'),
                /td,
                td('WIDTH=50%',align=center),
                input(type=reset),
                /td,
                /tr,
                /table
        ] ).
```

The above *form_footer/2* clause introduces ProWeb's notation for defining HTML elements and attributes in Prolog as opposed to including an HTML template file. Such notation can be placed in *proweb_page/2* and *proweb_form/2* clauses and any clauses called by them, such as the above *form_footer/2*. The notation should be self-explanatory when you compare it with the HTML code of the generated HTML page.

A reset button has a preset function of resetting all the data in the form to its original value; just like for a submit button, we could have included a VALUE element to change the text displayed in its button.

When TMPLATE1.HTM is utilised by ProWeb, it will be treated as a form. Unlike, say, the ProWeb 'launch page', the node selection form does not become a permanent HTML file on your hard disk; it is created 'on the fly' by ProWeb as and when required. If you were to view its HTML source code, it would be as follows:

```
<HTML>
   <HEAD><TITLE></TITLE></HEAD>
   <BODY BGCOLOR="white">
      <FORM METHOD=POST>
         <TABLE>
            <TD></TD>
            <TD>
             <H2>Network Paths Node Selection Form</H2>
             <IMG SRC="images/network.gif">
             <H3>Please select start and end nodes</H3>
            </TD>
         </TABLE>
         <TABLE WIDTH=100%>
            <TR>
             <TD WIDTH=50% ALIGN=CENTER>
                <INPUT TYPE=SUBMIT VALUE="Get Solutions">
             </TD>
             <TD WIDTH=50% ALIGN=CENTER>
                <INPUT TYPE=RESET>
             </TD>
            </TR>
         </TABLE>
      </FORM>
   </BODY>
</HTML>
```

Creating the Solutions Form

You are now going to define your Solutions Form, the page that will return all the solutions found by the search program developed in stage one back to the client. Enter the following clauses into your program's source code file (ensuring that predicates of the same name and arity are kept together):

```
form_footer( solutions_form,
        [       table('WIDTH=100%'),
                tr,
                td('WIDTH=50%',align=center),
                input(type=submit, value=`Get Next Solution`),
                /td,
                td('WIDTH=50%',align=center),
                include('html/netpaths/link_hp.htm'),
                /td,
                /tr,
                table
        ] ).
```

form_title_file(solutions_form, `Solutions Form`, 'html/netpaths/tmplate2.htm').

Just like for the Node Selection Form, another template HTML file, TMPLATE2.HTM, is being utilised, the content of which is:

```
<HTML>
   <HEAD><TITLE></TITLE></HEAD>
   <BODY>
      <FORM>
         <TABLE>
            <TD>
             <H2>Network Paths Solutions Form</H2>
             <IMG SRC="network.gif">
             <H3>
                 Solution <PROWEB VALUE="%2"> is
                     <PROWEB REPLY="solution">
             </H3>
            </TD>
         </TABLE>
      </FORM>
   </BODY>
</HTML>
```

Like the Node Selection Form, your Solutions Form will be created 'on the fly' by ProWeb as and when required during a conversation. As the *include/1* term is within a *proweb_form/2* clause, only the HTML code between the <FORM> and </FORM> tags will be utilised by ProWeb.

Defining the 'Main Goal' Predicate

When you click the submit button on your ProWeb 'launch page', ProWeb will look for a user-defined *main_goal/0* clause to execute. The user-defined *main_goal/0* clause, when fully-developed, is going to be quite complex; rather than give you all the source code now, we are going to build it up in stages. Enter the following into your source code and save under the filename PW_NTPTH.PL (or PW_DEMO.PL if you prefer) in your ProWeb Examples directory:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        proweb_returned_form( node_selection_form ),
        proweb_send_form( solutions_form ).
```

Running the Network Paths Program for the First Time

To run the ProWeb-based network paths program for the first time, follow these steps:

- Run your chosen web browser (e.g. Netscape Navigator or Internet Explorer).

- Load your ProWeb lauch page (i.e. http://<server_name>/proweb/index.htm) into your chosen web broswer.

- Click on this page's 'Run ProWeb Application' submit button.

- Your web browser will pass the URL line to PROWEB.EXE.

- *WIN-PROLOG*/ProWeb will execute your *main_goal/0* clause.

- Upon executing the first *proweb_send_form/1* call, your Node Selection Form will be created 'on the fly' and returned to the client. The Node Selection Form will be displayed in the client's browser.

- Upon sending an HTML page to the client, your application will terminate.

- Click on the Node Selection Form's 'Get Solutions' button.

- Your web browser will pass the URL line to PROWEB.EXE.

- *WIN-PROLOG*/ProWeb will execute your *main_goal/0* clause from the beginning again.

- The first *proweb_send_form/1* call, although succeeding, will not be acted upon as the Node Selection Form has already been sent to the client. ProWeb knows it has already sent this form because it asserted into the internal Prolog database, when the form was sent, a fact of the form *proweb_data_form_sent(1, null_context, node_selection_form)*.

- The next line, *proweb_returned_form(node_selection_form)*, will succeed as the Node Selection Form has indeed been returned by the client.

- Upon executing the next line, *proweb_send_form(solutions_form)*, your Solutions Form will be created 'on the fly' and returned to the client. The Solutions Form will be displayed in the client's browser. The strange output text, 'Solution %2 is ?? solution' is due to the fact that ProWeb is unable to substitute the <PROWEB VALUE="%2"> and <PROWEB REPLY="solution"> elements present in the TMPLATE2.HTM file because the relevant Prolog source code has yet to be written; such code will be added later on in this chapter.

- Upon sending an HTML page to the client, your application will terminate once again.

- Click on the Solutions Form's 'Get Next Solution' button.

- Your web browser will pass the form's data to PROWEB.EXE.

- ProWeb will execute your *main_goal/0* clause from the beginning once again.

- The first two lines, *proweb_send_form/1* and *proweb_returned_form(node_selection_form)* will again succeed but not acted upon.

- The second *proweb_send_form/1* call will succeed, but like the previous *proweb_send_form/1* call, it will not be acted upon as the Solutions Form has already been sent to the client.

- The *main_goal/0* clause has succeeded but ProWeb does not have a user-defined form to return to the client; it therefore creates its own 'Main program terminated with success!' page and sends this to the client's browser.

- Upon sending an HTML page to the client, your application terminates again.

Stage 3 - Interfacing: ProWeb Front End to Search Program

Having developed a simple search program in stage one and the core ProWeb front end in stage two, you are now going to take the source code generated from both these stages and join them together.

Ensuring the Search Program's Source Code is Loaded

Firstly, we need to ensure that the source code for the Prolog network paths search program (in the file NETPATH.PL) is automatically loaded along with the ProWeb front end; enter the following directive at the head of the source code for PW_NTPTH.PL (or PW_DEMO.PL):

```
:- ensure_loaded( examples(netpath) ).
```

Adding the Node Entry Fields to the Node Selection Form

In the simple search program, the start and end nodes were entered as the first and second arguments of *search/3*. For this web-based approach, these arguments will each receive their value from a higher level, that of selecting a node from a pull-down menu field in an HTML form.

At present, the Node Selection Form asks you to select your start and end nodes, but then gives you no opportunity to actually do so. We are now going to add some code to allow the start and end nodes to be selected. First of all we need to insert a couple of <PROWEB QUESTION...> elements into TMPLATE1.HTM, the template HTML file:

```
...
<H3>Please select start <PROWEB QUESTION="start_node"> and end
<PROWEB QUESTION="end_node"> nodes</H3>
...
```

Should you not like this format, feel free to rewrite or rearrange the text around the two <PROWEB QUESTION...> elements. As this template HTML file is used by ProWeb to create another page 'on the fly', these <PROWEB QUESTION...> elements will never be seen by a web browser. Should you wish to use an HTML editor, there is no problem in having such elements inserted in your file. If you were to load TMPLATE1.HTM into a web browser, the two <PROWEB QUESTION...> elements, because they have no meaning, would simply be ignored; nothing will appear on the screen although the elements would be present should you ask for the HTML source code to be displayed.

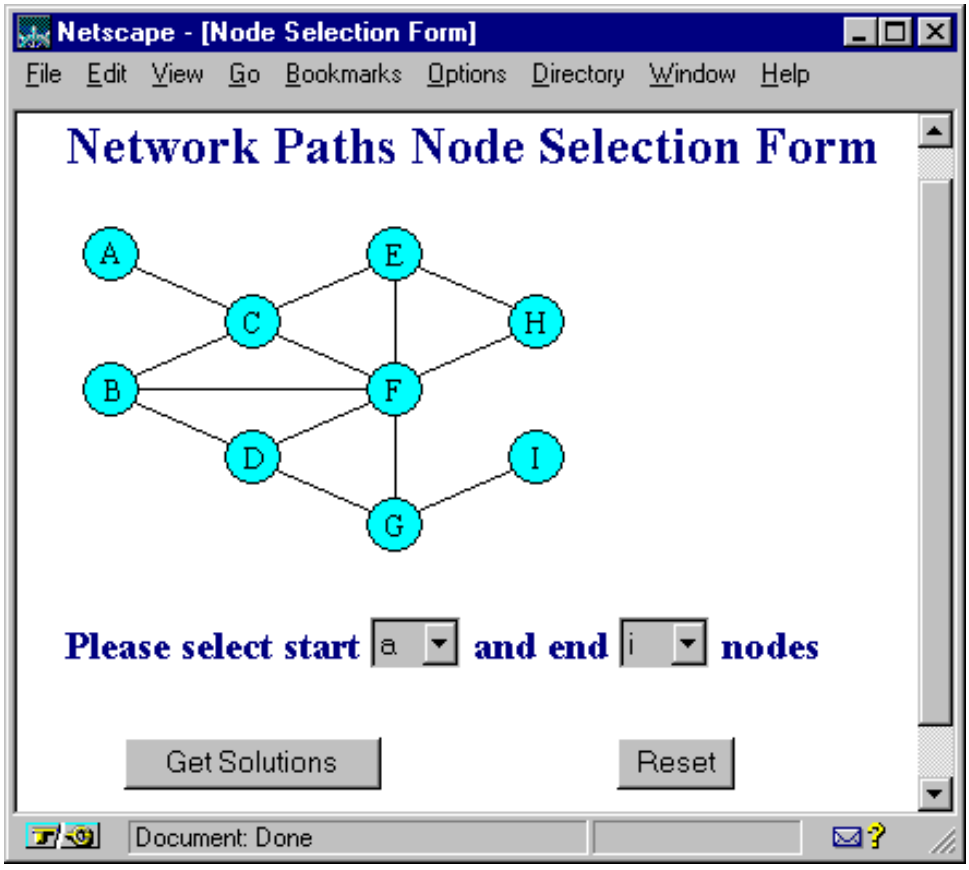
We next need to define, within our Prolog source code, a corresponding *proweb_question/2* clause for each:

```
proweb_question(    start_node,
        [       method      = menubox,
                select      = [a,b,c,d,e,f,g,h,i]
        ] ).

proweb_question(    end_node,
        [       method      = menubox,
                select      = [a,b,c,d,e,f,g,h,i],
                prefill     = i
        ] ).
```

The above *proweb_question/2* clauses will simply define two menubox fields with the names *start_node* and *end_node* respectively. The menu associated with each menubox field will possess an option for each node in the network; the start node, due to the absence of a *prefill* attribute, defaults to the first value in the list, namely 'a', whilst the end node takes on the value 'i'.

Re-compile the source code file now and save as required.

When your Network Paths program is next run, your Node Selection Form will look like:



The HTML source code of your revised Node Selection Form will be as follows:

```
<HTML>
   <HEAD><TITLE>Node Selection Form</TITLE></HEAD>
   <BODY BGCOLOR="white">
      <FORM METHOD=POST>
```

```
            <TABLE>
              <TD></TD>
              <TD>
               <IMG SRC="images/network.gif">
               <H2>Network Paths Node Selection Form</H2>
               <H3>Please select start
                   <SELECT NAME="q0001" SIZE=1>
                      <OPTION>a</OPTION>
                      <OPTION>b</OPTION>
                   <OPTION>c</OPTION>
                   <OPTION>d</OPTION>
                      <OPTION>e</OPTION>
                   <OPTION>f</OPTION>
                   <OPTION>g</OPTION>
                   <OPTION>h</OPTION>
                   <OPTION>i</OPTION>
                   </SELECT> and end
                   <SELECT NAME="q0002" SIZE=1>
                      <OPTION>a</OPTION>
                   <OPTION>b</OPTION>
                      <OPTION>c</OPTION>
                      <OPTION>d</OPTION>
                   <OPTION>e</OPTION>
                   <OPTION>f</OPTION>
                      <OPTION>g</OPTION>
                   <OPTION>h</OPTION>
                      <OPTION SELECTED>i</OPTION>
                   </SELECT> nodes
               </H3>
              </TD>
            </TABLE>
            <TABLE WIDTH=100%>
              <TR>
               <TD WIDTH=50% ALIGN=CENTER>
                  <INPUT TYPE=SUBMIT VALUE="Get Solutions">
               </TD>
               <TD WIDTH=50% ALIGN=CENTER>
                  <INPUT TYPE=RESET>
               </TD>
              </TR>
            </TABLE>
         </FORM>
      </BODY>
   </HTML>
```

Expansion of the Main Goal Predicate

For your program to 'pick-up' the start node and end node values from your submitted Node Selection Form, a *proweb_returned_answer/2* call for each needs to be added to *main_goal/0*:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        %  proweb_returned_form( node_selection_form ),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        proweb_send_form( solutions_form ).
```

You can liken *proweb_send_form/1* to asking a question and *proweb_returned_answer/2* to getting its answer. In ***WIN-PROLOG***, the question would be asked whereupon execution of the program would be suspended until an answer was given. With ProWeb, the question would be asked whereupon the application would be terminated. Upon an answer being given (i.e. the page is submitted by the client), the application would be rerun from the beginning, reaching and executing *proweb_returned_answer/2* in due course.

Whenever a client submits your Node Selection Form, the selected start and end nodes will be returned to your application. Each of the *proweb_returned_answer/2* calls will cause ProWeb to assert a *proweb_data_answer/4* fact into Prolog's internal database.

The line *proweb_returned_form(node_selection_form)* is no longer required and can be removed as *proweb_returned_answer(start_node,StartNode)* will do the same job (i.e. fail if the Node Selection Form has yet to be returned and succeed if it has).

You will next need to pass these values to the simple search program in order to compute the n'th solution to the problem of finding all the paths between the StartNode and the EndNode; this is done by adding a line which uses ***WIN-PROLOG***'s built-in *solution/2* predicate:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        solution( search(StartNode,EndNode,Solution ), Nth ),
        proweb_send_form( solutions_form ).
```

As each path through the network is generated, it needs to be stored away, ready to be retrieved by ProWeb as and when it generates the Solutions Form; add the following *proweb_post_reply/2* clause to *main_goal/0*:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        solution( search(StartNode,EndNode,Solution ), Nth ),
        proweb_post_reply( solution, verbatim @ Solution ),
        proweb_send_form( solutions_form ).
```

When storing a value via the *proweb_post_reply/2* predicate, care must be taken to store the value in the correct format; the *proweb_post_reply(solution, verbatim @ Solution)* line used above needs a little explanation. The value of Solution, as returned

from *search(StartNode,EndNode,Solution)*, is a list of the form [a,c,e,f,g,i]. Executing a line such as *proweb_post_reply(solution, Solution)* would cause ProWeb to assert into Prolog's internal database the fact *proweb_data_reply(solution, [a,c,e,f,g,i])*. You may well assume this to be correct, but when the time comes for ProWeb to pick this up and embed it into your HTML page in place of the *<PROWEB REPLY="solution">* element, it will end up as *<A>cefg<I>* due to encoding; the *a* and *i* nodes will be interpreted as HTML *<A>* and *<I>* tags respectively and the solution itself will display as *cefg*. By using the line *proweb_post_reply(solution, verbatim @ Solution)*, you cause ProWeb to assert into Prolog's internal database the fact *proweb_data_reply(solution, verbatim @ [a,c,e,f,g,i])* – when 'picked-up', it is embedded verbatim into the HTML page, displaying as *[a,c,e,f,g,i]* on your client's web browser.

Further Amendments to the Solutions Form

As stated earlier, *proweb_send_form(solutions_form)* will only allow *solutions_form* to be sent to a client once per conversation, resulting in only the first solution being returned. Although altering this line to *proweb_resend_form(solutions_form)* would allow further Solutions Forms to be sent, it is of no help as each one would still contain the first solution!

A solution is probably to add an argument to solutions_form; this will allow separate instances of the Solutions Form to be created, each with its own unique number. A good candidate for this unique number is *Nth*, the number of the solution itself. Modify your Solutions Form's *form_title_file/3* and *form_footer/2* clauses as follows:

```
form_title_file( solutions_form(_), `Solutions Form`, 'tmplate2.htm' ).

form_footer( solutions_form(_),
…
```

Further Expansion of the Main Goal Predicate

In order to generate the second and subsequent solutions, we need a way to force backtracking whenever we encounter a solution that has already been sent. The way to achieve this is to place the line *proweb_post_unique_reply(solution_nth, Nth)* directly after *solution(search(StartNode,EndNode,Solution), Nth)*; this will fail on each previously sent solution, the program therefore backtracking until a brand new solution is found.

The revised *main_goal/0* is as follows:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        solution( search(StartNode,EndNode,Solution ), Nth ),
        proweb_post_unique_reply( solution_nth, Nth ),
        proweb_post_reply( solution, verbatim @ Solution ),
        proweb_send_form( solutions_form(Nth) ).

main_goal.
```

In the revised code above, you will see two additional changes. The first is that the *Nth* argument has been added to *solutions_form*. The second is that another *main_goal/0* clause has been created; this is vital in preventing *main_goal/0* from failing. When all solutions have been returned, *solution(search(StartNode, EndNode, Solution), Nth)* itself will fail, thereby causing the second *main_goal/0* clause to be executed and *main_goal/0* as a whole to succeed.

This is but one method which enables ProWeb to support backtracking and multiple solutions. If the *solution/2* and *proweb_post_unique_reply/2* lines succeed, ProWeb puts the new solution into Prolog's internal database and then generates a new instance of the Solutions Form, uniquely identified by its Nth number.

The <PROWEB VALUE=...> Element

In the TMPLATE2.HTM file, if you remember, we had a <PROWEB VALUE="%2"> element. Whenever ProWeb encounters a <PROWEB VALUE="%#"> element, ProWeb replaces the element with the value of the #th-1 argument of the form being generated; the <PROWEB VALUE="%1"> element would match with the form name itself. Up until now, ProWeb has been trying to replace <PROWEB VALUE="%2"> with the Solutions Form's first argument; now that the Solutions Form has an argument, ProWeb will be able to replace this ProWeb element with its intended value.

Compile your source code file, saving as required, and then run your Network Paths example again.

Creating a No More Solutions Form

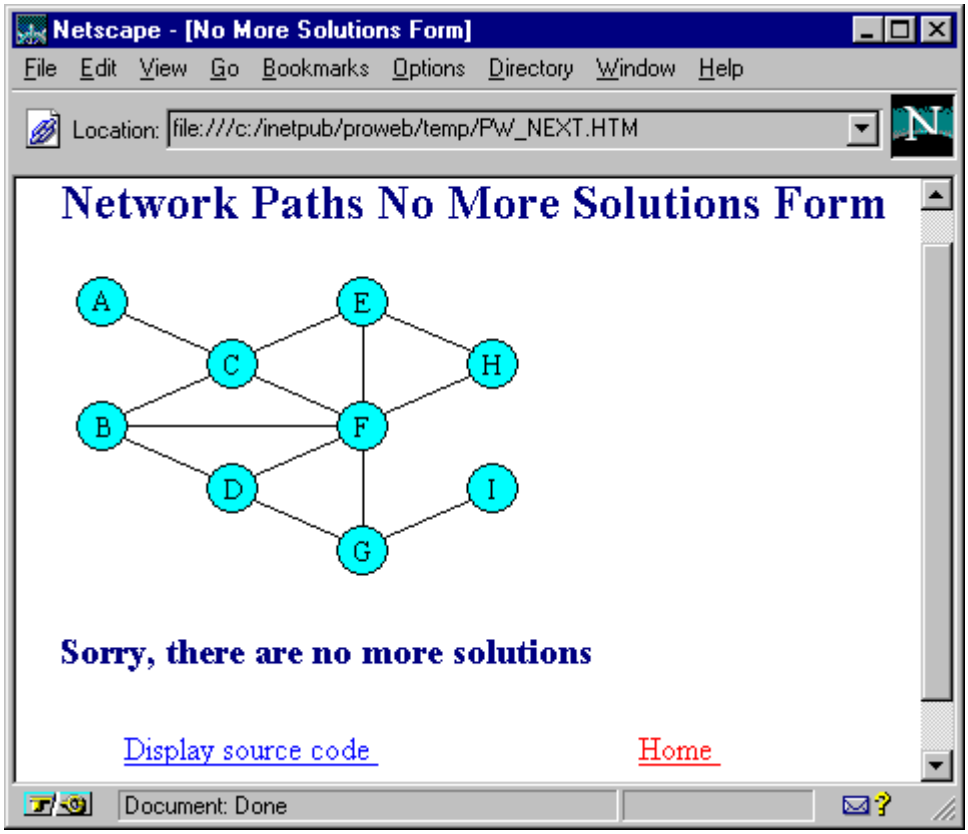
In a real application, ProWeb's 'Main program terminated with success!' page would be replaced with one of your own choosing. Your second *main_goal/0* clause is doing nothing except making *main_goal/0* succeed, so let's ask it to handle a No More Solutions Form, our replacement for ProWeb's 'Main program terminated with success!' page; amend the second *main_goal/0* clause as follows:

```
main_goal :-
        proweb_returned_form( node_selection_form ),
        proweb_send_form( no_more_solutions_form ).
```

Like the other pages you have asked ProWeb to create for this Network Paths example, you are going to need another user-defined *form_title_file/3* and *form_footer/2* clause:

```
form_title_file( no_more_solutions_form,
                 `No More Solutions Form`,
                 'html/netpaths/tmplate3.htm'
               ).

form_footer(  no_more_solutions_form,
                        [       table('WIDTH=100%'),
                                tr,
                                td('WIDTH=50%',align=center),
                                include('html/netpaths/link_sc.htm'),
                                /td,
                                td('WIDTH=50%',align=center),
                                include('html/netpaths/link_hp.htm'),
                                /td,
                                /tr,
                                /table
                        ]
                ).
```

When your application is compiled and run again, ProWeb's 'Main program terminated with success!' page will be replaced by the following:



Dynamic ProWeb Questions

When ProWeb assembles your Node Selection Form, the format used for the start and end node menu boxes is as defined in the two *proweb_question/2* clauses. Both of these clauses include the line *select = [a,b,c,d,e,f,g,h,i]*, which ProWeb translates into

the list of selectable options. If one or more nodes were added to your network you would be unable to select them from your Node Selection Form as, currently, the start and end node fields are not updated dynamically; however, as ProWeb creates your Node Selection Form 'on the fly', it is an easy matter to make them dynamic. The solution is to construct an ordered list 'on the fly' of the current nodes from the *arc/2* database clauses and pass this into each *proweb_question/2* clause. Modify the *proweb_question/2* clause for the *start_node* as shown below:

```
proweb_question(    start_node,
                [       method      = menubox,
                      select        = Nodes
                ]
        ) :-
        setof( Node, A^linked(Node,A), Nodes ).
```

As you will recall, the end node's *proweb_question/2* clause differed from the start node's in that its default value was set to the 'last' node in the network; it is therefore advisable to make this dynamic as well. Modify the end node's *proweb_question/2* clause as follows:

```
proweb_question( end_node,
                [       method      = menubox,
                        select      = Nodes,
                        prefill     = Last
                ]
        ) :-
        setof( Node, A^linked(Node,A), Nodes ),
        length(Nodes,Len),
        member(Last,Nodes,Len).
```

Further Amendments to the Solutions Form

As the solution itself is the most important information on your Solutions Form, it needs to stand out a lot more than it currently does. Instead of a line like 'Solution 1 is [a,c,e,f,g,i]', we are now going to show how to display it in graphical form.

Amend your program's source code as follows (note that the word *verbatim* in the *proweb_post_reply* line has been removed):

```
main_goal :-
        proweb_send_form( node_selection_form),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        solution( search( StartNode, EndNode, Solution ), Nth ),
        proweb_post_unique_reply( solution_nth, Nth ),
        construct_graphical_solution( Solution, GraphicalSolution ),
        proweb_post_reply( solution, GraphicalSolution ),
        proweb_send_form( solutions_form(Nth) ).
```

The value held by *Solution* upon *search(StartNode, EndNode, Solution)* suceeding will be a list of the form *[a,c,e,f,g,i]*; this list is then passed into the user-defined clause *construct_graphical_solution(Solution, GraphicalSolution)*, defined as follows:

```
construct_graphical_solution( [], [] ).

construct_graphical_solution( [Node|Nodes], [Tag|Tags] ) :-
        construct_graphical_node( Node, Tag ),
        construct_graphical_solution( Nodes, Tags ).

construct_graphical_node( Node, Tag ) :-
        Tag = img( src = verbatims @ [`images/netpaths/`,Node,`.gif`] ).
```

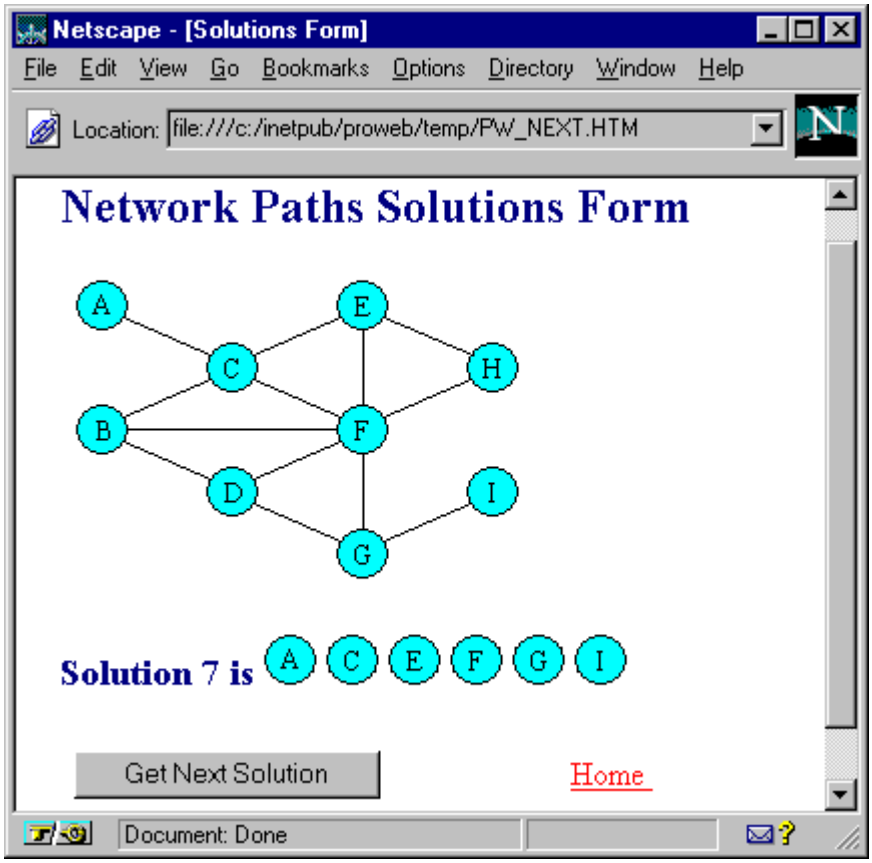
For each node in the network to be shown in graphical form, you are going to need nine GIF files; these should already be present in your C:\INETPUB\PROWEB\IMAGES\NETPATHS directory as A.GIF through to I.GIF. The above code converts a list of nodes into a list of HTML IMG elements. Given the list *[a,c,e,f,g,i]*, ProWeb will, upon executing *proweb_post_reply(solution, GraphicalSolution)*, asserts the fact:

```
proweb_data_reply( solution, [img(src = verbatims @
[`images/netpaths/`,a,`.gif`]),img(src = verbatims @
[`images/netpaths/`,c,`.gif`]),img(src = verbatims @
[`images/netpaths/`,e,`.gif`]),img(src = verbatims @
[`images/netpaths/`,f,`.gif`]),img(src = verbatims @
[`images/netpaths/`,g,`.gif`]),img(src = verbatims @ [`images/netpaths/`,i,`.gif`])]
)
```

into Prolog's internal database. When your Solutions Form is assembled, the above fact will be converted to:

```
<IMG SRC="images/netpaths/a.gif">
<IMG SRC="images/netpaths/c.gif">
<IMG SRC="images/netpaths/e.gif">
<IMG SRC="images/netpaths/f.gif">
<IMG SRC="images/netpaths/g.gif">
<IMG SRC="images/netpaths/i.gif">
```

in the resulting HTML source code; when displayed, your Solutions Form will look like:



Creating a You Are Already There! Form

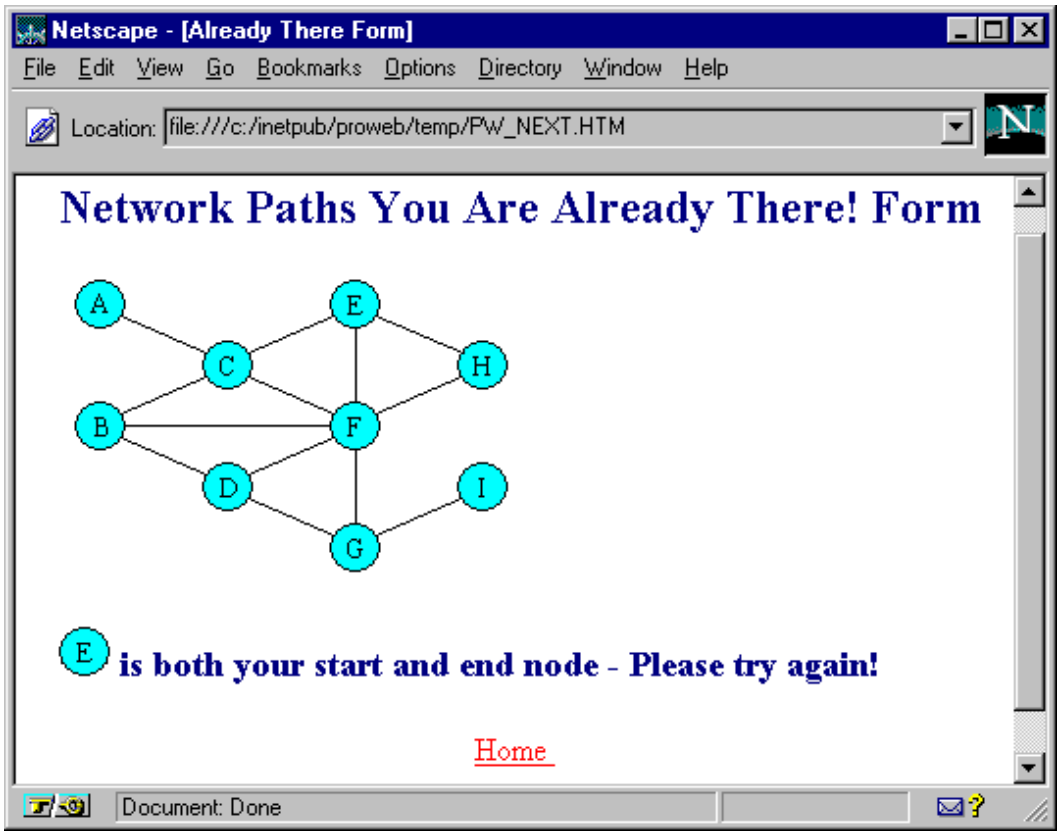
One situation that your application does not currently cater for is where the submitted start and end nodes are identical. As it seems unprofessional to return either a blank Solutions Form or a No More Solutions Form, a You Are Already There! Form needs to be defined:

```
form_title_file( already_there_form, `Already There Form`,
'html/netpaths/tmplate4.htm' ).

form_footer(  already_there_form,
              [ p(align='center') @ include('html/netpaths/link_hp.htm') ]
                    ).
```

In addition to the above, we also need to modify the first clause of *main_goal/0*:

```
main_goal :-
        proweb_send_form( node_selection_form ),
        proweb_returned_answer( start_node, StartNode ),
        proweb_returned_answer( end_node, EndNode ),
        (       StartNode = EndNode
        ->      (       write(`<IMG SRC= images/netpaths/`),
                        write(StartNode),
                        write(`.gif>`)
                ) ~>    GraphicalNode,
                proweb_post_reply( node, unencoded @ GraphicalNode ),
                proweb_send_form( already_there_form )
        ;       solution( search(StartNode,EndNode,Solution), Nth ),
                proweb_post_unique_reply( solution_nth, Nth ),
                construct_graphical_solution( Solution, GraphicalSolution ),
                proweb_post_reply(solution, GraphicalSolution ),
                proweb_send_form( solutions_form(Nth) )
        ).
```

When viewed in Netscape Navigator, your You Are Already There! Form will look like:



Presetting Answers

The default value for the start node has been set to 'a' by its *proweb_question/2* clause; likewise, the end node's default value is the last node in the network. Such a value can be overridden by a *proweb_preset_answer/2* call. Should you wish to set the start node's default value to 'b' and the end node's to 'h', your first *main_goal/0* clause can be amended as follows:

```
main_goal :-
        proweb_preset_answer( start_node, b ),
        proweb_preset_answer( end_node, h ),
        proweb_send_form( node_selection_form ),
        …
```

Caching the Solutions Between Pages to Avoid Re-Computation

As this network paths example computes the possible paths through the nine nodes of the network from the beginning each time, its speed can be dramatically increased by caching all the solutions during the very first run through. Such a caching mechanism also allows you to decide the order in which the solutions are presented to the client. Execute the following (all on one line) from the *WIN-PROLOG* command line:

?- **findall(Solution, (integer_bound(2,NodesVisited,9), search(a,i,Solution),len(Solution,NodesVisited)), Solutions), proweb_asserta(solutions(Solutions)).** <*enter*>

This algorithm gets the shortest solutions first. Its output will be:

```
Solution = _ ,
NodesVisited = _ ,
Solutions = [[a, c, f, g, i], [a, c, e, f, g, i], [a, c, f, d, g, i], [a, c, b, d, g, i], [a, c, b, f, g, i], [a, c, e, f, d, g, i], [a, c, e, h, f, g, i], [a, c, f, b, d, g, i], [a, c, b, d, f, g, i], [a, c, b, f, d, g, i], [a, c, e, f, b, d, g, i], [a, c, e, h, f, d, g, i], [a, c, e, h, f, b, d, g, i]]
```

To retrieve such cached solutions, execute the following from the *WIN-PROLOG* command line:

?- proweb_call(solutions(Solutions)). <enter>

You will find that the value of *Solutions* is the same as shown above.

Whenever *proweb_asserta(Clause)* or *proweb_assertz(Clause)* is executed during an interaction, ProWeb adds *Clause* to Prolog's internal database. When the next page is sent to the client, *Clause* is stored in the conversation database file associated with the conversation, ready to be added afresh to Prolog's 'cleared' internal database right at the beginning of the next interaction.

To implement the above into your network paths example, replace the line:

```
…
solution(search(StartNode,EndNode,Solution),Nth),
…
```

with the following:

```
…
(   proweb_call( solutions(Solutions) )
-> true
;   findall( Solution,
                ( integer_bound(2,NodesVisited,9),
                  search(StartNode,EndNode,Solution),
                  len(Solution,NodesVisited)
                ),
                Solutions
              ),
    proweb_asserta( solutions(Solutions) )
),
member( Solution, Solutions, Nth ),
…
```

The *proweb_call/1* clause will fail if *solutions/1* does not exist as a ProWeb-asserted fact in Prolog's internal database, resulting in *findall/3* finding all the solutions, from the shortest to the longest, which are then asserted via *proweb_asserta/1*. As a *solutions/1* clause is only being asserted where one does not already exist, you could have used *proweb_assertz/1* in the above code instead. The call to *member/3* will get the n'th solution, ready to pass on to the existing code.

Getting Information From Non-ProWeb Questions

If you have an existing web site to which ProWeb capabilities are being added, there may well be an occasion when you would want to use an existing HTML form to pass data onto ProWeb. As such a form was not created by ProWeb, ProWeb's procedure for receiving the data is slightly different to that already described. The phrase 'created by ProWeb' simply means that each input field within the form was generated from a *proweb_question/2* clause.

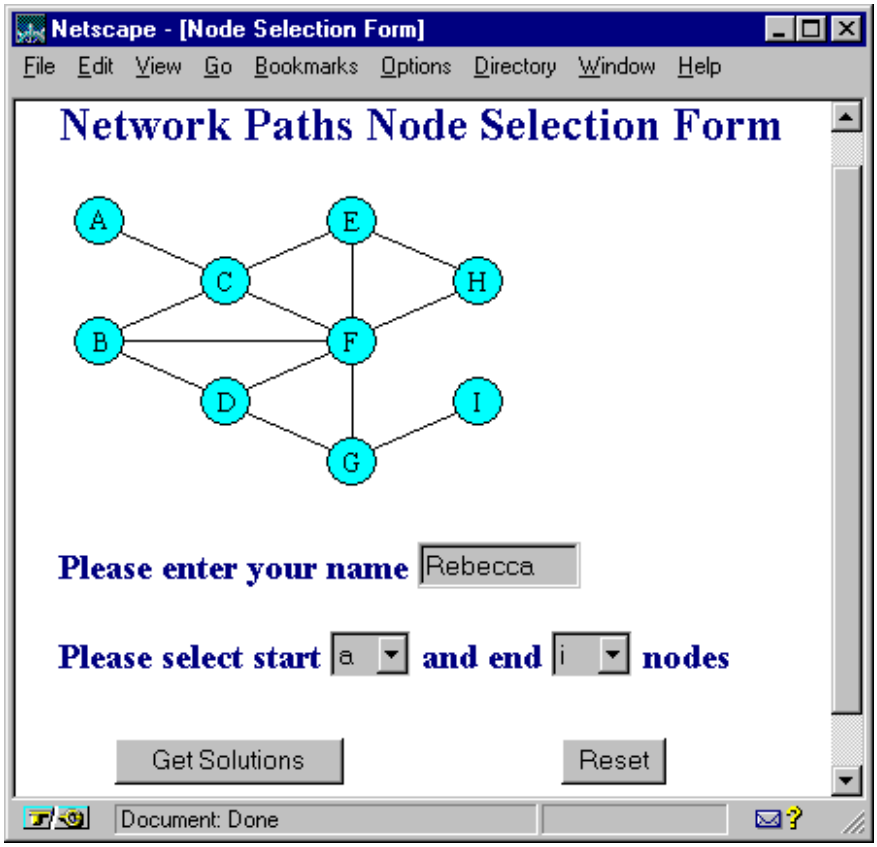
As you will recall, *proweb_returned_answer/2* is used to 'pick-up' the answer to a ProWeb-generated question; such a predicate will fail if asked to 'pick-up' an answer to a non-ProWeb-generated question. A 'non-ProWeb-generated' question is one where the HTML form code for the question is incorporated directly in the HTML page rather than via a <PROWEB QUESTION=…> element. The data entered into all the input fields of a form are sent to ProWeb when the HTML page is submitted; to 'pick-up' the answer to a 'non-ProWeb-generated' question, you need to use *proweb_returned_input/2* instead.

With *proweb_returned_answer/2*, ProWeb remembers the answer itself from one interaction of the conversation to the next. With *proweb_returned_input/2*, the answer is only remembered during the interaction in which it was input; should you need to remember such an answer, it must be manually asserted.

There is no problem in mixing ProWeb-generated and non-ProWeb-generated input fields within the same form. An example of this would be to add a non-ProWeb-generated text entry field to the Node Selection Form allowing the client's name to be entered; modify TMPLATE1.HTM as shown below:

```
<TABLE><TD></TD>
    <TD>
        <IMG SRC="images/network.gif">
        <H2>Network Paths Node Selection Form</H2>
        <H3>
            Please enter your name <INPUT NAME="my_name" SIZE="10">
        </H3>
        <H3>
            Please select start <PROWEB QUESTION="start_node">
             and end <PROWEB QUESTION="end_node"> nodes
        </H3>
    </TD>
</TABLE>
```

When displayed in Navigator, your revised Node Selection Form will look like:



When this page is submitted, the value of "my_name" (for illustration purposes, we have entered the name *Rebecca*) can be 'picked-up' via a call to *proweb_returned_input/2*. One important point that must be stressed is that this predicate will fail if the URL line does not contain the required field; The *proweb_returned_input/2* predicate must therefore be carefully placed so as to be called once only and directly following submission of the Node Selection Form. Amend the first *main_goal/0* clause as follows:

```
…
;       findall( Solution,
                 ( integer_bound(2,NodesVisited,9),
                   search(StartNode,EndNode,Solution),
                   len(Solution,NodesVisited)
                 ),
                 Solutions
               ),
        proweb_asserta( solutions(Solutions) ),
        proweb_returned_input( my_name, Name )
),
…
```

As we wish to demonstrate how a fact can be carried through a conversation, we are going to output the entered name on the No_More_Solutions_Form. Although the above code will obtain a value for *Name*, the program will forget it at the end of the interaction in which it was submitted unless it is asserted via a *proweb_asserta/1* or *proweb_assertz/1* call:

```
…
;       findall( Solution,
                 ( integer_bound(2,NodesVisited,9),
                   search(StartNode,EndNode,Solution),
                   len(Solution,NodesVisited)
                 ),
                 Solutions
               ),
        proweb_asserta( solutions(Solutions) ),
        proweb_returned_input( my_name, Name ),
        len( Name, L ),
        (       L > 0
        ->      cat([' ',Name],Name1,_)
        ;       Name1 = Name
        ),
        proweb_asserta( name(Name1) )
),
…
```

The additional code here simply puts a space character at the beginning of the name; you will appreciate why when you display the No More Solutions Form with and without a name.

Amend TMPLATE3.HTM file as shown below:

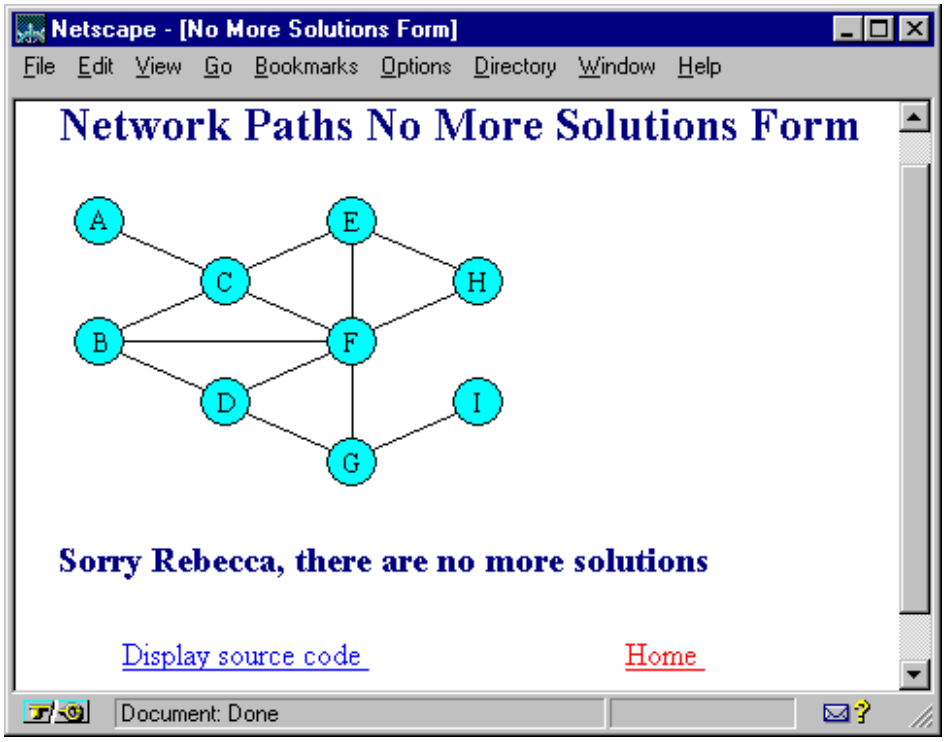
```
…
<FORM>
   <TABLE>
      <TD>
      </TD>
      <TD>
         <IMG SRC="network.gif">
```

```
            <H2>Network Paths No More Solutions Form</H2>
            <H3>Sorry <PROWEB REPLY="my_name">
            , there are no more solutions</H3>
          </TD>
        </TABLE>
      </FORM>
      ...
```

Prior to sending the No More Solutions Form, the asserted name needs to be retrieved with the *proweb_call/1* predicate; amend the second *main_goal/0* clause as follows:

```
main_goal :-
        proweb_returned_form( node_selection_form ),
        proweb_call( name(Name) ),
        proweb_post_reply( my_name, Name ),
        proweb_send_form( no_more_solutions_form ).
```

When run, your No More Solutions Form will now contain the line "Sorry <name>, there are no more solutions".



Conclusion

As you can now appreciate, ProWeb is extremely powerful when it comes to creating and controlling your web site, but programming a ProWeb-based application is not a trivial task. Due to the Web's asynchronous nature, you will find that applications destined for the Web need to be developed slightly differently from Windows-based ones; in fact, quite a lot of features normally found in Windows applications are simply not required for the web. The approach now adopted by LPA on many of its example programs is to develop the underlying algorithm separately from the front end; the exact same approach you have carried out in developing this example.

Chapter 11 - The Supplied ProWeb Directory

This chapter documents the contents of the supplied ProWeb directory. It also gives some idea as to what is going on behind the scenes when you run your ProWeb-based application.

Introduction

The supplied ProWeb directory is an almost complete installation of ProWeb. Only one file is missing; this being PROWEB.SYS. The file, PROWEB.SYS, is a renamed copy of the ***WIN-PROLOG*** executable, PRO386W.EXE. Copy PRO386W.EXE, which you'll find in your ***WIN-PROLOG*** root directory, to your ProWeb home directory (i.e. C:\INETPUB\PROWEB) and rename it PROWEB.SYS.

The Directory Structure

It is becoming common practice (if only for tidiness) for the directories of web-based applications to contain an IMAGES directory for image (i.e. GIF, JPEG and PNG) files, an HTML directory for HTML files and a TEMP directory for temporary files. To put you in good stead for the future, your ProWeb 'home directory' has been set up in this way. You may also have need for EXAMPLES and SYSTEM directories (these correspond to ***WIN-PROLOG***'s Examples and System directories), so these have been created for you as well.

The complete structure of the supplied ProWeb directory is as shown below:

C:\INETPUB\PROWEB\ EXAMPLES\

HTML\

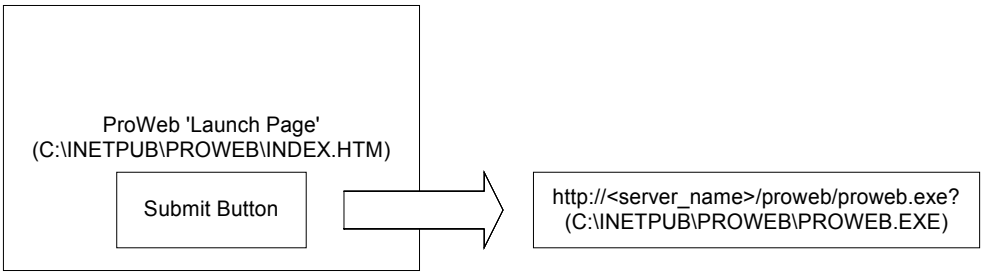
IMAGES\

SYSTEM\

TEMP\

INDEX.HTM

The files, INDEX.HTM, are 'launch pages' for ProWeb. Each of these pages, when submitted, calls PROWEB.EXE.



PROWEB.EXE

The file, PROWEB.EXE, is the 'glue' between the Common Gateway Interface (CGI) and the *WIN-PROLOG* executable, PROWEB.SYS.

PROWEB.SYS

The file, PROWEB.SYS, is simply a copy of the *WIN-PROLOG* executable, PRO386W.EXE, under another name.

PROWEB.OVL

The supplied PROWEB.OVL file contains the ProWeb Server toolkit. PROWEB.OVL is loaded automatically by PROWEB.SYS.

PROWEB.INI

The file, PROWEB.INI, contains the initialisation settings for a ProWeb session. This file is loaded automatically by PROWEB.SYS. You will need to change the settings, BASE_URL, TEMP_URL, HTML_PATH and TEMP_PATH to suit your particular installation.

The EXAMPLES Directory

The EXAMPLES directory contains example programs for you to try out.

The HTML Directory

The HTML directory contains HTML template files used by the supplied examples.

The IMAGES Directory

The IMAGES directory contains image (i.e. JPEG and GIF) files used by the supplied examples.

The SYSTEM Directory

The SYSTEM directory contains any system files requires by PROWEB.SYS.

The TEMP Directory

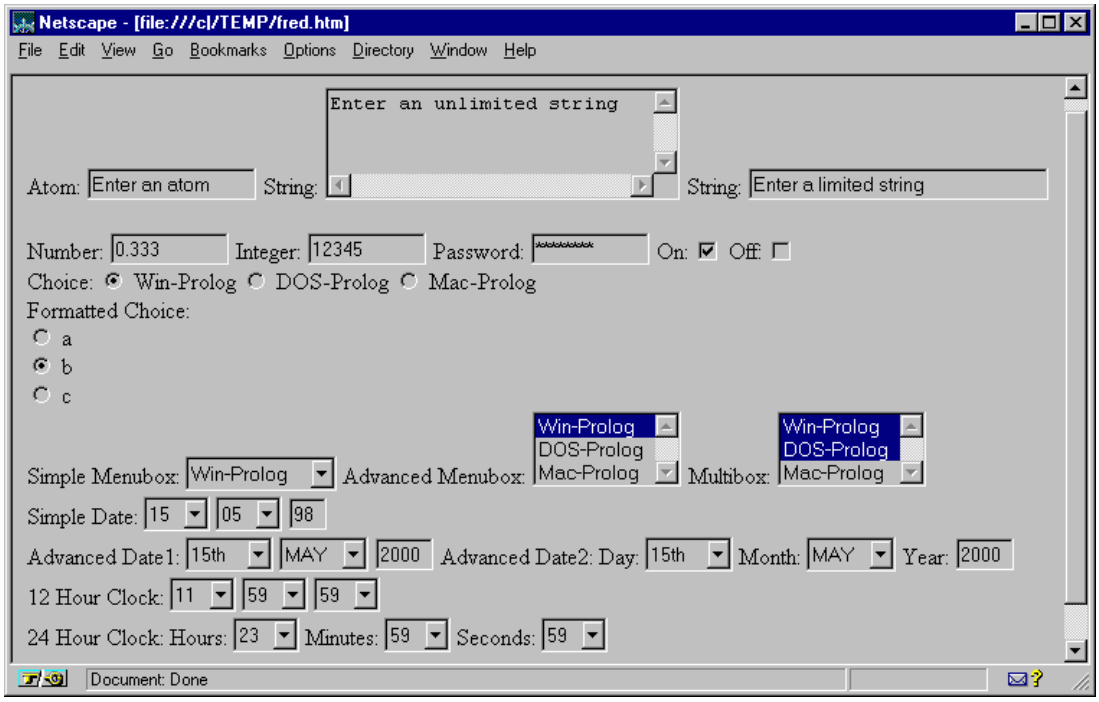
The TEMP Directory is where PROWEB.SYS places its "conversation" and log files.

Chapter 12 - Questions

This chapter describes how to define and ask questions in ProWeb.

Introduction

ProWeb supports all the input field types defined within HTML Forms in addition to its own date and time input fields.



Checkbox

The ProWeb checkbox field type maps onto the HTML form's <INPUT TYPE="checkbox"> type. In ProWeb, a checkbox question would be defined as:

```
proweb_question(    my_on_by_default_checkbox,
                    [       method      = checkbox,
                            prefill     = on
                    ] ).

proweb_question(    my_off_by_default_checkbox,
                    [       method      = checkbox,
                            prefill     = off
                    ] ).
```

Checkboxes

A question using the checkboxes method acts like a "multibox" but generates a list of checkboxes rather than a selection list.

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.

main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( multi_checkbox_input, Selection ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form,
          [ ?multi_checkbox_input,
            br,
            input(type=submit)
          ]
        ).

proweb_form( output_form, Selection ) :-
  proweb_returned_answer( multi_checkbox_input, Selection ).

proweb_question( multi_checkbox_input,
             [ method = checkboxes,
               select = [apple,banana,pear]
             ]
           ).
```

Date

The ProWeb date field type is unique to ProWeb and does not exist in HTML forms. In ProWeb, a date question would be defined as:

```
proweb_question(    date_input1,
                    [       method      = date,
                            format      = '[dd][mm][yy]'
                    ] ).

proweb_question(    date_input2,
                    [       method      = date,
                            format      = '[dd][mm][yyyy]',
                            prefill     = date(2000,5,15)
                    ] ).

proweb_question(    date_input3,
                    [       method      = date,
                            format      = '[d]th[mm][yyyy]',
                            prefill     = date(2000,5,15)
                    ] ).
```

```
proweb_question(    date_input4,
                    [       method      = date,
                            format      = '[d]th[M][yyyy]',
                            prefill     = date(2000,5,15)
                    ] ).

proweb_question(    date_input5,
                [       method      = date,
                        format      = 'Day: [d]th Month: [M] Year: [yyyy]',
                        prefill     = date(2000,5,15)
                ] ).

proweb_question(    date_input6,
                [       method      = date,
                        format      = 'Day: [dd]/[mm]/[yyyy]',
                        prefill     = date(2000,5,15)
                ] ).

proweb_question( date_input7,
                    [ method = date,
                      format  = '[m]/[d]/[yy]',
                      prefill   = date(1960,1,1)
                    ]
                ).
```

It is not possible to user-define lower and upper bounds for a date field, although ProWeb does ensure, via client-side Javascript, that any entered date is between 1st January 1901 and 31st December 2099 inclusive and that it is not an invalid date such as 30th February.

Your ProWeb application can also include four user-defined predicates, *proweb_month/3, proweb_day/3, proweb_weekday/3* and *proweb_ampm/3,* which allow you to convert between the integer and the string date representation:

proweb_month(?Month, ?M, ?MM)

Month	1-12	Month number
M	STRING	Short name (e.g. JAN)
MM	STRING	Long name (e.g. January)

proweb_day(?Day, ?D, ?DD)

Day	1-31	Day number
D	STRING	Short name (e.g. 1st)
DD	STRING	Long name (e.g. First)

proweb_weekday(?WeekDay, ?W, ?WW)

WeekDay	0 or 1-7	Day of Week number
W	STRING	Short name (e.g. MON)
WW	STRING	Long name (e.g. Monday)

proweb_ampm(?AMPM, ?M, ?MM)

AMPM	0 or 1	Morning or afternoon
M	STRING	Short name (e.g. am)
MM	STRING	Long name (e.g. A.M.)

Input

The ProWeb input field type maps onto either of the HTML form's <INPUT TYPE="text"> or <TEXTAREA>...</TEXTAREA> types depending on the number of rows required. ProWeb automatically handles for you the Prolog type (i.e. atom, string, integer or number) specified. A ProWeb text input field would be defined as follows:

```
proweb_question(    unlimited_atom_input,
                    [       method      = input,
                            type        = atom,
                            prefill     = 'Enter an atom',
                            cols        = 40
                    ] ).

proweb_question(    limited_atom_input,
                    [       method      = input,
                            type        = atom,
                            prefill     = 'Enter an atom',
                            cols        = 28,
                            maxlength   = 28
                    ] ).

proweb_question(    unlimited_string_input,
                    [       method      = input,
                            type        = string,
                            prefill     = `Enter an unlimited string`,
                            rows        = 3,
                            cols        = 32
                    ] ).

proweb_question(    limited_string_input,
                    [       method      = input,
                            type        = string,
                            prefill     = `Enter a limited string`,
                            cols        = 28,
                            maxlength   = 28
                    ] ).
```

```
proweb_question(    integer_input,
                    [       method      = input,
                            type        = integer,
                            prefill     = 1,
                            cols        = 2,
                            lwb         = 1, /* lower bound */
                            upb         = 9,  /* upper bound */
                            maxlength   = 10
                    ] ).

proweb_question(    number_input,
                    [       method      = input,
                            type        = number,
                            prefill     = 0.333,
                            cols        = 5,
                            lwb         = 0, /* lower bound */
                            upb         = 1, /* upper bound */
                            maxlength   = 10
                    ] ).
```

Keycode

The keycode method type allows you to ask a keycode question. A keycode consists of 15 alphanumeric characters arranged into five groups of three alphanumeric characters (e.g. 123-ABC-456-def-789).

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.

main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( keycode_input, Keycode ),
  proweb_send_form( output_form ).

proweb_page( _, [] ).

proweb_form( input_form,
          [ ?keycode_input,
            br,
            input(type=submit)
          ]
        ).

proweb_form( output_form, Keycode ) :-
  proweb_returned_answer( keycode_input, Keycode ).

proweb_question( keycode_input,
             [ method = keycode
             ]
           ).
```

Menubox

The ProWeb menubox field type maps onto the HTML form's <SELECT>...<OPTION>...</SELECT> type. A ProWeb menubox field would be defined as follows:

```
proweb_question(    threeline_menubox_input,
        [       method      = menubox,
                select      = ['Win-Prolog','DOS-Prolog','Mac-Prolog'],
                prefill     = 'Win-Prolog',
                rows        = 3
        ] ).

proweb_question(    number_menubox_input,
                    [       method      = menubox,
                            type        = number
                            select      = [3.3,3.5,3.6],
                            prefill     = 3.6,
                            rows        = 3
                    ] ).
```

Multibox

The ProWeb multibox field type maps onto the HTML form's <SELECT MULTIPLE>...<OPTION>...</SELECT> type. A ProWeb multibox field would be defined as follows:

```
proweb_question(    multibox_input,
        [       method      = multibox,
                select      = ['Win-Prolog','DOS-Prolog','Mac-Prolog'],
                prefill     = ['Win-Prolog','DOS-Prolog'],
                rows        = 3
        ] ).
```

Password

The ProWeb password field type maps onto the HTML form's <INPUT TYPE="password"> type. A ProWeb password field would be defined as follows:

```
proweb_question( password_input,
                    [       method      = password,
                            prefill     = 'mypassword',
                            maxlength   = 10
                    ] ).
```

Radio

The ProWeb radio field type maps onto the HTML form's <INPUT TYPE="radio"> type. A ProWeb radio field would be defined as follows:

```
proweb_question(    radio_input,
        [       method      = radio,
                select      = ['Win-Prolog','DOS-Prolog','Mac-Prolog'],
                prefill     = 'Win-Prolog'
        ] ).

proweb_question(    formatted_radio_input,
                    [       method      = radio,
                            type        = string,
                            select      = [`a`,`b`,`c`],
                            prefill     = `b`,
                            infix       = br
                    ] ).
```

Time

The ProWeb time field type is unique to ProWeb and does not exist in HTML forms. In ProWeb, a time question would be defined as:

```
proweb_question(    hour12_input,
                    [       method      = time,
                            format      = '[hh12][mm][ss]'
                    ] ).

proweb_question(    hour24_input1,
                    [       method      = time,
                            format      = '[hh24]:[mm]:[ss]',
                            prefill     = time(23,59,59,0)
                    ] ).

proweb_question(    hour24_input2,
                [       method= time,
                        format= 'Hours: [hh24] Minutes: [mm] Seconds: [ss]',
                        prefill = time(23,59,59,0)
                ] ).
```

Reference

Parameter	Valid Value(s)	Default Value	To be Used with
method	checkbox, date, input, menubox, multibox, password, radio and time	input	
type	atom, integer, number and string	atom	input, password, radio, menubox or multibox
format	ATOM or STRING		date or time
select	LIST(TYPE)		radio, menubox or multibox

			multibox
prefill	TYPE		input, password, radio or menubox
prefill	LIST(TYPE)		multibox
prefill	off, on	off	checkbox
prefill	date(YEAR , MONTH , DAY)		date
prefill	time(HOUR , MINUTE , SECOND, HUNDREDTHS)		time
infix	HTMLOBJECT		radio
rows	INTEGER	1	input, menubox or multibox
cols	INTEGER	10	input
maxlength	INTEGER		input or password
lwb	NUMBER		input number or input integer
upb	NUMBER		input number or input integer

Making A Question Compulsory

You can make a question compulsory by adding the 'compulsory = yes' or 'compulsory = on' attribute to the question's definition. The default value for this attibute is 'no' or 'off'.

```
proweb_question( compulsory_input,
                 [ method     = input,
                   type       = atom,
                   compulsory = yes
                 ]
               ).
```

Inputting and Checking Prolog Data Type

We are now going to look at how information is entered into and output from a ProWeb-based application; the following ProWeb-based program gives a demonstration of this. For simplicity, the two forms defined within this program can use the same user-defined *proweb_page/2* clause:

```
proweb_page( _, [] ).
```

As the underlying program to any ProWeb-based application is going to be written in Prolog, any information entered needs to be handled as either an atom, a string, an integer, a floating point number or a list. For this simple program, we are going to define a form into which can be entered an example of each of these LPA Prolog data types.

As you are already aware, a form is defined via a user-defined *proweb_form/2* clause:

```
proweb_form( input_form,
        [ p, `Please enter an atom: `, ?atom_input,
          p, `Please enter a string: `, ?string_input,
          p, `Please enter an integer: `, ?integer_input,
          p, `Please enter a float: `, ?float_input,
          p, `Please select an atom list: `, ?atom_list_input,
          p, input(type=submit)
        ]
      ).
```

Prior to being sent to the client, the above *proweb_form/2* clause would be 'expanded' by ProWeb. Although this complex process is performed automatically by ProWeb, such a form definition needs to be presented to ProWeb in a pre-defined format. The second argument of *proweb_form/2* is a list of terms; these terms can be the Prolog notation for HTML elements and attributes; for the clause above, *p* gets expanded to *<P>* and *input(type=submit)* gets expanded to *<INPUT TYPE=SUBMIT>*. A term such as *?atom_input* is a little more complex; this means embed the *atom_input* question at this point in the form.

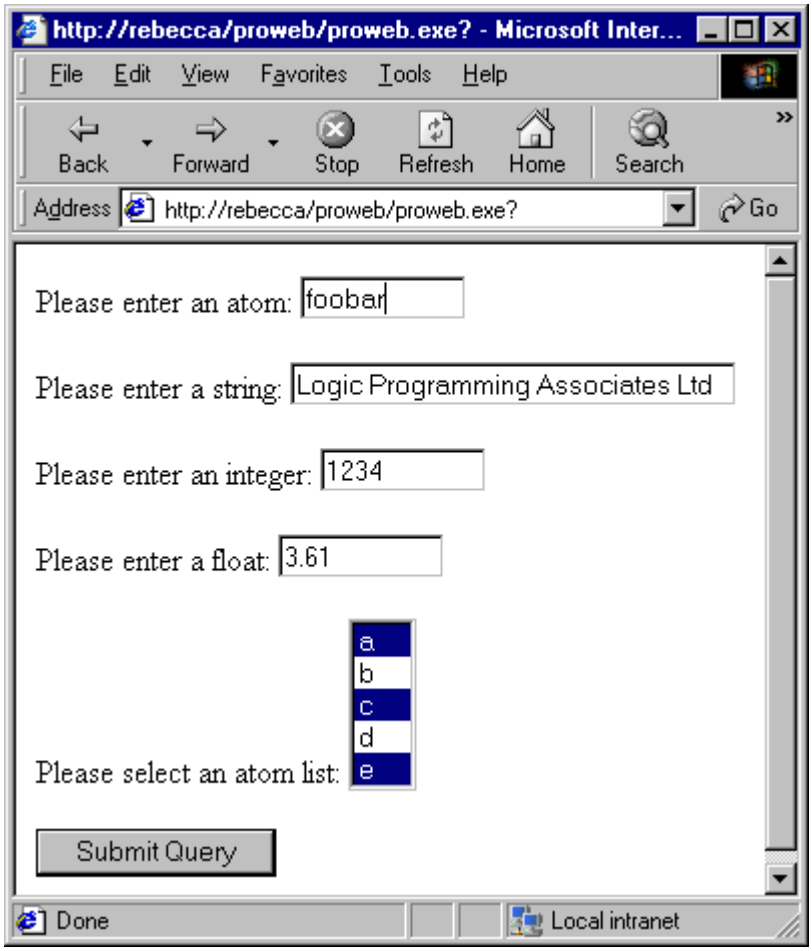
A question is defined via a user-defined *proweb_question/2* clause, for this example, we need five such clauses:

```
proweb_question( atom_input,
          [ method  = input,
            type    = atom,
            prefill = MyAtom
          ]
        ) :- MyAtom = foo, atom( MyAtom ).

proweb_question( string_input,
          [ method  = input,
            type    = string,
            prefill = MyString,
            cols    = 30
          ]
        ) :- MyString = `Logic Programming Associates`, string( MyString).
```

```
proweb_question( integer_input,
                 [ method  = input,
                   type    = integer,
                   prefill = MyInteger
                 ]
               ) :- MyInteger = 123, integer( MyInteger ).

proweb_question( float_input,
                 [ method  = input,
                   type    = number,
                   prefill = MyFloat
                 ]
               ) :- MyFloat = 3.6, float( MyFloat ).

proweb_question( atom_list_input,
                 [ method  = multibox,
                   type    = atom,
                   select  = MyAtomList,
                   rows    = 5
                 ]
               ) :- MyAtomList = [a,b,c,d,e], list( MyAtomList ).
```

A *proweb_question/2* clause, when 'expanded', becomes an <INPUT> element in the final HTML page. In the above *proweb_question/2* clauses, *method* specifies the HTML TYPE attribute value whilst *prefill* and *select* specify the initial value for the field.

Whenever ProWeb generates a question, it will not allow us to assign a value which is of the wrong data type; for this example, by way of clarification, we have gone a little 'overboard' in making sure the value assigned is of the right data type.



The program's 'main goal' is as follows:

```
main_goal :-
  proweb_send_form( input_form ),
  proweb_returned_answer( atom_input, AtomInput ),
  type( AtomInput, 3 ),
  proweb_returned_answer( string_input, StringInput ),
  type( StringInput, 4 ),
  proweb_returned_answer( integer_input, IntegerInput ),
  type( IntegerInput, 1 ),
  proweb_returned_answer( float_input, FloatInput ),
  type( FloatInput, 2 ),
  proweb_returned_answer( atom_list_input, AtomListInput ),
  ( type(AtomListInput,5); type(AtomListInput,6) ),
  proweb_send_form( output_form ).
```

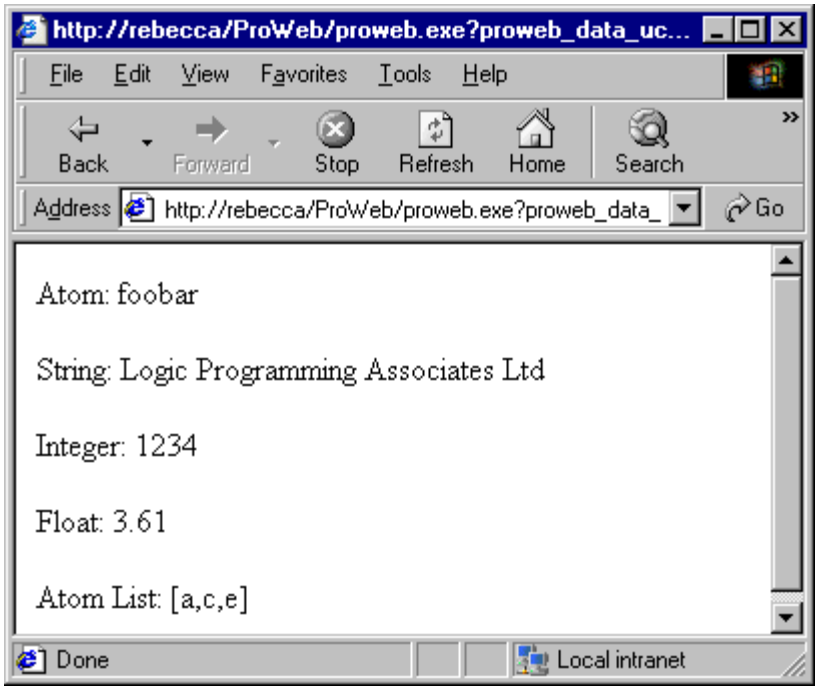
main_goal/0 initially sends the *input_form* to the client, whereupon the program terminates:

The client, upon receiving the form, enters a value into each of its five fields and then clicks on its submit button; whereupon *main_goal/0* is executed from the beginning again. With the submission of any form over an intranet or the Internet, the value for each field is returned to the server; for most ProWeb-based application, the *proweb_returned_answer/2* predicate is used to 'pick-up' such values. For this

example, *proweb_returned_answer(atom_input,AtomInput)* 'picks-up' the value of *atom_input* whilst *type(AtomInput,3)*, going 'overboard' again, just ensures that it really is of the atom data type.

Having retrieved each field's value and checked that it is of the correct data type, the *output_form* is sent to the client, the *proweb_form/2* clause for which is as follows:

```
proweb_form( output_form,
             [ p, `Atom: `, AtomOutput,
               p, `String: `, StringOutput,
               p, `Integer: `, IntegerOutput,
               p, `Float: `, FloatOutput,
               p, `Atom List: `, verbatim @ AtomListOutput
             ]
           ) :-
  proweb_returned_answer( atom_input, AtomOutput ),
  proweb_returned_answer( string_input, StringOutput ),
  proweb_returned_answer( integer_input, IntegerOutput ),
  proweb_returned_answer( float_input, FloatOutput ),
  proweb_returned_answer( atom_list_input, AtomListOutput ).
```



The reason for sending the *output_form* is just to confirm that the values were received within the program correctly. The 'verbatim @' before AtomListOutput prevents 'a' and 'b', if selected, being converted to HTML tags.

The above program is a little hypothetical in that you will normally never need to check that the value to a question is of a particular data type. ProWeb will build the client-side checking into the final HTML page itself via one or more Javascript programs. To test the former, enter a non-integer value into the integer field and try submitting the form; you will find an error message appears, preventing you from doing so.

One exception is the float data type. Defining a question with *type = number* ensures that ProWeb will only allow a number to be assigned to it, but such a number could also be an integer! In the above program, the *float/1* predicate is used to check that the value of this field is correct.

HTML TextArea Field Text to Prolog List

Should you ever need to generate a Prolog list from the text in an HTML TextArea field, where each line in the TextArea field is to become an element in the list, the following source code example may prove useful:

```
textarea_list( TextBox, List ) :-
  textarea_list2( [], ReversedList ) <~ TextBox,
  reverse( ReversedList, List ).
```

```
textarea_list2( Dummy, List ) :-
 find( `~M~J`, 3, Found ) ~> Element,
 (  Found = ``
 -> (  Element = ``
    -> List = Dummy
    ;  List = [Element|Dummy]
    )
 ;  (  Element = ``
    -> textarea_list2( Dummy,          List )
    ;  textarea_list2( [Element|Dummy], List )
    )
 ).
```

From the *WIN-PROLOG* command line, *textarea_list/2* would return:

```
?- textarea_list( `First Line~M~JLine 2~M~J~M~J~M~JLine3~M~JLast Line`,
List ). <enter>
List = [`First Line`,`Line 2`,`Line3`,`Last Line`]
```

What Happens To Questions Internally

All the fields generated by ProWeb from a *proweb_question/2* clause are allocated unique names from q0001 onwards. Internally, ProWeb maps Qxxxx onto the corresponding question. If a form contains five ProWeb-generated questions, their names will be q0001-q0005. Non-ProWeb questions would not appear with Qxxxx names, hence the reason why they need to be picked-up with *proweb_returned_input/2* rather than *proweb_returned_answer/2*.

Chapter 13 - Prolog Terms To HTML Code

This chapter documents the pseudo-HTML syntax that can be used in *proweb_page/2* and *proweb_form/2* clauses. It also describes the translation of Prolog terms to HTML code when used in such clauses.

The proweb_page/2 Clause

The first argument of a user-defined *proweb_page/2* clause specifies the name of the form to which the clause relates. This argument can be an underscore (which will match with all forms):

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [] ).

proweb_form( _, `A blank form!` ).
```

a single form name:

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( my_form, [] ).

proweb_form( my_form, `A blank form!` ).
```

or a list of form names:

```
main_goal :-
  proweb_send_form( [my_form,another_form] ).

proweb_page( [my_form,another_form], [] ).

proweb_form( [my_form,another_form], `A blank form!` ).
```

The second argument of a user-defined *proweb_page/2* clause specifies the actual HTML code for the page ProWeb is to generate. Such HTML code can be obtained from one or more template HTML files on disk or coded in Prolog itself. Rather than point a *proweb_page/2* clause at an HTML template file on disk, you can, as an alternative, specify the structure for the HTML page as a list of Prolog terms. The following *proweb_page/2* clause sets the background colour of the HTML page to black:

```
proweb_page( my_page,
 [ html,
   head,
   /head,
   body(bgcolor=black),
   /body,
   /html
 ]
).
```

The proweb_form/2 Clause

The second argument of a user-defined *proweb_form/2* clause specifies the actual HTML code for the form. Like *proweb_page/2*, such HTML code can be obtained from one or more template HTML files on disk or coded in Prolog itself.

HTML Elements

ProWeb recognises the following list of HTML 4.0 elements:

> !, !doctype, a, address, area, b, base, bgsound, big, blockquote, body, br, caption, cite, code, col, colgroup, comment, dd, dfn, div, dl, dt, em, embed, form, frame, frameset, h1, h2, h3, h4, h5, h6, head, hr, html, i, img, input, kbd, li, link, map, marquee, meta, nobr, noframes, object, ol, option, p, param, pre, samp, script, select, small, span, strong, style, sub, sup, table tbody, td, tfoot, th, thead, textarea, title, tr, tt, ul, var and wbr.

The following HTML elements deprecated in HTML 4.0 or already obsolete are also recognised:

> applet, basefont, center, dir, font, isindex, listing, menu, plaintext, s, strike, u and xmp.

HTML Attributes

ProWeb recognises the following list of HTML attributes:

> action, align, alt, background, behavior, bgcolor, bgproperties, border, bordercolor, bordercolordark, bordercolorlight, cellpadding, cellspacing, checked, class, classid, clear, code, codebase, codetype, color, cols, colspan, controls, content, coords, data, declare, direction, dynsrc, face, frame, frameborder, framespacing, height, href, hspace, http_equiv, ismap, language, link, loop, marginheight, marginwidth, maxlength, method, multiple, name, nohref, noshade, nowrap, palette, prompt, rel, rev, rows, rowspan, rules, scrollamount, scrolldelay, scrolling, selected, shape, shapes, size, span, src, standby, start, style, target, text, title, topmargin, type, usemap, valign, value, valuetype, vlink, vspace and width.

HTML Attribute Values

ProWeb recognises the following list of HTML attribute values:

above, all, alternate, baseline, below, _blank (entered as blank_), bottom, border, box, center, checkbox, circ, circle, cols, data, fileopen, fixed, get, groups, hidden, hsides, image, infinite, justify, left, lhs, middle, mouseover, no, none, nowrap, object, order_A, order_a, order_I, order_l, order_1, _parent (entered as parent_), password, poly, polygon, post, radio, rect, rectangle, ref, reset, rhs, right, rows, scroll, _self (entered as self_), slide, submit, text, textarea, top, _top (entered as top_), void, vsides and yes.

Translation of "Encoding" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
OBJECT	`This is "quoted"`	This is "quoted"
encoded(HOW) @ OBJECT	encoded(javascript) @ `Line "1"~M~lIndented Line 2`	Line "1"\r\tIndented Line 2
encoded(BYTE = OBJECT) @ OBJECT	encoded(13='/R') @ `Line "1"~M~lIndented Line 2`	Line "1"/R~lIndented Line 2
unencoded @ OBJECT	unencoded @ `This is "quoted"`	This is "quoted"

The default is to unencode ('unencoded @' behaviour is assumed).

You can also unencode included files; e.g. unencoded @ include('link.htm').

Translation of "Verbatim" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
verbatim @ OBJECT	b @ verbatim @ `This is verbatim`	<B>This is verbatim</B>
verbatim @ OBJECT	verbatim @ b @ `This is verbatim`	b @ This is verbatim
verbatims @ LIST	verbatims @ [`These`, `are`, `verbatim`]	Theseareverbatim
... @ writeq @ verbatim @ OBJECT	b @ writeq @ verbatim @ `This is quoted verbatim`	<B>`This is quoted verbatim`</B>
writeq @ verbatim @ OBJECT	writeq @ verbatim @ b @ `This is quoted verbatim`	b @ `This is quoted verbatim`

OBJECT		
writeq @ verbatims @ LIST	writeq @ verbatims @ [`These`, `are`, `quoted verbatim`]	`These``are``quoted verbatim`
unencoded @ … @ verbatim @ OBJECT	unencoded @ b @ verbatim @ `This is verbatim`	<B>This is verbatim</B>

Translation of "number" Prolog Terms to HTML Code

Definition	**Prolog Term Within proweb_form/2**	**HTML Code Generated**
number(DECIMALS) @ NUMBER	number(0) @ 123.456789	123
	number(2) @ 123.456789	123.46
	number(9) @ 123.456789	123.456789000

Translation of "List" Prolog Terms to HTML Code

Definition	**Prolog Term Within proweb_form/2**	**HTML Code Generated**
ELEMENT @ LIST dl denotes a definition list, normally used for a list of defined terms	dl @ [b @ alpha='One', b @ beta='Two', b @ omega='Last']	<DL> <DT><B>alpha</B> <DD>One <DT><B>beta</B> <DD>Two <DT><B>omega</B> <DD>Last</DL>
ELEMENT @ LIST ol denotes an ordered list, each item having a number or letter reference	ol(start=4, type=i) @ [alpha, beta, gamma]	<OL START=4 TYPE=i> <LI>alpha <LI>beta <LI>gamma </OL>

ELEMENT @ LIST ul formats lines of text as a bulleted list	ul @ [alpha, beta, gamma]	<UL> <LI>alpha <LI>beta <LI>gamma </UL>

Translation of "quotation" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
quotation(CHAR) @ OBJECT	b @ quotation(`"`) @ 'Bold'	<B>"Bold"</B>
	b @ quotation(`"`) @ i @ 'Bold and italic'	<B>"<I>Bold and italic</I>"</B>

Translation of "currency" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
currency @ NUMBER	currency @ -1234.56	(£1,234.56) See note below
currency(FORMAT) @ NUMBER	currency(bracketed,$,',','.') @ -1234.56	($1,234.56)

If 'currency @ NUMBER' is executed, the HTML code generated is dependant upon the settings of CURRENCY_SIGN, CURRENCY_SYMBOL, CURRENCY_THOUSAND and CURRENCY_DECIMAL in the PROWEB.INI file.

Translation of "date" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
date(FORMAT) @ date(YEAR, MONTH,DAY)	date('[W] [d][mm][yyyy]') @ date(1997,5,9)	FRI 9051997
	date('[WW] [dd][m][yyyy]') @ date(1997,5,11)	Sunday 1151997

	date('[d]th[MM][yy]') @ date(1997,12,11)	11thDecember97
	date('The date is [d]th[M] "[yy]"') @ date(2097, 12, 11)	The date is 11thDEC "97"

Important note: /yy produces the same output for both 1997 and 2097!

Translation of "Time" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
time(FORMAT) @ time(HOUR, MINUTE, SECOND, HUNDREDTH)	time('[hh24][mm][ss]') @ time(15, 1, 9, 0)	150109
	time('[hh12]:[m]:[s] [ampm]') @ time(3, 1, 9, 0)	3:1:9 am
	time('[hh24]00 hours') @ time(15, 1, 59, 0)	1500 hours
	time('The time is [hh12]:[mm]:[ss] [ampm]') @ time(15, 1, 59, 0)	The time is 3:01:59 pm

Translation of "Writing" Prolog Terms to HTML Code

Definition	Prolog Term Within proweb_form/2	HTML Code Generated
display @ OBJECT	display @ `A` - `B`	-(A,B)
write @ OBJECT	write @ `A` - `B`	A - B
write(PREDICATE) @ OBJECT	write(print_postfix) @ `A` - `B`	AB-
writeq @ OBJECT	writeq @ `A` - `B`	`A` - `B`

The html_print/1 and html_print_nl/1 Predicates

The ProWeb predicates, *html_print/1* and *html_print_nl/1*, take the text of their argument and convert it to recognisable HTML code. As a simple example, execute the following from the *WIN-PROLOG* command line (assuming PROWEB.OVL is being used instead of PRO386W.OVL):

```
?- html_print( i @ `Hello`).<enter>
<I>Hello</I>yes
```

Should you wish to capture the above output together with a *<carriage return>* character and re-direct it to a string, the following instruction should be issued:

```
?- html_print_nl( i @ `Hello`) ~> String.<enter>
String = `<I>Hello</I>~M~J`
```

When used in conjunction with *tab/1* and *write/1*, complex strings can be built-up and embedded within your HTML pages:

```
main_goal :-
        create_string ~> String,
        proweb_post_reply( encoded_string,   encoded @ String     ),
        proweb_post_reply( unencoded_string, unencoded @ String ),
        proweb_send_form( first_form ).

proweb_page( [first_form], [] ).

proweb_form(        first_form, [ include('page.htm') ] ).

create_string :-
        html_print_nl( table ),
        html_print_nl( td ),
        html_print( b ),
        html_print( i @ `Hello` ),
        tab( 5 ),
        write( `World!` ),
        html_print_nl( /b ),
        html_print_nl( /td ),
        html_print_nl( /table ).
```

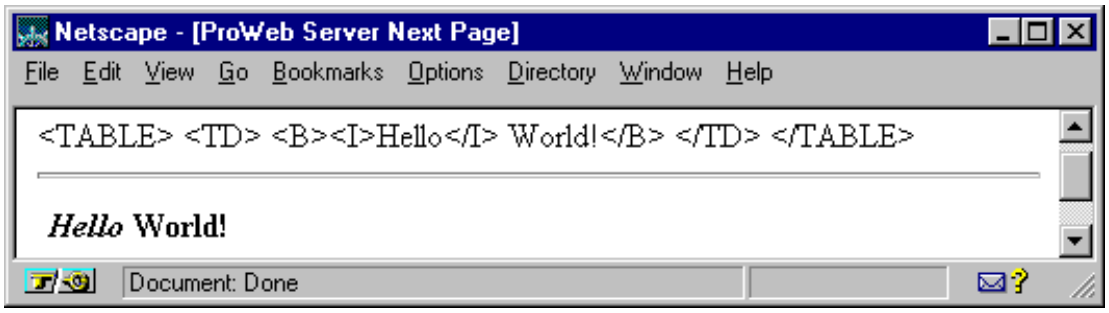
The output from the user-defined *create_string/0* clause is re-directed to the variable *String*. As two or more adjacent spaces in HTML code are interpreted as just one, the *tab/1* predicate is really only of assistance when used within a <PRE> element.

The text of PAGE.HTM, referred to in the above program, is as follows:

```
<HTML>
   <HEAD>
      <TITLE>
      </TITLE>
   </HEAD>
   <BODY>
```

```
            <PROWEB REPLY="encoded_string">
            <HR>
            <PROWEB REPLY="unencoded_string">
        </BODY>
    </HTML>
```

When run, the created HTML page will look like:



Although the string is created only once by the program, it is asserted into Prolog's internal database in two distinct ways. When 'picked-up' by ProWeb the first, being encoded, has its '<' and '>' characters converted to *<* and *>* respectively. The second, being unencoded, has its '<' and '>' characters accepted without modification.

The actual source code for this page is as follows:

```
<HTML>
    <HEAD>
        <TITLE>ProWeb Server Next Page</TITLE>
    </HEAD>
    <BODY>
        <FORM METHOD=POST>
        <INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[133]">
        <INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
        &lt;TABLE&gt;
        &lt;TD&gt;
        &lt;B&gt;&lt;I&gt;Hello&lt;/I&gt;    World!&lt;/B&gt;
        &lt;/TD&gt;
        &lt;/TABLE&gt;
        <HR>
        <TABLE>
        <TD>
        <B><I>Hello</I>    World!</B>
        </TD>
        </TABLE>
        </FORM>
    </BODY>
</HTML>
```

The predicates, *html_print/1* and *html_print_nl/1*, can also be used to create an HTML page and store it on disk. As the resulting HTML page is a fixed file on your server, it can be accessed in the same way as any other HTML page; such an HTML page could be hyperlinked to from other pages on your web site, including the pages of your

ProWeb application. The example below creates an HTML file named MY_PAGE.HTM in ProWeb's temporary directory:

```
create_and_store_html_page :-
        fcreate( my_page, 'c:\inetpub\proweb\temp\my_page.htm', -1,0,0 ),
        fopen( my_page, 'c:\inetpub\proweb\temp\my_page.htm', 1 ),
        output( Current ),
        output( my_page ),
        header( Header ),
        write( Header ),
        body( Body ),
        write( Body ),
        footer( Footer ),
        write( Footer ),
        output( Current ),
        fclose( my_page ).

header( Header ) :-
        (
        html_print(html),
        html_print(head),
        html_print(/head)
        ) ~> Header.

body( Body ) :-
        (
        html_print(body),
        html_print(`My Page`),
        html_print(/body)
        ) ~> Body.

footer( Footer ) :-
        html_print(/html) ~> Footer.
```

There may also be occasions when you want to convert a list of Prolog pseudo-HTML terms into real HTML; the following program may prove of assistance:

```
prolog_terms_to_html( List, HTML ) :-
  prolog_terms_to_html2( List ) ~> HTML.

prolog_terms_to_html2( [] ).

prolog_terms_to_html2( [Head|Body] ) :-
  html_print( Head ),
  prolog_terms_to_html2( Body ).
```

When executed, prolog_terms_to_html/2 performs as follows:

```
?- prolog_terms_to_html( [h3 @ `Heading`,`This is some
text`,input(type=submit)], HTML ). <enter>
HTML = `<H3>Heading</H3>This is some text<INPUT TYPE=SUBMIT>`
```

The include/1 Term

One facility missing from HTML is the ability to embed one HTML file within another; ProWeb, on the other hand, does possess such an ability in the form of the *include/1* term:

```
proweb_page( my_form, [include('template.htm')] ).
```

The *include/1* term can only be used within *proweb_page/2* or *proweb_form/2* clauses. The first and only argument of an *include/1* term is the name of an HTML template file (or text file) on disk. Such an HTML template file can use one of a number of character encodings such as ASCII, Unicode, etc.

The above *proweb_page/2* clause tells ProWeb that when it is constructing the page in which to embed *my_form*, it should use the HTML code found within TEMPLATE.HTM.

A complex example of *include/1* in use might be:

```
main_goal :-
 proweb_send_form( my_form ).

proweb_page( Form,
        [ include('head.htm'),
          `This is my title`,
          include('body.htm'),
          proweb(Form),
          include('foot.htm')
        ]
       ).

proweb_form( my_form, [ p(align='center') @ include('text.htm'), unencoded @
include('link.htm')] ).
```

If HEAD.HTM contains:

```
<HTML>
 <HEAD>
  <TITLE>
```

BODY.HTM contains:

```
</TITLE>
 </HEAD>
 <BODY BGCOLOR="blue">
```

FOOT.HTM contains:

```
   </BODY>
</HTML>
```

TEXT.HTM contains:

```
This is some text
```

and LINK.HTM contains:

```
<A HREF="mailto:support@lpa.co.uk">LPA Support</A>
```

the HTML code of the page generated by ProWeb will be:

```
<HTML>
 <HEAD>
  <TITLE>This is my title</TITLE>
 </HEAD>
 <BODY BGCOLOR="blue">
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[51]">
   <INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
   <PROWEB my_form>
   <P ALIGN=CENTER>This is some text</P>
   <A HREF="mailto:support@lpa.co.uk">LPA Support</A>
  </FORM>
 </BODY>
</HTML>
```

Whenever *include/1* is encountered by ProWeb, it causes ProWeb to insert, at that point in the page or form it is constructing, the text contained in the given HTML template file. The *include/1* term simply sets the current input stream to the HTML file given, reads in the file and then sets the current input stream back to what it was before.

The two lines, <INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[51]"> and <INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">, form part of the HTML form but are invisible when the page is displayed in the web browser. They are used by ProWeb for housekeeping purposes.

When ProWeb undertakes the task of constructing a page or form, any HTML code missing from an HTML template file that ProWeb deems necessary for an HTML page to be 'complete' will be automatically added to the page it is generating; be careful as certain combinations of 'incomplete' HTML template files can cause unexpected results!

The following example shows what happens when you state that a form is to be made up of code within an HTML template file, FORM.HTM, but additional code is included in the file outside of the form. The program is:

```
main_goal :-
 proweb_send_form( my_form ).

proweb_page( _, [] ).

proweb_form( my_form, include('form.htm') ).
```

The HTML code in FORM.HTM is:

```
This line will NOT appear in the final page
<FORM>
This is my form!
</FORM>
This line will NOT appear in the final page
```

The resulting HTML page generating by ProWeb is:

```
<HTML>
 <HEAD>
  <TITLE>
  </TITLE>
 </HEAD>
 <BODY>
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[61]">
   <INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
   This is my form!
  </FORM>
 </BODY>
</HTML>
```

You will notice that the two lines,"This line will NOT appear in the final page", which are outside of the <FORM> tags do, indeed, not appear in the final page!

8-Bit and Unicode Characters

Although HTML files are based on the 7-bit character set, some web browsers are able to display Unicode characters contained within them. A Unicode character is specified in an HTML file by use of a special notation.

ProWeb automatically converts certain 8-bit character codes into their special HTML notation:

Character Code	**HTML Notation**
034	&quot;
038	&amp;
060	&lt;
062	&gt;
160	
161	&iexcl;
162	&cent;
163	&pound;

164	&curren;
165	&yen;
166	&brvbar;
167	&sect;
168	&uml;
169	&copy;
170	&ordf;
171	&laqno;
172	&not;
173	&shy;
174	&reg;
175	&macr;
176	&deg;
177	&plusmn;
178	&sup2;
179	&sup3;
180	&acute;
181	&micro;
182	&para;
183	&middot;
184	&cedil;
185	&sup1;
186	&ordm;
187	&raquo;
188	&frac14;
189	&frac12;

190	¾
191	¿
192	À
193	Á
194	Â
195	Ã
196	Ä
197	Å
198	Æ
199	Ç
200	È
201	É
202	Ê
203	Ë
204	Ì
205	Í
206	Î
207	Ï
208	Ð
209	Ñ
210	Ò
211	Ó
212	Ô
213	Õ
214	Ö
215	×

216	Ø
217	Ù
218	Ú
219	Û
220	Ü
221	Ý
222	Þ
223	ß
224	à
225	á
226	â
227	ã
228	ä
229	å
230	æ
231	ç
232	è
233	é
234	ê
235	ë
236	ì
237	í
238	î
239	ï
240	ð
241	ñ

242	&ograve;
243	&oacute;
244	&ocirc;
245	&otilde;
246	&ouml;
247	&divide;
248	&oslash;
249	&ugrave;
250	&uacute;
251	&ucirc;
252	&uuml;
253	&yacute;
254	&thorn;
255	&yuml;

The following example outputs a pound sign and an ampersand:

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [] ).

proweb_form( my_form,
             [ encoded @ `~(26) ~(7E) ~(A3)`,
               br,
               encoded @ `& ~~ £`,
               br,
               unencoded @ `&amp;amp; ~~ &amp;pound;`,
               br,
               unencoded @ `&amp;#38; &amp;#126; &amp;#163;`
             ]
           ).
```

The generated HTML code is as follows:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[427]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
&amp;amp; ~ &amp;pound;
<BR>
&amp;amp; ~ &amp;pound;
<BR>
&amp;amp; ~ &amp;pound;
<BR>
&amp;#38; &amp;#126; &amp;#163;
</FORM>
</BODY>
</HTML>
```

Chapter 14 - The <PROWEB> Element

This chapter describes what the <PROWEB> element is and how to use it. It also documents the link between the question mark notation used within a *proweb_form/2* clause and the <PROWEB> element.

Introduction

A common practice with applications which manipulate HTML files is to embed their own elements within the files which only they understand. ProWeb, being such an application, has its own element – the <PROWEB…> element. The <PROWEB> element is embedded within HTML template files.

It is a feature of every web browser to ignore any elements found in a loaded HTML file which it does not understand. It is, therefore, perfectly alright to load into a web browser an HTML page containing one or more <PROWEB…> elements; it is also perfectly alright to create such pages with an HTML editor such as Microsoft's FrontPage or Adobe's PageMill.

When reading in a 'template' HTML file, ProWeb treats each <PROWEB…> element as a directive to itself to insert something in its place, such as the form-related HTML code for asking a question, giving a reply, inserting a link to helpful information, inserting a value or inserting the form(s) at a particular point.

Where you do not wish to use a template HTML file, an alternative notation exists for inserting a form, a question, a reply or helpful text:

proweb(Form)	inserts Form (i.e. one or more forms) at this point
?my_question	inserts my_question defined via proweb_question/2
??my_reply	inserts my_reply defined via proweb_post_reply/2
???my_help	inserts my_help defined via proweb_help/3

Such notation would be placed within user-defined *proweb_page/2* and *proweb_form/2* clauses:

```
proweb_page( [Form], [ …, proweb(Form), … ] ).

proweb_form( my_form,
             [ ?my_question,
               ??my_reply,
               ???my_help
             ]
           ).
```

The <PROWEB FORM> Element

Whenever ProWeb encounters a <PROWEB FORM> tag in an HTML template file, it will insert the form directly after the <PROWEB FORM> tag. The purpose of the <PROWEB FORM> tag is to give the developer total control over the positioning of the ProWeb-generated form within the HTML page.

Note that there is no <PROWEB FORM="..."> version; there is currently no way to name a form in ProWeb.

The Prolog term equivalent is to place *proweb(form)* within the second argument of a *proweb_page/2* clause.

The <PROWEB QUESTION="..."> Element

If ProWeb were to encounter a <PROWEB QUESTION="foo"> tag in an HTML template file, it would look for a *proweb_question/2* clause with 'foo' as its first argument in the Prolog source code; the tag would then be replaced by ProWeb's HTML equivalent of the clause.

The Prolog term equivalent to, say, <PROWEB QUESTION="myquestion"> is to place *?myquestion* within the second argument of a *proweb_form/2* clause.

The <PROWEB REPLY="..."> Element

If ProWeb were to encounter a <PROWEB REPLY="foo"> tag in an HTML template file, it would look for a fact with the name 'foo' in the Prolog database that had been asserted using *proweb_post_reply/2*; the tag would then be replaced by the fact's value.

The Prolog term equivalent to, say, <PROWEB REPLY="myreply"> is to place *??myreply* within the second argument of a *proweb_form/2* clause.

Note: A <PROWEB REPLY="..."> will not be replaced if it is contained within double quotes such as in <IMG SRC="<PROWEB REPLY="my_gif">">.

The <PROWEB HELP="..."> Element

If ProWeb were to encounter a <PROWEB HELP="foo"> tag in an HTML template file, it would look for a *proweb_help/3* clause with 'foo' as its first argument in the Prolog source code; the tag would then be replaced by ProWeb's HTML equivalent of the clause.

The Prolog term equivalent to, say, <PROWEB HELP="myhelp"> is to place *???myhelp* within the second argument of a *proweb_form/2* clause.

An alternative to defining a separate *proweb_help/3* clause for a question is to include two additional parameters, *help_button* and *help_string*, in the question definition itself:

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.
```

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [] ).

proweb_form( my_form,
          [ ?my_question,
            ???my_question
          ]
        ).

proweb_question( my_question,
             [ method      = input,
               type        = atom,
               help_button = `help`,
               help_string = `This is some help text`
             ]
           ).
```

The <PROWEB PAGENUMBER> Element

The element, <PROWEB PAGENUMBER>, found in an HTML template file would be replaced by ProWeb with the number of the current page in the conversation.

The <PROWEB VALUE="..."> Element

The element, <PROWEB VALUE="%2">, found in an HTML template file would be replaced by ProWeb with the first argument of the form embedded in the resulting page; similarly, the element, <PROWEB VALUE="%1"> would be replaced with the name of the form itself.

Note that there is no Prolog term equivalent.

An Example

The following program demonstrates how to use some of the above <PROWEB> elements:

```
main_goal :-
  proweb_post_reply( my_reply(1), `This is the 1st instance of this form`),
  proweb_post_reply( my_reply(2), `This is the 2nd instance of this form`),
  proweb_post_reply( my_reply(3), `This is the 3rd instance of this form`),
  proweb_send_form( my_form(1) ),
  proweb_send_form( my_form(2) ),
  proweb_send_form( my_form(3) ).

proweb_page( _, [] ).

proweb_form( my_form(Nth),
          [ include('html/template.htm') ]
        ).
```

```
proweb_question( my_question(Nth),
                 [ method = Method,
                   select = Choice
                 ]
               ) :-
  question_choice( Nth, Method, Choice ).

question_choice( 1, menubox,  [a,b,c]         ).
question_choice( 2, radio,    [car,bus,train]  ).
question_choice( 3, multibox, [crawl,walk,run] ).
```

The text of the HTML template file, TEMPLATE.HTM, is:

```
<HTML>
 <HEAD>
  <TITLE>
  </TITLE>
 </HEAD>
 <BODY>
  <P>The form used to create this page was <B><PROWEB VALUE="%1"></B>
  <P>This page features instance number <B><PROWEB VALUE="%2"></B> of this
form
  <P>The choice this time is between <B>
   <PROWEB QUESTION="my_question(%2)"></B>
  <P>The phrase this time is <B><PROWEB REPLY="my_reply(%2)"></B>
  <P><INPUT TYPE=SUBMIT>
 </BODY>
</HTML>
```

When executed, the above program will send the first instance of the form, my_form, to the client. In the course of expanding the template HTML file, ProWeb will replace:

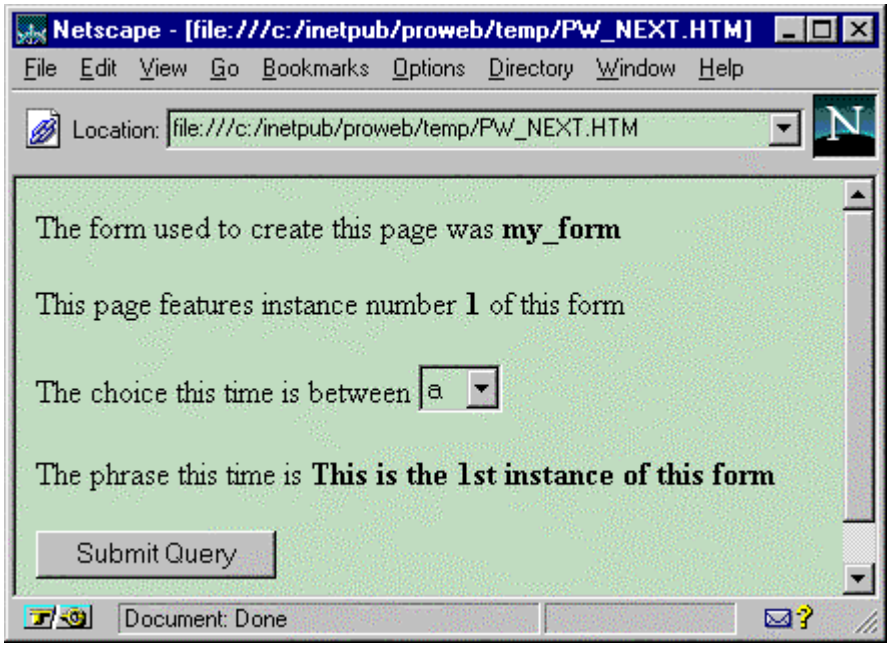
<PROWEB VALUE=”%1”> with the form’s name,

<PROWEB VALUE="%2"> with the instance of the form (i.e. the form’s first argument),

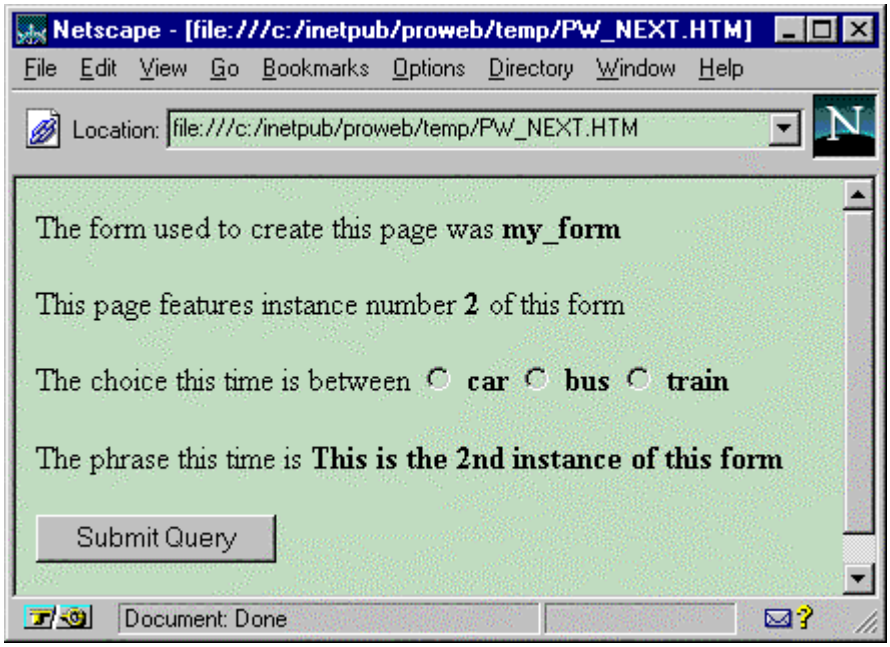
<PROWEB QUESTION="my_question(%2)> with the question having the same instance number as the form itself, and

<PROWEB REPLY="my_reply(%2)"> with the reply having the same instance as the form itself.

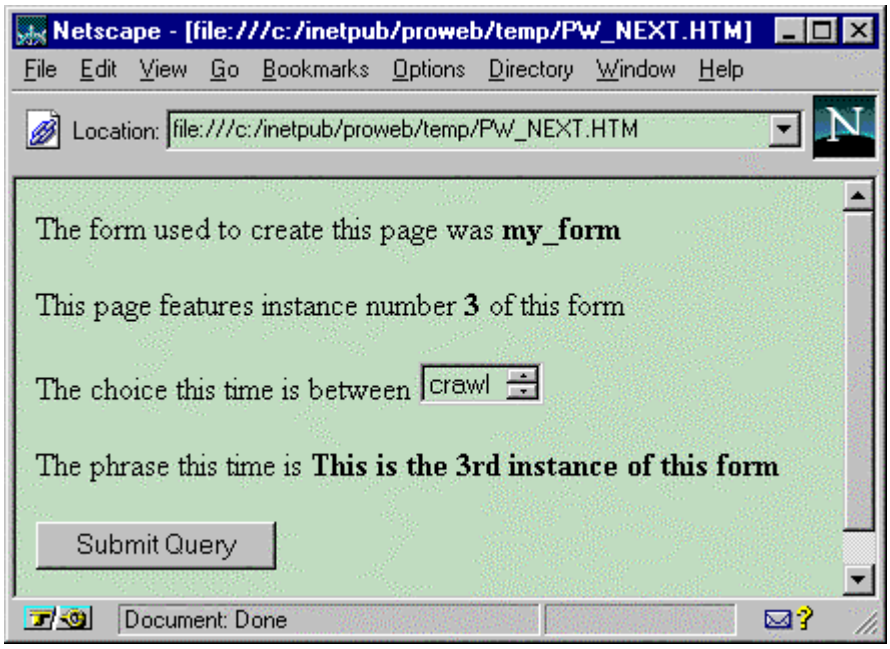
When *my_form(1)* is displayed, it will look like:



Submitting the first instance of *my_form* will display *my_form(2)*; this will look like:



Submitting *my_form(2)* will display the third instance of *my_form*; this will look like:



Chapter 15 - The Prolog Database

This chapter describes how ProWeb uses the Prolog database.

Introduction

It is normally the case that the memory occupied by a *WIN-PROLOG* application is only accessed by one user at a time; if they assert or retract anything, only they are aware of it. As a ProWeb-based application has many clients, the normal Prolog assert and retract predicates are best avoided as the actions of one client are 'visible' to all others.

To make assertions and retractions in a ProWeb-based application invisible to the other clients, the ProWeb toolkit provides the following predicates:

proweb_assert/1 to be used instead of *assert/1*.

proweb_asserta/1 to be used instead of *asserta/1*.

proweb_assertz/1 to be used instead of *assertz/1*.

proweb_call/1 calls ProWeb-asserted clauses.

proweb_dynamic/1 to be used instead of *dynamic/1*.

proweb_posted_reply/2 retrieves assertions made using *proweb_post_reply/2*.

*proweb_post_reply/2*assertion for use later as a reply within a form.

proweb_retract/1 to be used instead of *retract/1*.

proweb_retractall/1 to be used instead of *retractall/1*.

An Example Program

```
:- multifile( proweb_page/2 ).
:- multifile( proweb_form/2 ).

main_goal :-
  proweb_dynamic( my_fact/1 ),
  proweb_send_form( my_facts_form(none) ),
  proweb_assert( my_fact(green) ),
  proweb_send_form( my_facts_form(after_assert) ),
  proweb_asserta( my_fact(blue) ),
  proweb_send_form( my_facts_form(after_asserta) ),
  proweb_assertz( my_fact(red) ),
  proweb_send_form( my_facts_form(after_assertz) ),
  proweb_retract( my_fact(green) ),
  proweb_send_form( my_facts_form(after_retract) ),
  proweb_retractall( my_fact(_) ),
  proweb_send_form( my_facts_form(after_retractall) ).
```

```
proweb_page( my_facts_form(_), [] ).

proweb_form( my_facts_form(MyFact),
          [
            p @ `The following facts are currently in memory:`,
            ul @ MyFacts,
            input(type=submit)
          ]
        ) :-
 findall( MyFact,
        proweb_call( my_fact(MyFact) ),
        MyFacts
       ).
```

Definition for a proweb_unique_assert/1 Clause

As a ProWeb application is run from the beginning for each interaction, it is possible that certain facts you assert on one interaction are re-asserted during later interactions. To prevent this, you need a predicate that only allows something to be asserted once; a proweb_unique_assert/1 clause could be defined as:

```
proweb_unique_assert( X ) :-
  (      proweb_call(X)
  ->     true
  ;      proweb_assert(X)
  ).
```

Chapter 16 - Debugging

This chapter describes the various methods available to debug a ProWeb-based application.

Checking What's Running

If ProWeb fails to respond and you are using Windows NT, you can check the Windows NT Task Manager to see what ProWeb-related tasks are running. Only two tasks will ever appear - PROWEB.EXE and PROWEB.SYS. Some ProWeb problems can be cured by waiting for PROWEB.SYS to time out and then restarting.

Viewing ProWeb Settings And Environment Strings Via Your Browser

Once a ProWeb-based application has been deployed, the normal way to ascertain the state of an individual ProWeb setting would be to manually view the PROWEB.INI file; the following program allows such settings to be viewed via your web browser. It is possible to pick up environment strings and their values via a ProWeb application and display them within an HTML page, as the following example also shows:

```
main_goal :-
        env( EnvironmentStrings ),
        findall( (verbatim@Env = unencoded@Value),
                   member( (Env,Value), EnvironmentStrings ),
                   Envs
                 ),
        findall(
                (verbatim@String = verbatim@Value),
                proweb_string( String, Value ),
                Strings
               ),
        findall(
                (verbatim@Setting = verbatim@Value),
                proweb_setting( Setting, Value ),
                Settings
               ),
        findall(
                verbatim@File,
                source_file( File ),
                Files
               ),
        proweb_send_form( environment(Envs,Strings,Settings,Files) ).
```

```
proweb_page(
        [environment(_,_,_,_)],
        [
          html,
          head @ title @ verbatim @ `Environment`,
          body @ form(target='pws_demo_frame') @ proweb(form),
          /html
        ]
      ).

proweb_form(
        environment(Envs,Strings,Settings,Files),
        [
          h2 @ `ProWeb Strings`,
          p @ ol @ Strings,
          h2 @ `ProWeb Settings`,
          p @ ol @ Settings,
          h2 @ `Environment Variables`,
          p @ ol @ Envs,
          h2 @ `Source Files`,
          p @ ol @ Files
        ]
      ).
```

If required, such a facility could be built into your deployed ProWeb-based application, perhaps password protected.

The *env/1* predicate should be used with care in a ProWeb application as the environment strings are read when ***WIN-PROLOG*** (PROWEB.SYS) starts up and so may become out of date due to PROWEB.SYS "hanging around".

Tracing Your ProWeb Application

ProWeb possesses the ability to output trace information to a file named PROWEB.LOG in the conversation folder. To enable such output, set ***WIN-PROLOG***'s 'y' switch to an integer greater than 0 by adding the following to the COMMAND value in the PRO386W section of the PROWEB.INI file:

```
[PRO386W]
COMMAND=/y1
```

The output of such information is surpressed if ***WIN-PROLOG***'s 'y' switch is set to 0.

The contents of the PROWEB.LOG file can then be viewed in a text editor which supports Unicode files or in the ***WIN-PROLOG*** development environment.

In addition to ProWeb's own tracing output, this file can be used to capture output from your own ProWeb-based application whilst it is running. Should you wish to trace the execution of your ProWeb application, place one or more *proweb_trace/1* predicate calls at the appropriate points in your program's code. For each piece of information

you wish to output, simply place it as the first argument to a *proweb_trace/1* call at the appropriate point in your source code. The following program demonstrates:

```
:- proweb_trace( `File loading` ).

main goal :-
  proweb_trace( `Main goal starting` ),
  ...
  proweb_trace( `Main goal finished` ).

:- proweb_trace( `File loaded` ).
```

A Prolog listing could, for example, be output to PROWEB.LOG using:

```
...
listing ~> String,
proweb_trace( String ),
...
```

There is currently no global way to turn the ProWeb user tracing facility on or off; the setting of the 'y' switch has no effect on the ProWeb user tracing facility. For example, whenever *proweb_trace(Info)* is executed in your code, *Info* will always be output to PROWEB.LOG, regardless of the value of the 'y' switch.

Redefining The Built-In Forms

ProWeb possesses eight built-in forms which you will probably encounter from time to time. These can be user-redefined if required to aid debugging. They are described in their own chapter.

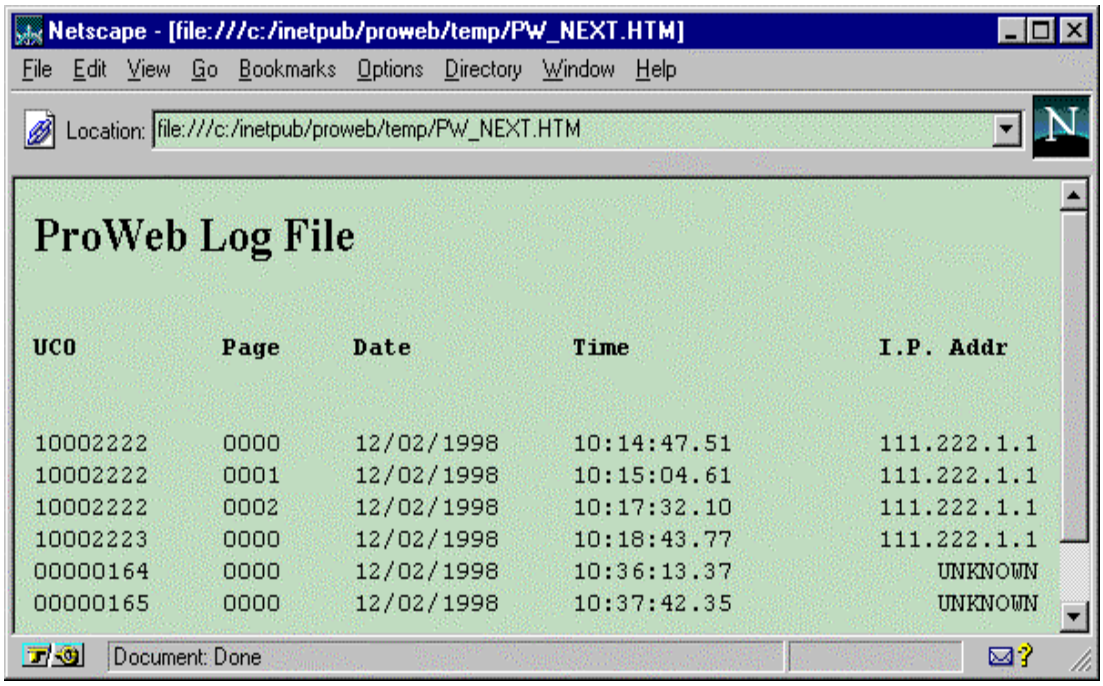
The PROWEB.LOG and TIMEOUT.LOG files

ProWeb maintains, within its conversation folder, three files, HITS.LOG, PROWEB.LOG and TIMEOUT.LOG.

In HITS.LOG ProWeb maintains a log of each interaction of each conversation. The information stored per interaction is the client's Unique Conversation Ordinal (UCO) automatically assigned by ProWeb, the page number within the conversation, the date and time of the interaction and the client's I.P. address. The following ProWeb-based program will display HITS.LOG within your web browser:

```
main_goal :-
  see( 'c:\inetpub\proweb\temp\hits.log' ),
  copy(65535,_) ~> LogDetails,
  seen,
  proweb_post_reply( log_details, LogDetails ),
  proweb_send_form( log_form ).

proweb_page( _, [] ).
```

```
proweb_form( log_form,
        [ h2 @ `ProWeb Log File`,
          pre,
          b @ `UCO       Page    Date        Time          I.P. Addr~M~J`,
          ??log_details,
          /pre
        ]
      ).
```



The resulting HTML page will look something like:

Such a facility will allow you to ascertain how many clients are visiting your web site and running your ProWeb-based application.

The value of the proweb setting, temp_timeout, specifies the number of seconds before ProWeb's default temp_timeout predicate is called. This predicate automatically deletes any file over an hour old and updates (i.e. appends to the end) the TIMEOUT.LOG file with the deletion details of each non-.LOG file in the ProWeb temporary directory. If the value of temp_timeout is low and the number of files in the temp directory high, the size of TIMEOUT.LOG can become extremely large. With an extreme number of files, one of the error messages #5 "Text space full" or #9 "Output space full" can be generated. Note that a frame-based ProWeb application generates more files in the ProWeb temporary directory than a non-frame ProWeb application.

These two files can become, over time, very large, although they should be truncated by ProWeb automatically in accordance with the temp_maxsize setting. Should you wish to delete them every time your ProWeb application's source code file is loaded, two directives like the following should do the trick:

```
:- initialization del('c:\inetpub\proweb\temp\proweb.log').
:- initialization del('c:\inetpub\proweb\temp\timeout.log').
```

Viewing *WIN-*PROLOG's Main And Console Windows

Normally, when running a ProWeb-based application on the same machine as the server, ***WIN-*PROLOG**'s Main and Console windows are invisible because it is being run in a different user's workspace. However, if you were to log onto the web server software using the same user that you logged onto Windows as, you will find that ***WIN-*PROLOG**'s Main and Console windows become visible. In use, all you need to do is minimize ***WIN-*PROLOG**'s Main window and return to the web browser. If you wish to programatically minimize ***WIN-*PROLOG**'s Main window, place the directive:

```
:- wshow( 0, 2 ).
```

in your source code.

Two ProWeb Debugging Predicates

There are two ProWeb debugging predicates that you might like to call within your source code – *proweb_debug/0* and *proweb_nodebug/0*. The predicate, *proweb_debug/0*, turns the debugging facility on, causing diagnostic information to be appended to each subsequent page sent to the client. The predicate, *proweb_nodebug/0*, turns the debugging facility off.

Running PROWEB.SYS Directly

If PROWEB.EXE is getting no response from PROWEB.SYS, you could try running PROWEB.SYS directly from the Windows command line. If Windows refuses to run a SYS file, you could try renaming PROWEB.SYS and PROWEB.OVL to, say, FRED.EXE and FRED.OVL. Hopefully, in the console window, you will see the cause of your problem. Warning: Running PROWEB.SYS in this way will prevent it from closing down automatically; you will need to quit PROWEB.SYS (or rather FRED.EXE) from the Windows Task Manager.

TRACE.PL

Supplied with ***WIN-*PROLOG** is a library file called TRACE.PL. This program can be used to trace your ProWeb application. You will need to have scroll lock enabled on the server.

```
:- ensure_loaded( library(trace) ).

:- multifile( proweb_page / 2 ).
:- multifile( proweb_form / 2 ).
```

```
main_goal :-
  Goal1 = write(fred),
  trace_goal( Goal1, single_solution, TraceLog1 ),
  Goal2 = member(X,[a,b,c]),
  trace_goal( Goal2, multiple_solutions, TraceLog2 ),
  proweb_send_form( my_trace_form(TraceLog1,TraceLog2) ).

proweb_page( my_trace_form(_,_), [] ).

proweb_form( my_trace_form(TraceLog1,TraceLog2),
          [ p @ `The trace of the single solution goal is as follows:`,
            pre @ TraceLog1,
            p @ `The trace of the multiple solution goal is as follows:`,
            pre @ TraceLog2
          ]
        ).

trace_goal( Goal, Mode, TraceLog ) :-
  fcreate( tracelog, [], -2, 1024, -2 ),
  (   Mode = single_solution
  -> !,
     ?( Goal, tracelog )
  ;   forall( ?( Goal, tracelog ), true )
  ),
  input(OldIn),
  input(tracelog),
  inpos( 0 ),
  copy( 1024, Copied ) ~> TraceLog,
  input(OldIn),
  fclose( tracelog ).
```

The ouput of this example is as follows:

```
The trace of the single solution goal is as follows:

C write(fred)
S write(fred)

The trace of the multiple solution goal is as follows:

C member(_34432,[a,b,c])
S member(a,[a,b,c])
S member(b,[a,b,c])
S member(c,[a,b,c])
F member(_34432,[a,b,c])
```

Obtaining Information about your Version of ProWeb

If you should ever need to contact LPA concerning a problem with ProWeb, the version number of each piece of software (ProWeb and *WIN-PROLOG*) you are using must be provided. Executing *proweb_string(version, X)* from the *WIN-PROLOG* command line will obtain ProWeb's version number:

```
?- proweb_string( version, X ). <enter>
X = `1.42`
```

Additional information about ProWeb can also be obtained with *proweb_string/2*; alternative values for the first argument are *title*, *owner*, *copyright*, *build* and *author*.

Direct Execution From The *WIN-*PROLOG Command Line

Certain ProWeb predicates can be tested in isolation by executing them directly from the ***WIN-*PROLOG** command line. You will, of course, need to be using PROWEB.OVL rather than PRO386W.OVL. One such predicate is *html_print/1*:

```
?- Type=submit, Text=`Click Me!`, html_print( input( type=Type, value=Text )
).<enter>
<INPUT TYPE=SUBMIT VALUE="Click Me!">Type = submit ,
Text = `Click Me!`
```

Killing Off PROWEB.SYS From The Task Manager

Should you find that PROWEB.SYS has completely died and refuses to remove itself from memory, the only course of action is to highlight the PROWEB.SYS process in the Windows Task Manager dialog and then click on the "End Process" button. If you are logged on as a different user to the one that launched PROWEB.SYS, Windows will not let you end the process. You will need to log off as the current user and log back on again as the user that launched PROWEB.SYS; this user is normally "IUSR_<some_name>".

ProWeb's Execution Mechanism

Assumptions

This text makes the assumption that MAX FORMS PER PAGE is set to 0 (zero).

ProWeb Normal Behaviour

Suppose you have the following program:

```
main_goal :-
  proweb_send_form( first_form  ),
  beep( 440, 250 ),
  proweb_send_form( second_form ).
```

When launched, a beep will sound and then the first page will appear in the client's browser, indicating that its execution is as follows:

```
?           main_goal :-
succeed       proweb_send_form( first_form  ),
succeed       beep( 440, 250 ),
?             proweb_send_form( second_form ).
```

So how do we know whether *proweb_send_form(second_form)* is succeeding or failing? Suppose we rewrote the program as follows:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ),
  beep(440,250).
```

When launched, the first page will appear in the client's browser but a beep will NOT sound, indicating that its execution is as follows:

```
fail                main_goal :-
succeed               proweb_send_form( first_form  ),
fail                  proweb_send_form( second_form ),
never reached         beep( 440, 250 ).
```

The next answer we need to know is does it matter to ProWeb if *main_goal/0* succeeds or fails? If we rewrite our program as follows:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ).

main_goal :-
  beep(440,250).
```

When launched, a beep will sound and then the first page will appear in the client's browser, indicating that its execution is as follows:

```
fail        main_goal :-
succeed       proweb_send_form( first_form  ),
fail          proweb_send_form( second_form ).

succeed     main_goal :-
succeed       beep(440,250).
```

In this example, the second clause of *main_goal/0* is succeeding whereas it was failing in the earlier examples. So as far as ProWeb is concerned, it doesn't matter whether *main_goal/0* succeeds or fails.

Forcing Failure

Once we've told ProWeb we want to send a form with the line, *proweb_send_form(first_form),* we need a way to stop it executing the code between one *proweb_send_form/1* and the next *proweb_send_form/1*. One way to do this is to test whether a form that has been sent to the client has been returned to the server:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_returned_form( first_form ),
  beep( 440, 250 ),
  proweb_send_form( second_form ).
```

When launched, the first page will appear in the client's browser but a beep will NOT sound, indicating that its execution is as follows:

```
fail            main_goal :-
succeed           proweb_send_form( first_form  ),
fail              proweb_returned_form( first_form ),
never reached     beep( 440, 250 ),
never reached      proweb_send_form( second_form ).
```

There is also a *proweb_unreturned_form/1* which succeeds if a form has been sent (or earmarked for sending) to a client but not yet returned.

In addition to checking whether a sent form has been returned, we can also check whether an answer to an asked ProWeb-generated question is known:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_returned_answer( first_question, Answer ),
  beep( 440, 250 ),
  proweb_send_form( second_form ).
```

For this to make sense, *first_question* will have had to have been asked on *first_form*.

If we submit the first page, a beep will sound and the second page will appear in the client's browser, indicating that its execution is as follows:

```
succeed         main_goal :-
succeed           proweb_send_form( first_form  ),
succeed           proweb_returned_form( first_form ),
succeed           beep( 440, 250 ),
succeed           proweb_send_form( second_form ).
```

As you can see, *proweb_send_form(first_form)* succeeded but the first form was not re-sent, only the second. This is because *proweb_send_form/1* will always succeed for a form that ProWeb remembers it has already sent.

There is another ProWeb predicate like *proweb_send_form/1* called *proweb_send_unique_form/1*. Whereas *proweb_send_form/1* always succeeds, *proweb_send_unique_form/1* succeeds the first time (when the form is actually sent) and fails on every subsequent execution.

Allowing ProWeb To Send More Than One Form On a Page

Having said all this, we may actually want ProWeb to send as many forms as it can on the same page. Suppose we had the following program:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ),
  beep(440,250),
  proweb_send_form( third_form ).
```

```
proweb_friends( first_form, second_form ).
```

When launched, a beep will sound and then the first and second forms will be present in the page which appears in the client's browser, indicating that its execution is as follows:

```
fail        main_goal :-
succeed       proweb_send_form( first_form  ),
succeed       proweb_send_form( second_form ),
succeed       beep(440,250),
fail          proweb_send_form( third_form ).

succeed     proweb_friends( first_form, second_form ).
```

Due to the *proweb_friends(first_form,second_form)* fact allowing the first form and the second form to be sent together on the same page, the line, *proweb_send_form(second_form)*, succeeded (whereas previously, it would have failed).

Sending a Form Again

Using *proweb_resend_form/1* instead of *proweb_send_form/1* allows us to resend a form. Suppose you have the following program:

```
main_goal :-
  proweb_resend_form( first_form  ),
  beep( 440, 250 ),
  proweb_send_form( second_form ).
```

When launched, a beep will sound and then the first form will appear in the client's browser. When submitted, a beep will sound and the first form will again appear in the client's browser. Because the first form and the second form are not 'friends' the line, *proweb_send_form(second_form)*, always fails. ProWeb is unable to send the second form because it is always sending or resending the first form. Its execution is therefore always as follows:

```
fail        main_goal :-
succeed       proweb_resend_form( first_form  ),
succeed       beep( 440, 250 ),
fail          proweb_send_form( second_form ).
```

The same behaviour to the above program can be achieved using *proweb_send_form/1* and *proweb_forget_forms_sent/0* together:

```
main_goal :-
  proweb_send_form( first_form  ),
  beep( 440, 250 ),
  proweb_forget_forms_sent,
  proweb_send_form( second_form ).
```

When launched, a beep will sound and then the first form will appear in the client's browser. When submitted, a beep will sound and the first form will again appear in the client's browser. Because execution is reaching the *proweb_forget_forms_sent* line, ProWeb is no longer remembering that it has already sent (or earmarked for sending) the first form.

```
fail          main_goal :-
succeed         proweb_send_form( first_form  ),
succeed         beep( 440, 250 ),
succeed         proweb_forget_forms_sent,
fail            proweb_send_form( second_form ).
```

Running An Entire Conversation Again

The correct use of *proweb_forget_forms_sent/0* is to allow an entire conversation to be run again:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ),
  proweb_forget_forms_sent.
```

When launched, the first form will appear in the client's browser. When submitted, the second form will appear in the client's browser. When submitted, the first form will again appear in the client's browser. When submitted, the second form will again appear in the client's browser. When submitted, the first form will appear for the third time in the client's browser. When submitted, the second form will appear for the third time in the client's browser.

Parameterising a Form

ProWeb allows us to give one or more parameters to a form's name. Whereas *proweb_send_form(first_form)* will always succeed but will only send the form once, *proweb_send_form(first_form(N))* will always send the first form so long as N is different in each case:

```
main_goal :-
  time( 0, N ),
  proweb_send_form( first_form(N) ).
```

When launched, first_form(N) will appear in the page sent to the client's browser. When submitted, first_form(N+1) will appear in the page sent to the client's browser.

Chapter 17 – ProWeb's Built-In Forms

ProWeb possesses eight built-in forms which you will probably encounter from time to time. These can be user-redefined if required to aid debugging, give diagnostic information to the client or give your ProWeb application a consistent look and feel.

The proweb_terminate_form(true) Form

The form, *proweb_terminate_form(true)*, is generated by ProWeb and sent to the client whenever the main goal terminates with success.

The following program demonstrates how to cause the *proweb_terminate_form(true)* form to be sent and then redefines the form:

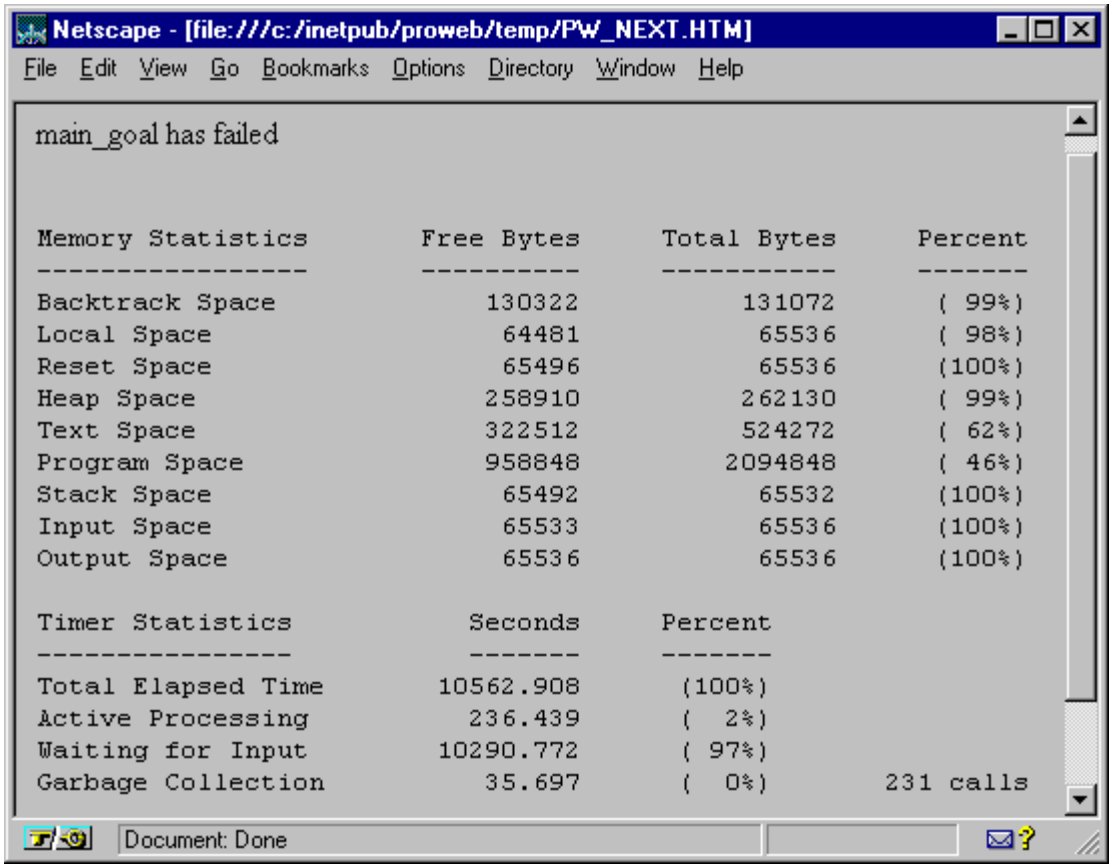
```
main_goal.

proweb_page( _, [] ).

proweb_form( proweb_terminate_form( true ), [ `Your main goal has terminated
with success` ] ).
```

The proweb_terminate_form(fail) Form

The form, *proweb_terminate_form(fail)*, is generated by ProWeb and sent to the client whenever the main goal terminates with failure.

The following program demonstrates how to cause the *proweb_terminate_form(fail)* form to be sent and then redefines the form to output the name of the main goal and Prolog's memory usage:

```
main_goal :-
  fail.

proweb_page( _, [] ).

proweb_form( proweb_terminate_form(fail),
                [ MainGoal,
                  ` has failed`,
                  pre,
                  Stats,
                  /pre
                ]
              ) :-
  proweb_setting(main_goal, MainGoal ),
  (statistics ~> Stats ).
```



The proweb_terminate_form(abort) Form

The form, *proweb_terminate_form(abort)*, is generated by ProWeb and sent to the client whenever the application aborts.

The following program demonstrates how to cause the *proweb_terminate_form(abort)* form to be sent and then redefines the form:

```
main_goal :-
  abort.

proweb_page( _,
             []
           ).

proweb_form( proweb_terminate_form( abort ),
             [ `Your program has aborted`
             ]
           ).
```

The proweb_terminate_form(error(When,Error,Goal)) Form

The form, *proweb_terminate_form(error(When,Error,Goal))*, is generated by ProWeb and sent to the client whenever the main or post (When) goal terminates with an Error for a particular Goal.

The following program demonstrates how to cause the *proweb_terminate_form(error(When,ErrorNumber,Goal))* form to be sent and then redefines the form to return details of the error which has just occurred, the code for the predicate in which the error occurred, the contents of Prolog's console window, a note of where Prolog thinks its main directory is, a list of all source files loaded together with a capture of the current listing/0 output:

```
main_goal :-
  an_undefined_predicate.

proweb_page( _,
             []
           ).
```

```
proweb_form( proweb_terminate_form(error(When,ErrorNumber,Goal)),
        [ h3 @ `Sorry, an error has occured within this ProWeb application`,
          table(width='100%',border=3),
          tr,
          td,`Predicate name: `,b,When,/b,/td,
          td,
          `Prolog error number: `,b,ErrorNumber,` (`,ErrorMessage,`)`,/b,
          /td,
          td,`Failed subgoal: `,b,Goal,/b,/td,
          /tr,
          tr,
          td(colspan=3),`Predicate listing:`,pre,WhenString,/pre,/td,
          /tr,
          tr,
          td(colspan=3),`Prolog Console Window:`,b,pre,Console,/pre,/b,/td,
          /tr,
          tr,
          td(colspan=3),
          `The main directory of Prolog is `,b,AbsFileName,/b,
          /td,
          /tr,
          tr,
          td(colspan=3),`Source files loaded:`,p,b,ul @ Files,/b,/td,
          /tr,
          tr,
          td(colspan=3),
          `Prolog listing:`,
          b,pre,ListingString,/pre,/b,
          /td,
          /tr,
          /table
        ]
      ) :- error_message( ErrorNumber, ErrorMessage ),
           listing(When) ~> WhenString,
           absolute_file_name( prolog(''), AbsFileName ),
           findall( File,
                    source_file( File ),
                    Files
                  ),
           listing ~> ListingString,
           wtext((1,1),Console).
```

The proweb_terminate_form(javascript_disabled) Form

There is potential for returning bad data if JavaScript had been disabled at the client end. The error message form, *proweb_terminate_form(javascript_disabled)*, is generated by ProWeb and sent to the client whenever JavaScript has been disabled at the client end.

The following program demonstrates how to cause the *proweb_terminate_form(javascript_disabled)* form to be sent and then redefines the form; you will, of course, need to temporarily disable Javascript in your client's web browser.

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ).

proweb_page( _, [] ).

proweb_form( first_form,
          [ p @ `This is the first form`,
            p @ ?enter_an_integer,
            p @ input(type=submit)
          ]
        ).

proweb_form( second_form,
          [ `This is the second form`
          ]
        ).

proweb_form( proweb_terminate_form(javascript_disabled), [ `Please turn
Javascript on!`] ).

proweb_question( enter_an_integer, [ method=input, type = integer ] ).
```

The proweb_timeout_form(client) Form

The form, *proweb_timeout_form(client)*, is generated by ProWeb and sent to the client whenever the Client response exceeds the ProWeb temp_timeout setting. **Note: This built-in form can not currently be user-redefined.**

The proweb_timeout_form(server) Form

The form, *proweb_timeout_form(server)*, is generated by ProWeb and sent to the client whenever the time taken for the main goal to terminate exceeds the ProWeb query_timeout setting.

The following program demonstrates how to cause the *proweb_timeout_form(server)* form to be sent and then redefines the form; to get a quicker response, set the query_timeout setting to a lower value.

```
main_goal :-
  main_goal.

proweb_page( _, [] ).
```

```
proweb_form( proweb_timeout_form( server ),
          [ p @ `Your query took too long!`,
            p,
            `Please increase your query_timeout setting, which is currently set to `,
            QueryTimeOutSetting,
            /p
          ]
        ) :-
  proweb_setting( query_timeout, QueryTimeOutSetting ).
```

The proweb_undefined_form(Form) Form

The form, *proweb_undefined_form(Form)*, is generated by ProWeb and sent to the client whenever an application Form is sent by the main program for which there is no legitimate definition. **Note: This form is currently out of use.**

Trying To Recover From a ProWeb Error

Depending on the nature of the error suffered by ProWeb, it is sometimes possible to recover from the ProWeb error and continue with the same conversation. The following code creates a hyperlink that can be placed within a ProWeb error page. Your ProWeb program will need to act correctly upon being sent such a URL; you will probably need to change the 'action=recover' attribute-value pair to something else.

```
generate_recovery_hyperlink( RecoveryHyperlink ) :-
  proweb_setting( base_url, BaseURL ),
  proweb_data_uco( UCO ),
  proweb_data_page( Page ),
  ( write( BaseURL ),
    write(`knowgrid.exe?proweb_dotexe_added_to=form&proweb_data_uco=[` ),
    write(UCO),
    write(`']&proweb_data_page=[` ),
    write( Page ),
    write(`]&action=recover`)
  ) ~> RecoveryURL,
  RecoveryHyperlink = [p,a(href=RecoveryURL),`Click here to try to
recover`,/a,/p].
```

Chapter 18 - Execution Differences

This chapter documents the differences in execution between a *WIN-PROLOG*-only application and a *WIN-PROLOG*/ProWeb one.

The env/1 Predicate

The *env/1* predicate should be used with care in a ProWeb application as the environment strings are read when *WIN-PROLOG* (PROWEB.SYS) starts up and so may become out of date due to PROWEB.SYS "hanging around".

Directives

There is a difference in the way *WIN-PROLOG* in isolation and ProWeb/*WIN-PROLOG* execute directives as these four examples show:

1st ProWeb example:

```
:- proweb_trace( `This is the first line in the file` ).

hello :-
  proweb_trace( `Starting main_goal/0 (1st clause)` ),
  proweb_trace( `Ending main_goal/0 (1st clause)` ).

hello :-
  proweb_trace( `Starting main_goal/0 (2nd clause)` ),
  proweb_trace( `Ending main_goal/0 (2nd clause)` ).

:- proweb_trace( `This is the last line in the file` ).
```

PROWEB.LOG will contain:

```
This is the first line in the file
This is the last line in the file
Starting main_goal/0 (1st clause)
Ending main_goal/0 (1st clause)
```

Note: On loading the file, both directives have been executed and then the main goal has started and finished.

2nd ProWeb Example:

```
:- proweb_trace( `This is the first line in the file` ).

:- ensure_loaded( rubbish ).

hello :-
  proweb_trace( `Starting main_goal/0 (1st clause)` ),
  proweb_trace( `Ending main_goal/0 (1st clause)` ).
```

```
hello :-
  proweb_trace( `Starting main_goal/0 (2nd clause)` ),
  proweb_trace( `Ending main_goal/0 (2nd clause)` ).

:- proweb_trace( `This is the last line in the file` ).
```

PROWEB.LOG will contain:

```
This is the first line in the file
This is the last line in the file
Starting main_goal/0 (1st clause)
Ending main_goal/0 (1st clause)
```

Note: On loading the file, both directives have been executed and then the main goal has started and finished. In *WIN-PROLOG*, the line, *:-ensure_loaded(rubbish).*, would cause the file to be aborted when rubbish could not be found. With ProWeb, this line silently fails.

3rd ProWeb Example:

```
:- initialization proweb_trace( `This is the first line in the file` ).

:- initialization ensure_loaded( rubbish ).

hello :-
  proweb_trace( `Starting main_goal/0 (1st clause)` ),
  proweb_trace( `Ending main_goal/0 (1st clause)` ).

hello :-
  proweb_trace( `Starting main_goal/0 (2nd clause)` ),
  proweb_trace( `Ending main_goal/0 (2nd clause)` ).

:- initialization proweb_trace( `This is the last line in the file` ).
```

PROWEB.LOG will contain:

```
This is the first line in the file
Starting main_goal/0 (1st clause)
Ending main_goal/0 (1st clause)
```

Note: On loading the file, the first directive is executed and then the main goal starts and finishes. The line ":- initialization ensure_loaded(rubbish).", although silently failing, causes all later 'initialization' directives (executed after the whole file has been loaded) to be ignored, hence 'This is the last line in the file' not appearing in PROWEB.LOG.

4th ProWeb Example:

```
:- initialization proweb_trace( `This is the first line in the file` ).

:- ensure_loaded( rubbish ).
```

```
hello :-
  proweb_trace( `Starting main_goal/0 (1st clause)` ),
  ensure_loaded( rubbish ),
  proweb_trace( `Ending main_goal/0 (1st clause)` ).

hello :-
  proweb_trace( `Starting main_goal/0 (2nd clause)` ),
  proweb_trace( `Ending main_goal/0 (2nd clause)` ).

:- initialization proweb_trace( `This is the last line in the file` ).
```

PROWEB.LOG will contain:

```
This is the first line in the file
This is the last line in the file
Starting main_goal/0 (1st clause)
```

Note: On loading the file, the three directives are executed and then the main goal starts. The line ":- ensure_loaded(rubbish)." silently fails. The line "ensure_loaded(rubbish)," in hello/0, causes an error HTML page to be sent to the client, hence hello/0 never finishes; it does NOT silently fail as the second clause of hello/0 is never called.

Using The unknown_predicate_handler/2 Predicate

If you want ***WIN-PROLOG*** to handle a particular unknown predicate in a different way from the norm, you would assert an *unknown_predicate_handler/2* clause into the Prolog database; i.e.:

```
unknown_predicate_handler( an_unknown_predicate, fail ).
```

This would make a call to an_unknown_predicate silently fail rather than generate error 20 - Predicate not defined. In ProWeb, any applicable unknown_predicate_handler/2 clause is ignored and error 20 - Predicate not defined generated anyway.

The switch/2 Predicate

WIN-PROLOG's *switch/2* predicate is not applicable to a ProWeb program. Although a line such as switch(b,4096) will succeed in a ProWeb-based application, it will have no effect on ***WIN-PROLOG***'s settings in such an environment.

Chapter 19 - Advanced ProWeb Techniques

This chapter covers a number of advanced techniques that you may find useful.

Passing Information Out To The URL Line And Back In Again

With ProWeb, it is possible to pass information out embedded within the next HTML page in such a way that it is returned to your ProWeb application when the page is submitted.

If you placed the following line in a *proweb_form/2*:

```
input(type=submit,name=button_just_clicked(MyInformation),value='Click Me!')
```

it would generate a submit button containing the text 'Click Me!'. When you submit the page 'button_just_clicked(<the value of MyInformation>)' would be returned to ProWeb. If MyInformation was instantiated to the integer 1, 'button_just_clicked(1)' would have been returned.

You can then pick up such information via *proweb_returned_input/2*:

```
proweb_returned_input( button_just_clicked(MyInformation), 'Click Me!' )
```

You might need to do some type conversion when retrieving information using this method as, for example, the atom '1' will be returned as the integer 1.

Detecting Which Submit Button within a Form was Clicked

There may be occasions when you require not only two or more submit buttons to be present within a form but you also need to know which of the buttons was used to submit the form.

This first example program shows a form containing a 'Yes' button and a 'No' button; a client's click on one of these buttons could represent their answer to a question your application has asked:

```
main_goal :-
 ask_question(1).

ask_question(Number) :-
 proweb_send_form( question_form(Number) ),
 (  proweb_returned_input( question(Number), Answer )
 -> proweb_assert( question(Number,Answer) )
 ;  true
 ),
 Number1 is Number + 1,
 ask_question(Number1).

proweb_page( _, [] ).
```

```
proweb_form( question_form(Number),
        [ input(type=submit,name=question(Number),value='Yes'),
          input(type=submit,name=question(Number),value='No')
        ]
      ).
```

When submitting the page containing the first question, *question(1)=<answer>* will have been returned. When *proweb_returned_input/2* is executed, it will succeed as the question it is looking for is present. When the page containing the second question is submitted, *question(1)=<answer>,* previously submitted, is both lost from the current submission and from Prolog's internal database as ProWeb does not remember information obtained with *proweb_return_input/2*, unlike *proweb_returned_answer/2*,. As it is likely that such answers will want to be remembered later on in the conversation, *proweb_assert/1* is used to assert the same into the Prolog database. The *proweb_returned_input/2* call itself is placed within an if-then-else structure where the 'else' part, only reached when the submission does not contain the required question, simply succeed. The number is then incremented by one and the next question asked.

After three pages, the Prolog internal database would contain something like:

```
proweb_data_fact(1,question(1),'Yes').
proweb_data_fact(2,question(2),'No').
proweb_data_fact(3,question(3),'Yes').
```

As an aside, instead of Number as the first parameter to question, you could have used the actual text of the question.

This next example program creates an HTML page containing one form which itself contains two submit buttons. The text displayed in each button indicates the number of times it has been clicked upon. As you click on one or other of the buttons, a new page will be generated 'on the fly' with the displayed value of the button clicked upon increased by one:

```
main_goal :-
        process_pressed_button,
        proweb_resend_form(main).

process_pressed_button :-
        proweb_returned_input( firstbutton, _ ),
        proweb_call( buttonvalue(firstbutton,ButtonValue) ),
        increase_button_value(firstbutton,firstbuttonvalue,ButtonValue),
        set_button_value( secondbutton, secondbuttonvalue ).

process_pressed_button :-
        proweb_returned_input( secondbutton, AtomButtonValue ),
        number_atom( ButtonValue, AtomButtonValue ),
        increase_button_value( secondbutton, secondbuttonvalue, ButtonValue ),
        set_button_value( firstbutton, firstbuttonvalue ).
```

```
process_pressed_button :-
        proweb_post_reply( firstbuttonvalue, 0 ),
        proweb_asserta( buttonvalue(firstbutton,0) ),
        proweb_post_reply( secondbuttonvalue, 0 ),
        proweb_asserta( buttonvalue(secondbutton,0) ).

increase_button_value( ButtonName, ButtonValueName, ButtonValue ) :-
        NewButtonValue is ButtonValue + 1,
        proweb_post_reply( ButtonValueName, NewButtonValue ),
        proweb_retract( buttonvalue(ButtonName,_) ),
        proweb_asserta( buttonvalue(ButtonName,NewButtonValue) ).

set_button_value( ButtonName, ButtonValueName ) :-
        proweb_call( buttonvalue(ButtonName,ButtonValue) ),
        proweb_post_reply( ButtonValueName, ButtonValue ).

proweb_page( main, [] ).

proweb_form( main, include('2button.htm') ).
```

The text of 2BUTTON.HTM is as follows:

```
<HTML>
   <HEAD><TITLE></TITLE></HEAD>
   <BODY>
      <FORM METHOD=POST>
      <INPUT TYPE=SUBMIT NAME="firstbutton"
         VALUE="<PROWEB REPLY=firstbuttonvalue>">
      <INPUT TYPE=SUBMIT NAME="secondbutton"
         VALUE="<PROWEB REPLY=secondbuttonvalue>">
      </FORM>
   </BODY>
</HTML>
```

The program shows that the current value of a button can be ascertained in one of two ways. The current value of the button clicked upon can be obtained from a *proweb_returned_input/2* call; Note that as it is returned in the second argument as an atom it needs to be converted, for the above example, to an integer.

For all other buttons, no such information is returned in the submission, and so must be obtained via a previously asserted fact.

Immediately prior to a page being sent, the relevant fact in Prolog's internal database is updated via the *proweb_asserta/1* call to reflect the new value of the clicked button; During the next interaction, such a value can be retrieved via *proweb_call/1*.

Success or Failure upon Sending Unique Forms to a Client

One ProWeb predicate we have used quite extensively is *proweb_send_form/1*; this predicate will send a form to a client, but only if the form has not already been sent. A similar, but as yet unmentioned, ProWeb predicate is *proweb_send_unique_form/1*;

like *proweb_send_form/1*, this sends a form to a client if it has not already been sent, but unlike *proweb_send_form/1*, it will fail if the form has been sent before.

By way of example, take a look at the following program:

```
main_goal :-
        proweb_send_form(first_form).

proweb_page( [first_form], [] ).

proweb_form( first_form, [input(type=submit)] ).
```

When launched from a suitable HTML page, the above program will return its first page; this will contain just a submit button. When this page is submitted by the client, ProWeb will then send its default "Main program terminated with success!" page. If you were to change *proweb_send_form(first_form)* to *proweb_send_unique_form(first_form)*, you would find that it fails following submission of the first page, causing ProWeb to issue its default "Main program terminated with failure!" page.

Checking What has been Sent To and Received From a Client

You should now be quite familiar with the ProWeb predicate *proweb_returned_form/1*; to recap, this predicate tests whether or not a form has been sent to a client and subsequently returned. A similar, but as yet unmentioned, ProWeb predicate is *proweb_unreturned_form/1*; this tests whether or not a form has been sent to a client but not yet returned. As an example, take a look at the following program:

```
main_goal :-
        proweb_send_form(first_form),
        proweb_unreturned_form(first_form),
        proweb_resend_form(second_form).

main_goal:-
        proweb_send_form(third_form).

proweb_page( _, [] ).

proweb_form(    first_form,
                        [       `Submit first page: `,
                                input(type=submit),
                                p
                        ]
                ).

proweb_form(    second_form,
                        [       `Submit second page: `,
                                input(type=submit)
                        ]
                ).
```

```
proweb_form(    third_form,
                    [   `Submit third page: `,
                        input(type=submit)
                    ]
                ).

proweb_friends( first_form, second_form).
```

When *main_goal/0* is executed, *first_form* will be ear-marked for sending to the client; continuing to execute, *proweb_unreturned_form(first_form)* will then succeed as the ProWeb-asserted fact stating that the page has been sent to the client (but not returned) is now in Prolog's internal database. Continuing to execute further, *proweb_resend_form(second_form)* will also succeed; due to the presence of a *proweb_friends/2* clause for the two forms, ProWeb will ear-mark *second_form* for sending to the client as well. With *main_goal/0* having now succeeded, an HTML page containing the two forms will be sent to the client. Upon submitting this page, *proweb_send_form(first_form)* in *main_goal/0* succeeds again but *proweb_unreturned_form(first_form)* fails as the form has now been returned. Execution then backtracks to the second *main_goal/0* resulting in *third_form* being sent to the client.

Shutting ProWeb Down

Should you ever need to stop PROWEB.SYS, execute the following URL:

```
http://<server_name>/proweb/proweb.exe?!+001
```

where <server_name> is your server name. Executing the above URL will immediately kill the instance of PROWEB.SYS identified as '001' running on the server. If successfully shut down, a ProWeb (REX:001:0) error message page will be returned.

Redirecting a Form To Something Other Than PROWEB.EXE

ProWeb was never designed to generate a form that, when submitted, went to something other than PROWEB.EXE. However, the following example shows you a technique that may prove useful if this is something you need to do; this program generates an HTML page containing two forms - the first is the normal ProWeb one whilst the second is a user-defined form that, when submitted, goes to whatever you want.

```
:- multifile( proweb_page / 2 ).
:- multifile( proweb_form / 2 ).

main_goal :-
  proweb_send_form( multi_form ).
```

```
proweb_page( multi_form,
  [ form,
    proweb(multi_form),
    /form,
    form(method=post,action='fred.exe'),
    input(type=submit,value='Submit User Defined Form'),
    /form
  ]
  ).

proweb_form( multi_form, input(type=submit,value='Submit ProWeb Form') ).
```

The source code of the generated HTML page is as follows:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="/ProWeb/PROWEB.exe">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[1028]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<PROWEB multi_form>
<INPUT TYPE=SUBMIT VALUE="Submit ProWeb Form">
</FORM>
<FORM METHOD=POST ACTION="fred.exe">
<INPUT TYPE=SUBMIT VALUE="Submit User Defined Form">
</FORM>
</BODY>
</HTML>
```

Positioning ProWeb's Form To Maintain Your Web Site's Look and Feel

Most web sites have a similar 'look and feel' applied to all its pages. Let's now see how such a 'look and feel' could be applied to the 'Hello World!' example given much earlier. Within the *proweb_page/1* clause, we can instruct ProWeb to use a particular HTML page on disk as a template when constructing the page to send to the client; such a page might have been developed using Microsoft FrontPage or Adobe PageMill:

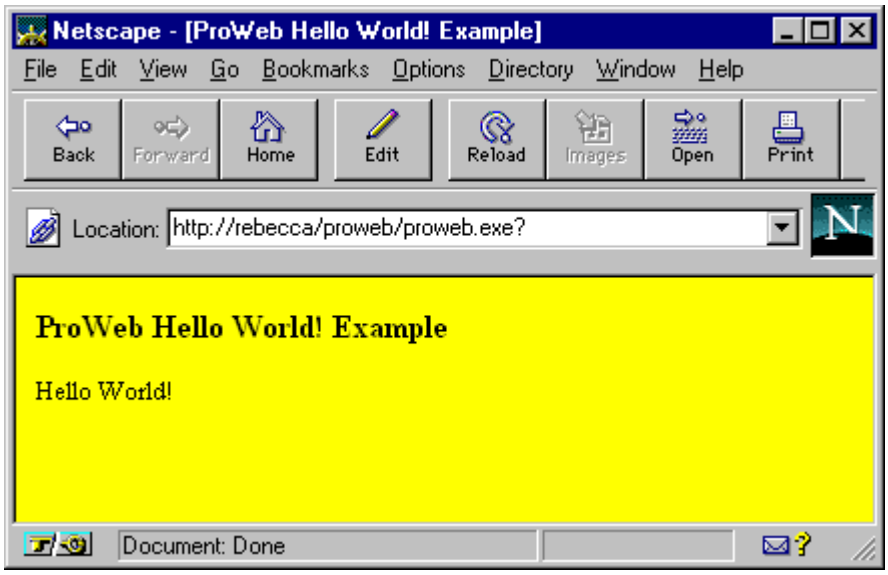
```
proweb_page( hello_form, [include('html/hw1.htm')] ).
```

The above line states that when ProWeb comes to construct the HTML page in which to embed *hello_form*, it should use the HTML code from the file, HW1.HTM, in the HTML directory. The HTML source code within HW1.HTM is:

```
<HTML>
 <HEAD>
  <TITLE>ProWeb Hello World! Example</TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
```

```
  <H3>ProWeb Hello World! Example</H3>
 </BODY>
</HTML>
```

Click on the "Run ProWeb Application" button in your ProWeb 'launch page'. ProWeb will be executed and whatever program is in PW_DEMO.PL will be loaded and run. Your web browser should now be showing:



The source code of this ProWeb-generated page is:

```
<HTML>
 <HEAD>
  <TITLE>ProWeb Hello World! Example</TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <H3>ProWeb Hello World! Example</H3>
   Hello World!
  </FORM>
 </BODY>
</HTML>
```

indicating that ProWeb has successfully managed to embed *hello_form* into the page definition code obtained from HW1.HTM. ProWeb has not altered HW1.HTM in any way; the page now shown in your web browser has been created by ProWeb 'on the fly' and sent to the client.

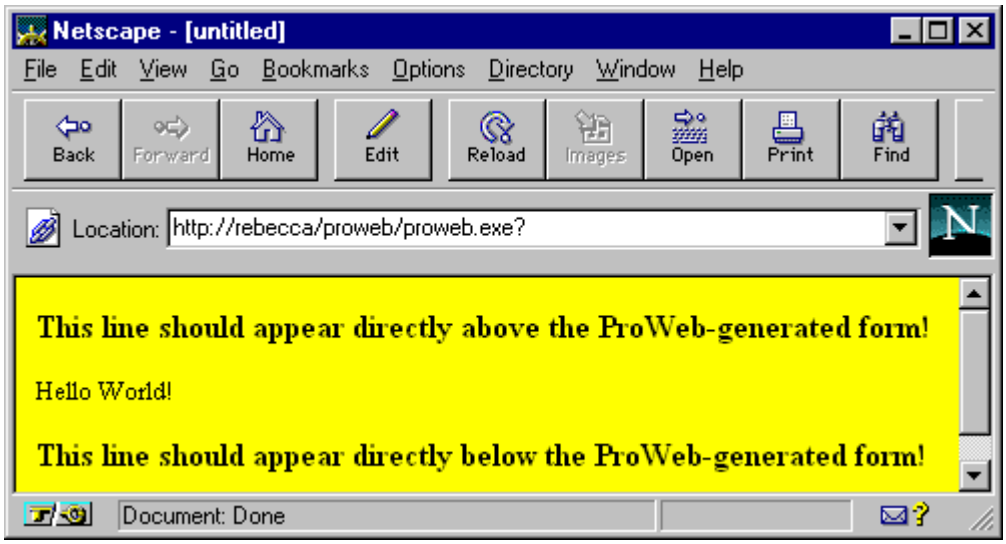
You might now be asking what happens if I have a more complex HTML page which I wish ProWeb to use to generate a page. Will I, for example, have control over where ProWeb embeds the form? In the modified 'Hello World!' example above, the line "Hello World!" was inserted after the contents of the <BODY>; however, everything within the <BODY> tags (i.e. the line "ProWeb Hello World! Example") became part of the form. Let's look at another example; the ProWeb code is pretty much the same as before, except that a different HTML template page is being used:

```
main_goal :-
 proweb_send_form( hello_form ).

proweb_page( hello_form, [include('html/hw2.htm')] ).

proweb_form( hello_form, 'Hello World!' ).
```

The HTML code of HW2.HTM is:

```
<HTML>
 <HEAD>
  <TITLE>
  </TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <H3>This line should appear directly above the ProWeb-generated form!</H3>
  <PROWEB FORM>
  <H3>This line should appear directly below the ProWeb-generated form!</H3>
 </BODY>
</HTML>
```

You will notice that we have placed within the above HTML code a tag specific to ProWeb, namely the <PROWEB FORM> tag. This tag will be ignored by your web browser as it has no meaning whatsoever in HTML. To ProWeb, however, it is very important; this tag is used as an indicator by ProWeb that any form it generates should be inserted at this point in the page:



The source code of the ProWeb-generated page is as follows:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <H3>This line should appear directly above the ProWeb-generated form!</H3>
```

```
    <PROWEB FORM>
    Hello World!
    <H3>This line should appear directly below the ProWeb-generated form!</H3>
   </FORM>
  </BODY>
</HTML>
```

You will notice that the entire contents of the body (everything between the <BODY>tags) still forms part of the HTML form.

In this third modification of the 'Hello World!' example, we are going to modify the HTML page on disk so that the ProWeb-generated form does not include everything within the <BODY> tags. Again, the ProWeb code is much the same as before, only using yet another HTML page on disk:

```
main_goal :-
 proweb_send_form( hello_form ).

proweb_page( hello_form, [include('html/hw3.htm')] ).

proweb_form( hello_form, 'Hello World!' ).
```

The HTML source code of HW3.HTM is as follows:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <H3>This line should appear directly above the ProWeb-generated form!</H3>
  <FORM>
   <PROWEB FORM>
  </FORM>
  <H3>This line should appear directly below the ProWeb-generated form!</H3>
 </BODY>
</HTML>
```

You will notice that we have added a pair of <FORM> tags; the opening <FORM> tag comes before the <PROWEB FORM> tag and </FORM> comes after.

When run, the HTML source code of the ProWeb-generated page is as follows:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY BGCOLOR="yellow">
  <H3>This line should appear directly above the ProWeb-generated form!</H3>
  <FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
   <PROWEB FORM>
   Hello World!
  </FORM>
```

```
  <H3>This line should appear directly below the ProWeb-generated form!</H3>
 </BODY>
</HTML>
```

The body of the HTML page is now correct with "<H3>This line should appear directly above the ProWeb-generated form!</H3>" before the form and "<H3>This line should appear directly below the ProWeb-generated form!</H3>" after the form. Notice also that ProWeb has expanded the opening <FORM> tag by adding the necessary attributes.

Deploying Multiple ProWeb-Based Applications

It is highly probable that on your web site you are going to want to run several ProWeb-based applications. By modifying the main_goal setting in PROWEB.INI, it is possible to run one of several ProWeb-based applications from the same ProWeb home directory.

For each ProWeb-based application the launch page is to launch, a different attribute/value pair needs to be sent to ProWeb; if we wanted to launch "Hello World!" and "Network Paths" from the same page, for example, we could have two different hyperlinks:

```
<A HREF="/proweb/proweb.exe?eg=hw">Run Hello World Example</A>
<A HREF="/proweb/proweb.exe?eg=np">Run Network Paths Example</A>
```

At the Prolog/ProWeb end, we need to ascertain the value of the *eg* attribute to determine which of the examples is to run:

```
overall_main_goal :-
 (  proweb_returned_input( eg, App )
 -> proweb_assert( eg(App) )
 ;  proweb_call( eg(App) )
 ),
 app( App, Goal ),
 call( Goal ).

app( hw, hello ).
app( np, netpaths ).
```

The *main_goal/0* 'entry point' into Hello World! and Network Paths would, of course, need to be renamed; above, we have chosen to rename these predicates *hello/0* and *netpaths/0* respectively.

Whenever your ProWeb 'launch page' is now submitted, *overall_main_goal/0* will be executed; the application chosen by the client will be 'picked-up' and the appropriate goal (e.g. *hello/0* or *netpaths/0*) executed.

When executing more than one ProWeb-based application on the same web site, the following issues should be considered:

- Will the source code of one application be abolished prior to loading the code of another?

- Are predicate name clashes likely to occur between the source code of two or more applications?

- Are you using an underscore as the first argument of, say, a proweb_page/2 clause which will be picked up by two different applications, one of them by accident?

- Is it possible to have the source code for two or more applications loaded at the same time?

Does the application alter its own program whilst running? If so, changes during one conversation will affect all others.

Loading Source Code 'On The Fly'

If you are likely to alter the source code to the various ProWeb-based applications on your web site on a regular basis, it is probably better to have such code loaded into *WIN-PROLOG* 'on the fly'. The following code may help:

```
overall_main_goal :-
  ensure_loaded( prolog( my_egs.pl ) ),
   (  proweb_returned_input( eg, Name )
   -> proweb_asserta( eg(Name) )
   ;  proweb_call( eg(Name) )
   ),
   eg( Name, MainGoal, _ ),
   call( MainGoal ).

overall_post_goal :-
   ensure_loaded( prolog( my_egs.pl ) ),
   (  proweb_returned_input( eg, Name )
   -> proweb_asserta( eg(Name) )
   ;  proweb_call( eg(Name) )
   ),
   eg( Name, _, PostGoal ),
   call( PostGoal ).
```

One directory that *WIN-PROLOG* can refer to directly is its EXAMPLES directory. For a web site where the source code to more than one application is loaded 'on the fly' it is recommended that you create such a directory within C:\INETPUB\PROWEB. Within this EXAMPLES directory itself, you would then create one directory for each application; such a directory would hold its source code, template HTML and graphics files.

You will also need a file in your C:\INETPUB\PROWEB directory to hold the *eg/3* clauses, one for each individual ProWeb-based application on your web site. The above code assumes this file is named MY_EGS.PL.

```
eg( np,
      ( ensure_loaded( examples('pw_ntpth.pl' ) ),
        netpaths
      ),
      true
    ).
```

Whilst viewing a page on your web site, a client submits a URL of *http://<server_name>/proweb/proweb.exe?eg=np*. With *overall_main_goal* set as your main goal, *overall_main_goal/0* is executed. The file MY_EGS.PL is loaded into Prolog. The value of 'eg' is then 'picked-up' from the URL line via the *proweb_returned_input/2* call. So as not to fail on the second and subsequent pages of a conversation, the value is asserted into the Prolog internal database. Once *eg* has a value, the relevant *eg/3* clause is found. PW_NTPTH.PL within C:\INETPUB\PROWEB\EXAMPLES is loaded and the Network Paths' 'main goal', *netpaths/0*, executed. If a post goal has been specified, rather than *true* as above, this will be executed as well following execution of the main goal.

Sending Unicode To ProWeb From A Launch Page

To send Unicode to ProWeb from an HTML launch page, you need to include the following <META> tag within the HTML file:

<META HTTP-EQUIV="content-type" CONTENT="text/html;charset=utf-8">

Embedding One Form Within Another Form

The '&' notation allow you to embed one form within another form. The following example embeds three forms (each defined via their own *proweb_form/2* clause) into a fourth form.

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.

main_goal :-
  proweb_send_form( high_level_form ).

proweb_page( _, [] ).

proweb_form( high_level_form,
          [ &low_level_form1,
            hr,
            &low_level_form2,
            hr,
            &low_level_form3
          ]
        ).
```

```
proweb_form( low_level_form1,
          [ `This is low level form 1`
          ]
        ).

proweb_form( low_level_form2,
          [ `This is low level form 2`
          ]
        ).

proweb_form( low_level_form3,
          [ `This is low level form 3`
          ]
        ).
```

Note that the generated HTML page only has one set of <FORM> tags.

The '&&' Notation And proweb_text/2 Predicate

The '&&' notation allows you to insert text, defined in a user-defined *proweb_text/2* fact, at a specific place within an HTML page.

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.

main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [] ).

proweb_form( my_form,
          [ &&(canned_text1),
            br,
            &&(canned_text2)
          ]
        ).

proweb_text( canned_text1, `This is some canned text` ).
proweb_text( canned_text2, `This is some more canned text` ).
```

Continuing With Different Parameters on the URL Line

ProWeb provides the ability to create a hyperlink that allows an existing ProWeb conversation to be continued but with different parameters passed back in on the URL line.

A line similar to the following:

```
<A HREF=_proweb DATA="var1=val1&var2=val2">ProWeb</A>
```

placed in an HTML template page or:

```
a(href='_proweb',data='var1=val1&var2=val2'), `ProWeb`, /a
```

placed in a *proweb_form/2* clause will be expanded by ProWeb to become a hyperlink in the generated page:

```
<A
HREF="http://localhost/proweb/proweb.EXE?proweb_dotexe_added_to=ancho
r&amp;proweb_data_uco=['00114798941357360']&amp;proweb_data_page=[1]
&amp;var1=val1&amp;var2=val2">ProWeb</A>
```

Such a hyperlink allows the same ProWeb conversation to be continued but with any data you care to pass back into it on the URL line.

Embedding Hidden Comments

You can embed hidden comments in your HTML pages using ProWeb's '!' notation which maps onto an HTML <!> tag. This example code:

```
proweb_form( my_form,
          [ !(EmbeddedComment)
          ]
        ) :-
 time( 1, DateTime ),
 DateTime = (Date,Time),
 time( Date, Year, Month, Day ),
 ( write(`Page creation date: `),
   write(Day),
   write(`-`),
   write(Month),
   write(`-`),
   write(Year)
 ) ~> EmbeddedComment.
```

would place the following comment tag in an HTML page:

```
<! Page creation date: 26-3-2003>
```

Filtering What Goes On The URL Line

Occasionally, you may wish to filter the data that gets passed back to ProWeb on the URL line. For example, your ProWeb-generated page might contain a number of checkboxes and you only want to know about the ones that were checked and to ignore all the unchecked ones. The following HTML code uses Javascript to filter out any unchecked checkboxes on the URL line when it submits the page back to ProWeb:

```
<HTML>
<SCRIPT LANGUAGE=JAVASCRIPT TYPE="TEXT/JAVASCRIPT">
function submit_via_javascript()
{ baseurl = document.URL.split("?")[0]
  uco = document.forms[0].proweb_data_uco.value
  page = document.forms[0].proweb_data_page.value
  firstfield = 3
  lastfield = document.forms[0].length – 2
  form_data = ''
  for(i=firstfield;i<=lastfield;i++)
  { fieldname = document.forms[0].elements[i].name
    switch(document.forms[0].elements[i].type)
    { case "checkbox":
      if (document.forms[0].elements[i].checked) {form_data = form_data + '&' +
fieldname + '=' + 'yes'};
      break;

      default:
      fieldvalue = document.forms[0].elements[i].value;
      form_data = form_data + '&' + fieldname + '=' + fieldvalue;
      break;
    }
  }
  extra_data  = '&var1=val1'
  url         = baseurl + '?proweb_dotexe_added_to=anchor&proweb_data_uco=' +
uco + '&proweb_data_page=' + page + form_data + extra_data
  window.location.href=url
}
</SCRIPT>
<BODY>
<FORM METHOD=POST ACTION="http://localhost/proweb/proweb.exe">
<INPUT TYPE=HIDDEN NAME="proweb_dotexe_added_to"
VALUE="form">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco"
VALUE="['123456789']">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<P><INPUT TYPE=CHECKBOX NAME="q0001" VALUE="no"></P>
<P><INPUT TYPE=CHECKBOX NAME="q0002" VALUE="yes"></P>
<P><INPUT TYPE=CHECKBOX NAME="q0003" VALUE="no"></P>
<P><INPUT TYPE=BUTTON VALUE="Submit via Javascript"
onClick="submit_via_javascript()"></P>
</FORM>
</BODY>
</HTML>
```

Human-Computer Interaction

During the development and testing of a ProWeb-based application, certain human-computer interaction techniques should be considered throughout:

- Standardise on your use of file extensions; don't use a mixture of .HTM and .HTML. It is also worth bearing in mind that when you come to archive your web site by compressing it into, say, a ZIP file you may find that a file's four character file extension causes its leaf name to be altered.

- Standardise on the case used within path names as some web servers (i.e. Netscape FastTrack Server) are case sensitive; don't refer to the same file sometimes in upper case and other times in lower case.

- Give each page a consistent 'look and feel' throughout.

- At the end of a 'conversation', provide a hyperlink back to your ProWeb 'launch page' and web site's home page.

- Ensure that each page's submit button appears in a consistent place. If it's a long page, consider having a submit button at the top and at the bottom.

- Only have a reset button on a page if it is really required.

- Don't make pages 'too wide'; some people are still using VGA screens.

- Load each page into as many web browsers as you can to make sure it displays as you intended.

Chapter 20 - Scripting Language Support

This chapter documents ProWeb's support for scripting languages.

Client-Side Validation

For simple client-side validation of user input, ProWeb is capable of generating the necessary Javascript code for you. The program below creates a form with a single field; this field must be given an integer value between 0 and 9 inclusive before the page can be submitted:

```
main_goal :-
  proweb_send_form( my_page ).

proweb_page( [my_page], [] ).

proweb_form( my_page,
        [       ?my_question,
                input(type = submit)
        ]).

proweb_question( my_question,
            [method = input,
             type = integer,
             prefill = 0,
             lwb  = 0,  /* lower bound */
             upb  = 9   /* upper bound */
            ]
           ).
```

The source code of the ProWeb-generated HTML page is as follows:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
</HEAD>
<BODY>
<SCRIPT LANGUAGE="JavaScript">
<!--
function validateInt(fld)
{ var strval = fld.value ; var fltval = parseFloat( strval ) ; var intval = parseInt(
strval ) ; if ( isNaN( fltval ) == true || intval == 0 && strval != "0" || intval !=
fltval ) { alert( "\nNot an integer!" ) ; fld.focus() ; return( false ) ; } else { return(
true ) ; } }
function validateLwbUpb(fld,lwb,upb)  { var strval = fld.value ; var fltval =
parseFloat( strval ) ; if ( fltval < lwb || fltval > upb ) { alert( "\nNot in range [" +
lwb + "," + upb + "]" ) ; fld.focus() ; return( false ) ; } else { return( true ) ; } }
function validateForm(frm)
{ return ( validateInt(frm.q0001) && validateLwbUpb(frm.q0001,0,9) ) ; }
<!-- End Script -->
</SCRIPT>
<FORM NAME="ProWebForm" onSubmit="return
validateForm(ProWebForm)">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[78]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<INPUT NAME="q0001" TYPE=TEXT SIZE="10,1"  VALUE="0">
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Attaching Help to a Question

To keep the form within an HTML page tidy, you probably would not want to clutter it up with lines of text related to how its questions should be completed. The best solution to this problem is to have a 'Help' button associated with each question which, when clicked upon, displays another window containing such helpful text; ProWeb, with the help of Javascript, can do exactly that.

For each question requiring Javascript help, you will need to define a *proweb_help/3* clause in your source code:

```
proweb_help( my_question,
             `Help`,
             `~M~JHere is the helpful text for my question~M~J`
           ).
```

where the first argument is the name of the question, the second is the text to be displayed within the button itself and the third is the text to be displayed in the Javascript window. The '~M' will be converted to '\n'; the '~J' will be ignored.

Within your template HTML file, you will need to place a <PROWEB HELP=...> element near to its associated <PROWEB QUESTION=...> element:

```
<PROWEB QUESTION="my_question"><PROWEB HELP="my_question">
```

Alternatively, it could be included as a Prolog term within your source code; for example:

```
proweb_form( my_form,
             [ ?my_question,
               ???my_question
             ]
           ).
```

When the above code is utilised by ProWeb to dynamically create the next page, the following HTML code will be present in the resulting page:

```
<INPUT TYPE=BUTTON VALUE="Help" onClick="return alert( "\n\nHere is the helpful
text for my question\n ")">
```

The last thing that needs to be done is to set the ProWeb setting, *help*, to *on*:

```
?- proweb_setting( help, on ). <enter>
```

Adding Your Own Javascript Code

A <SCRIPT> element can appear within either the HEAD or BODY section, although it normally resides in the former. The following ProWeb-based program embeds a user-defined Javascript program within the HEAD section:

```
:-    multifile( proweb_page      / 2 ).
:-    multifile( proweb_form      / 2 ).

main_goal :-
  proweb_send_form( script_form ).

proweb_page( script_form,
          [ head,
            unencoded @ `<SCRIPT TYPE="TEXT/JAVASCRIPT"
LANGUAGE=JAVASCRIPT>~M~J`,
            unencoded @ '<!-- ~M~J',
                    `function showField(myText) {~M~J`,
            unencoded @ `    alert("Text:"+myText);~M~J`,
                    `}~M~J`,
            unencoded @ '// -->',
            /script,
            /head
          ] ).
```

```
proweb_form( script_form,
        [ unencoded @ `<INPUT TYPE="text" NAME="myeditfield">`,
          unencoded @ `<INPUT TYPE=BUTTON VALUE="Click Me!"
onClick="showField(document.forms[0].myeditfield.value)">`
        ]
      ).
```

The unencoded @ OBJECT lines ensure that characters such as ", < and > are preserved as entered rather than being converted to *"*, *<* and *>* respectively. The source code for the HTML page produced by the above program is as shown below:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
<SCRIPT TYPE="TEXT/JAVASCRIPT" LANGUAGE=JAVASCRIPT>
<!-
function showField(myText) {
   alert("Text:"+myText);
}
// ->
</SCRIPT>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[400]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<INPUT TYPE="text" NAME="myeditfield"><INPUT TYPE=BUTTON VALUE="Click
Me!" onClick="showField(document.forms[0].myeditfield.value)">
</FORM>
</BODY>
</HTML>
```

Chapter 21 - HTML Frameset Support

This chapter descibes how to develop a frame-based ProWeb-based application.

Developing a Frame-Based ProWeb Application

A frameset consists of two or more HTML pages. A frameset basically splits the web browser window into several smaller windows. The following ProWeb-based program handles such a frameset.

The HTML frameset document, FRAMESET.HTM, states that it consists of three HTML pages. The first two pages come from physical HTML template files (i.e. INFO.HTM and NAVIGATE.HTM) stored in the same directory as FRAMESET.HTM itself. The third page is going to be created by ProWeb 'on the fly'; this will involve embedding one of two forms, *question_form* or *answer_form*, within another template HTML file, PAGE.HTM. The Prolog source code for the program is:

```
:- multifile( proweb_page     / 2 ).
:- multifile( proweb_form     / 2 ).
:- multifile( proweb_question / 2 ).
:- multifile( proweb_frameset / 2 ).

main_goal :-
  proweb_post_reply( some_more_text, `You can have many frames defined this
way but they will all appear identically!` ),
  proweb_send_form( question_form ),
  proweb_send_form( answer_form ).

proweb_frameset( include('frames/frameset.htm') ).

proweb_page( question_form, include('frames/page.htm'    ) ).

proweb_page( answer_form,   include('frames/page.htm'    ) ).

proweb_page( navigate,      include('frames/navigate.htm') ).

proweb_page( info,          include('frames/info.htm'    ) ).

proweb_form( question_form,
           [ `This is the question: `,
             ?my_question,
             p @ input(type=submit)
           ]
         ).
```

```
proweb_form( answer_form,
          [ `This is your answer: `,
            MyAnswer
          ]
        ) :-
  proweb_returned_answer( my_question, MyAnswer ).

proweb_question( my_question,
             [ method =  radio,
               select =  [yes,no],
               prefill = yes
             ]
           ).
```

A user defined *proweb_frameset/2* clause specifies the filename of an HTML frameset document and the ProWeb forms to which it applies. The first argument can be either the name of a single ProWeb form within the frameset or a list of ProWeb forms constituting a part of the frameset. The second argument is a term which specifies the HTML file hosting the form(s). *proweb_frameset/2* is called during the expansion stage (just like *proweb_page/2* and *proweb_form/2*) prior to the page being sent. You now need a *proweb_page/2* clause for each frame in the frameset. In the course of assembling the frameset, ProWeb will save a number of HTML files to the conversation folder (i.e. C:\INETPUB\PROWEB\TEMP); collectively, these files will constitute all the HTML documents making up the frameset.

The ProWeb setting, *max_forms_per_page*, refers to the number of forms embedded within the "active" frame; the frame generated by ProWeb 'on the fly'.

The source code of C:\INETPUB\PROWEB\HTML\FRAMES\FRAMESET.HTM is:

```
<HTML>
 <HEAD>
  <TITLE>
  FrameSet Example - Page <PROWEB PAGENUMBER>
  </TITLE>
 </HEAD>
 <FRAMESET COLS="240,*" FRAMEBORDER="yes" BORDER=1
BORDERCOLOR="#888888">
  <FRAMESET ROWS="*,80" FRAMEBORDER="yes" BORDER=1
BORDERCOLOR="#888888">
   <FRAME NAME="info"    SRC="_PROWEB_STATIC_PAGE_info"   >
   <FRAME NAME="navigate" SRC="_PROWEB_STATIC_PAGE_navigate">
  </FRAMESET>
   <FRAME NAME="active"  SRC="_PROWEB_ACTIVE_PAGE"      >
 </FRAMESET>
 <NOFRAMES>
  <BODY>
   <BLOCKQUOTE>
    <P>This page can only be viewed within a browser that supports frames.</P>
   </BLOCKQUOTE>
  </BODY>
```

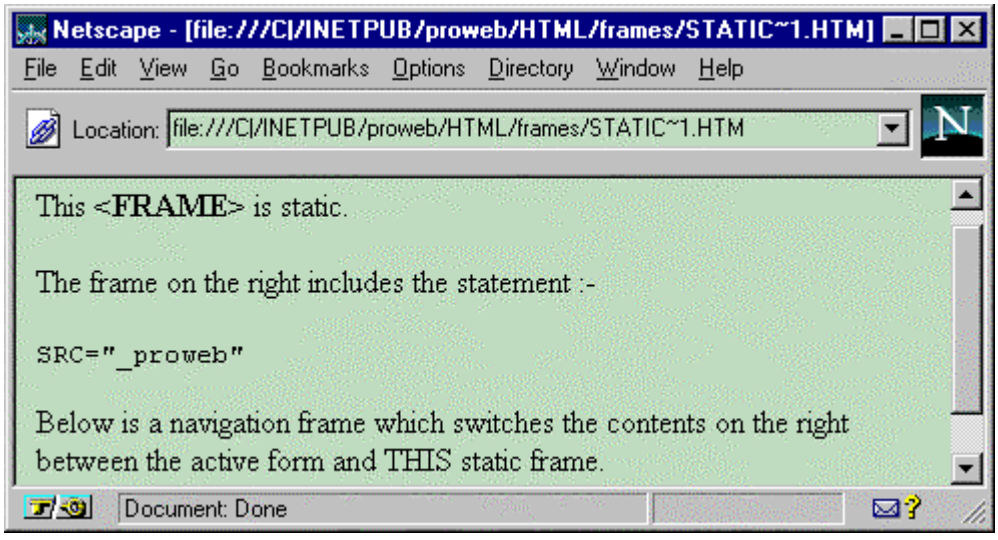
```
 </NOFRAMES>
</HTML>
```

The static frames of a frameset need to have '_PROWEB_STATIC_PAGE_' added to their <FRAME SRC=...> tags.

The source code of C:\INETPUB\PROWEB\HTML\FRAMES\INFO.HTM is:

```
<HTML>
   <HEAD><TITLE></TITLE></HEAD>
   <BODY>
      <P>This <B>&lt; FRAME&gt;</B> is static.</P>
      <P>The frame on the right includes the statement :-</P>
      <P><PRE>SRC="_PROWEB_ACTIVE_PAGE"</PRE></P>
      <P><PROWEB REPLY="some_more_text"></P>
      <P>Below is a navigation frame which switches the contents on the right
between the active form and THIS static frame.</P>
   </BODY>
</HTML>
```

INFO.HTM is a normal HTML file within a frameset; note, however, that it has no <FORM> element. It is 'read in' by ProWeb and the <PROWEB REPLY="some_more_text"> element is replaced by the value of *some_more_text*; the page is then saved in the conversation folder (i.e. C:\INETPUB\PROWEB\TEMP) as a ProWeb-named HTML file (for this example, 00000011.HTM). Displayed on its own within a web browser, INFO.HTM will look like:



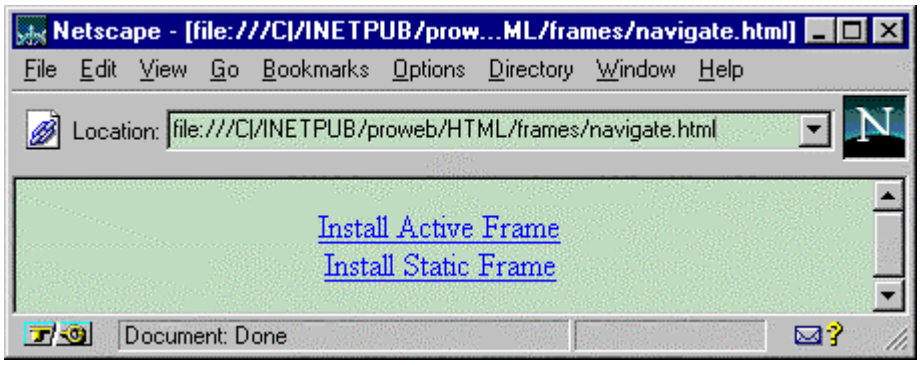
The source code of C:\INETPUB\PROWEB\HTML\FRAMES\NAVIGATE.HTM is:

```
<HTML>
 <HEAD>
  <TITLE></TITLE>
 </HEAD>
 <BODY>
  <CENTER>
   <A HREF="_PROWEB_ACTIVE_PAGE"    TARGET="active">Install Active
Frame</A>
```

```
    <BR>
    <A HREF="_PROWEB_STATIC_PAGE_info" TARGET="active">Install Static
Frame</A>
  </CENTER>
 </BODY>
</HTML>
```

The static frames of a frameset need to have '_PROWEB_STATIC_PAGE_' added to their <A HREF=...> HTML tags.

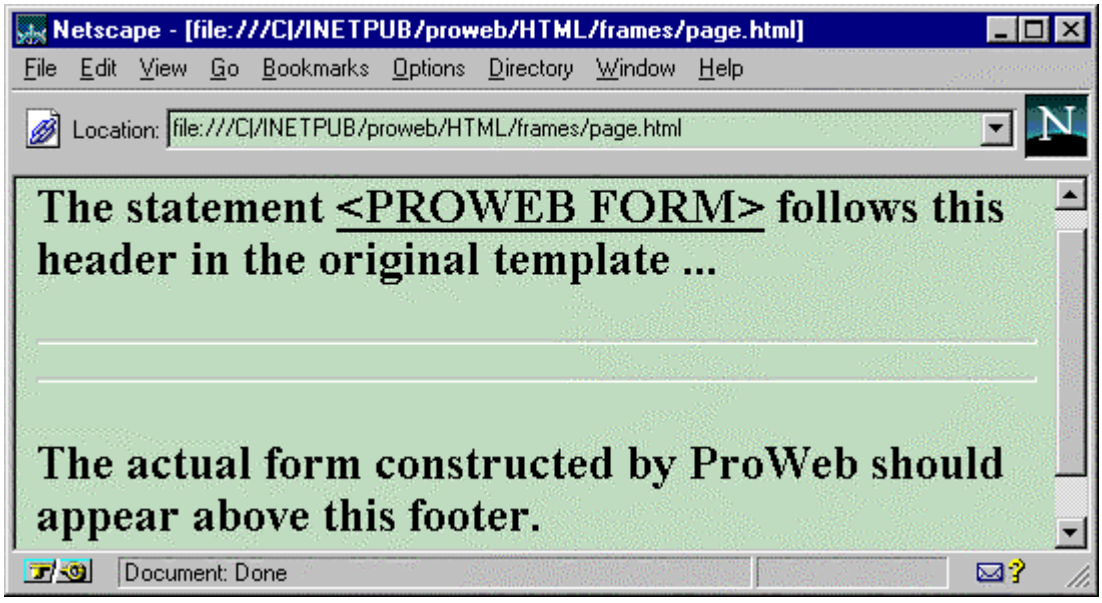
In a normal frameset, NAVIGATE.HTM would appear within the frameset as is; when included within a frameset handled by ProWeb, the two HREF values are altered to point to the HTML files created and saved in the conversation folder (i.e. C:\INETPUB\PROWEB\TEMP) by ProWeb; in fact, NAVIGATE.HTM, having been 'read in' and expanded by ProWeb, becomes a file in the conversation folder itself (for this example, 00000012.HTM). Displayed on its own within a web browser, NAVIGATE.HTM will look like:



The source code of C:\INETPUB\PROWEB\HTML\FRAMES\PAGE.HTM is:

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<FORM TARGET="_parent">
<H2>The statement <U>&lt; PROWEB FORM&gt;</U> follows this header in the
original template ...</H2>
<HR>
<PROWEB FORM>
<HR>
<H2>The actual form constructed by ProWeb should appear above this footer.</H2>
</FORM>
</BODY>
</HTML>
```

The <PROWEB FORM> element in the above code is an indication to ProWeb where the ProWeb form (either *question_form* or *answer_form*) is to be inserted when this template HTML file is expanded by ProWeb. Displaying PAGE.HTM on its own, without ProWeb's help, within a web browser, the screen will look like:



When the program is run, launched from a suitable ProWeb launch page, the following steps will be performed:

main_goal/0 is called.

ProWeb begins to execute the first line, *proweb_send_form(question_form)*.

Finding a generic user-defined *proweb_page/2* clause, *proweb_page(_,include('html/frames/page.htm'))*, ProWeb reads in PAGE.HTM.

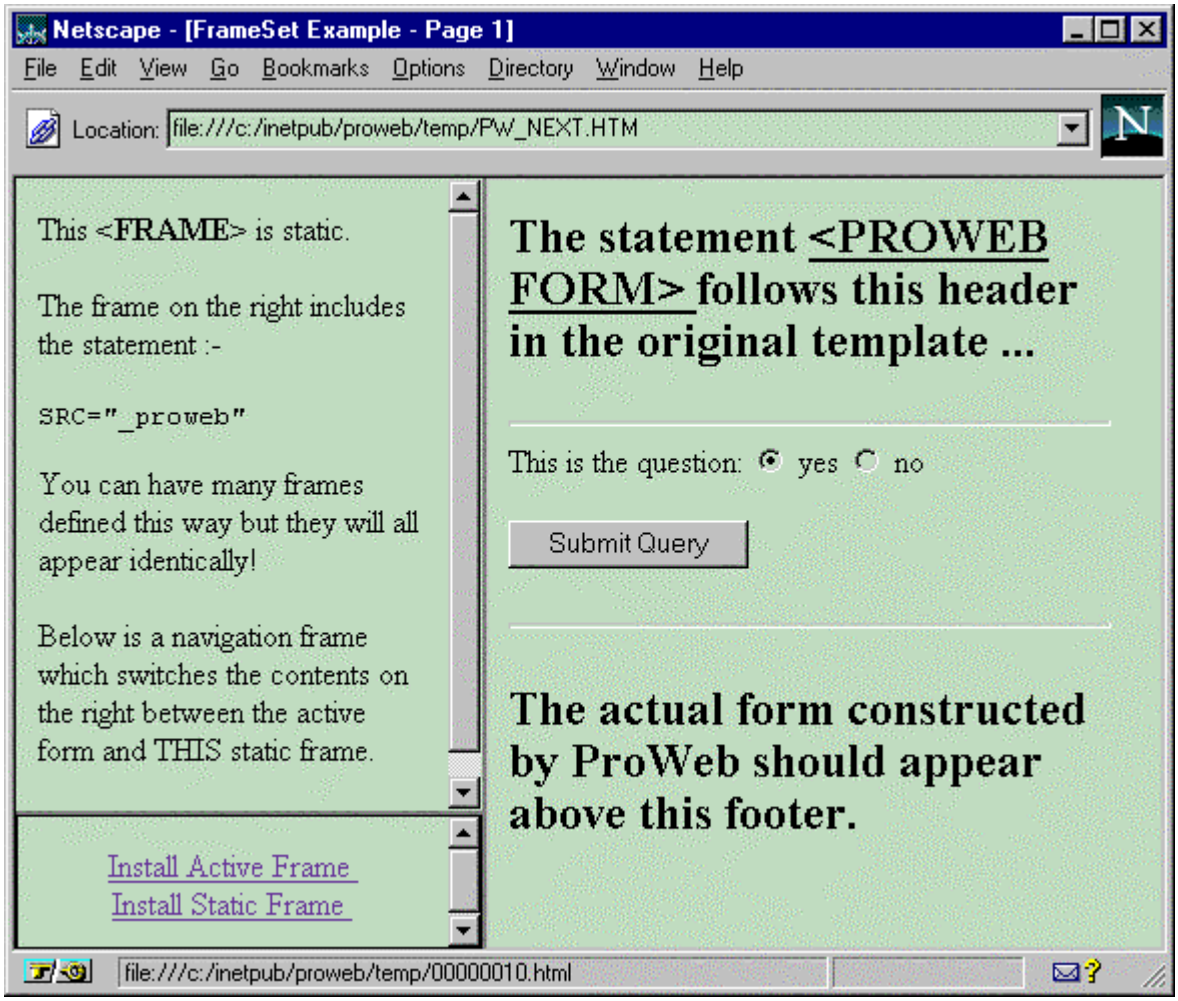
Finding the required *proweb_form/2* and *proweb_question/2* clauses for question_form, ProWeb embeds question_form within the HTML code obtained from PAGE.HTM after the <PROWEB FORM> element.

ProWeb, upon finding a user-defined *proweb_frameset/2* clause to which all forms within the program apply, does not send question_form as it is, but saves it to the conversation folder, to be referenced later.

FRAMESET.HTM and the HTML documents to which it refers are 'read in', amended as required (i.e. original filenames become ProWeb-generated filenames) and saved to disk in the conversation folder. Where the SRC of an HTML document is "_PROWEB_ACTIVE_PAGE", ProWeb replaces it with its filename for the question_form.

ProWeb sends a revised FRAMESET.HTM to the client which then 'picks-up' the three ProWeb-created HTML documents (i.e. 00000010.HTM, 00000011.HTM and 00000012.HTM) saved in ProWeb's conversation folder.

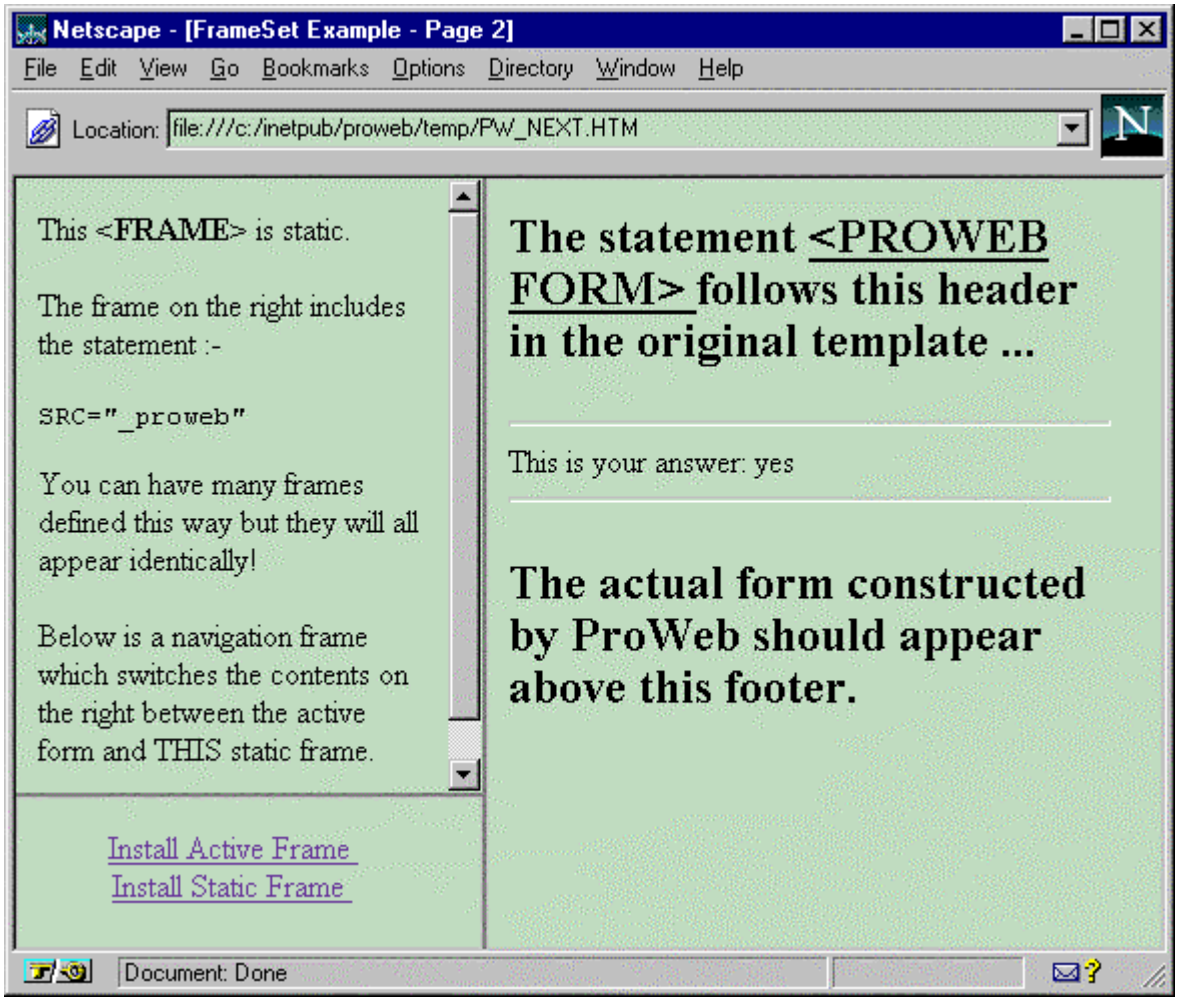
This first instance of the frameset is now displayed in the client's web browser:



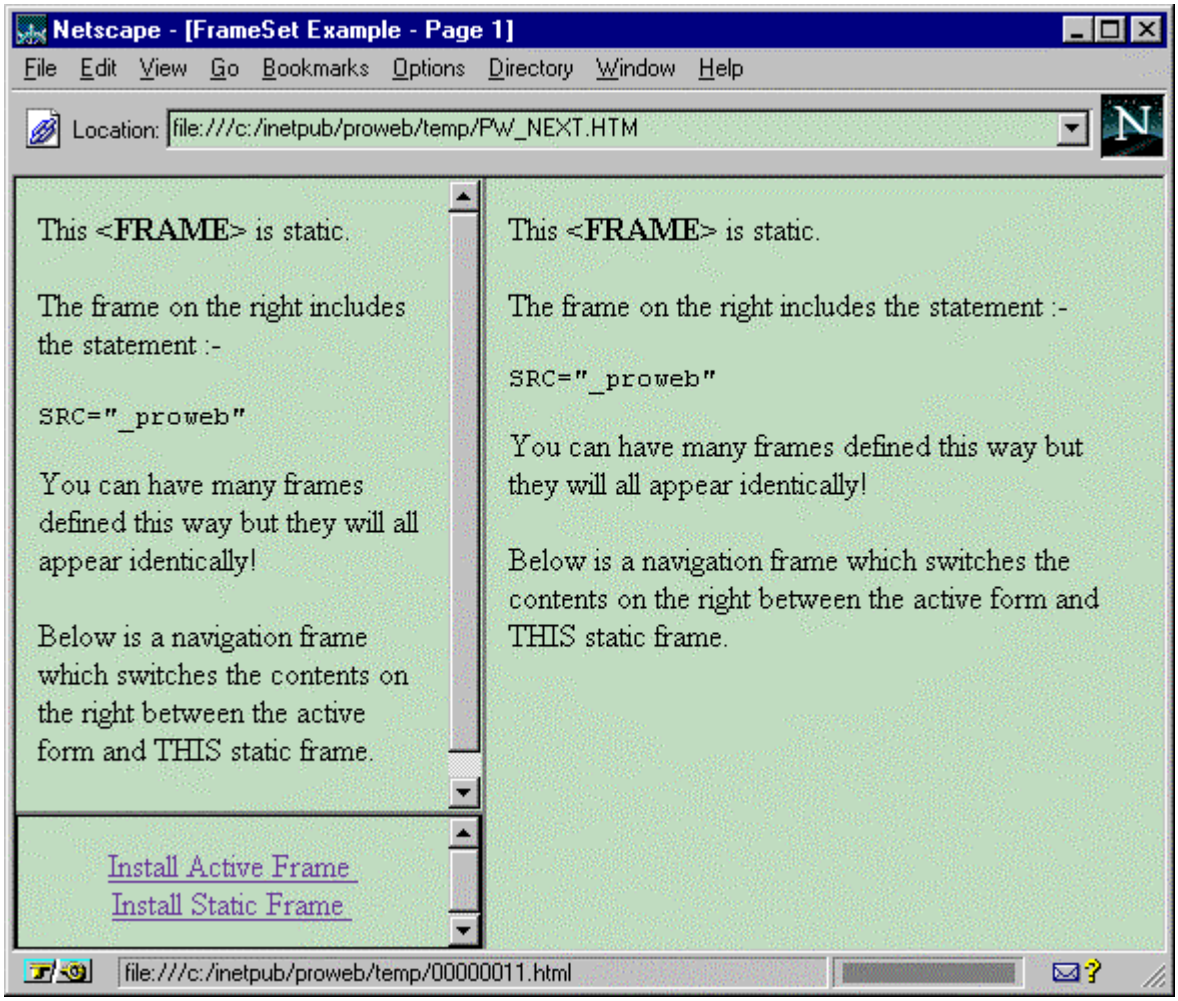
Upon answering the question, the frameset is submitted.

The program is run from the beginning again with *main_goal/0* called as before. The line, proweb_send_form(question_form), succeeds but is not acted upon as the form has already been sent.

ProWeb begins to execute the next line, *proweb_send_form(answer_form)*. Steps 3 to 7 above are executed again, but this time, with answer_form instead of question_form. When displayed for a second time, the frameset's three ProWeb-created HTML documents will be 00000013.HTM, 00000014.HTM and 00000015.HTM:



Regardless of whether it is the first or second instance of the frameset displayed, the "Install Static Frame" hyperlink can be clicked upon; this will cause the "active" frame to display the "info" frame as well:



For the first instance of the frameset, the source code of the ProWeb-generated page (the expanded FRAMESET.HTM) is:

```
<HTML>
<HEAD>
<TITLE>FrameSet Example - Page 1</TITLE>
</HEAD>
<FRAMESET COLS="240,*" FRAMEBORDER=YES BORDER=1
BORDERCOLOR="#888888">
<FRAMESET ROWS="*,80" FRAMEBORDER=YES BORDER=1
BORDERCOLOR="#888888">
<FRAME NAME="info" SRC="00000011.htm">
<FRAME NAME="navigate" SRC="00000012.htm">
</FRAMESET>
<FRAME NAME="active" SRC="00000010.htm">
</FRAMESET>
<NOFRAMES>
<BODY>
<FORM>
```

```
<BLOCKQUOTE>
<P>This page can only be viewed within a browser that supports frames.</P>
</BLOCKQUOTE>
</FORM>
</BODY>
</NOFRAMES>
</HTML>
```

For the first instance of the frameset, the source code of C:\INETPUB\PROWEB\TEMP\00000012.HTM (NAVIGATE.HTM as expanded by ProWeb) is:

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<CENTER>
<A HREF="00000010.htm" TARGET="active">Install Active Frame</A>
<BR>
<A HREF="00000011.htm" TARGET="active">Install Static Frame</A>
</CENTER>
</BODY>
</HTML>
```

If hyperlinks such as these are not converted to ProWeb-generated filenames in your own programs then there is something wrong with either its pathname, leafname or extension.

For the first instance of the frameset, the source code for the "active" frame (i.e. 00000010.HTM) is:

```
<HTML>
<HEAD><TITLE></TITLE></HEAD>
<BODY>
<FORM TARGET=_parent>
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[234]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<H2>The statement <U>&lt; PROWEB FORM&gt;</U> follows this header in the
original template ...</H2>
<HR>
<PROWEB FORM>
This is the question:
<INPUT TYPE=RADIO NAME="q0001" VALUE="yes" CHECKED>yes
<INPUT TYPE=RADIO NAME="q0001" VALUE="no">no
<P><INPUT TYPE=SUBMIT></P>
<HR>
<H2>The actual form constructed by ProWeb should appear above this footer.</H2>
</FORM>
</BODY>
</HTML>
```

For the second instance of the frameset, the source code of C:\INETPUB\PROWEB\TEMP\00000015.HTM (NAVIGATE.HTM as expanded by ProWeb) is:

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<CENTER>
<A HREF="00000013.htm" TARGET="active">Install Active Frame</A>
<BR>
<A HREF="00000014.htm" TARGET="active">Install Static Frame</A>
</CENTER>
</BODY>
</HTML>
```

For the second instance of the frameset, the source code for the "active" frame (i.e. 00000013.HTM) is:

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<FORM TARGET=_parent>
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[239]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[2]">
<H2>The statement <U>&lt; PROWEB FORM&gt;</U> follows this header in the
original template ...</H2>
<HR>
<PROWEB FORM>
This is your answer: yes
<HR>
<H2>The actual form constructed by ProWeb should appear above this footer.</H2>
</FORM>
</BODY>
</HTML>
```

Specifying The Path Correctly Within The <FRAME> Tags

Suppose you have an HTML top-level frameset page called FRAMESET.HTM which refers to an HTML file named NAVIGATE.HTM. If ProWeb were not being used and both files resided in the same directory, no path need be specified in the <FRAME> tag for NAVIGATE.HTM. With ProWeb, the <FRAME> tag for NAVIGATE.HTM must be specified in relation to the value of ProWeb's html_path setting. The hyperlinks within NAVIGATE.HTM itself must also be specified in relation to the value of ProWeb's html_path setting. When the frameset is displayed in the web browser, all the hyperlinks from NAVIGATE.HTM (and indeed the hyperlink to NAVIGATE.HTM itself) will have been redirected to new HTML files in ProWeb's temporary directory. If a hyperlink within the displayed frameset points to its original HTML file, it indicates that ProWeb could not locate the file in the path specified.

Specifying The Path Correctly For Local Hyperlinks

If you place a local hyperlink within a template HTML file which is then used as part of a frameset within ProWeb, the local hyperlink is modified by having the value of html_path prefixed to it. If the template HTML file contains:

```
...
<A HREF="#section_one">Section One</A>
...
<A NAME="section_one"><H3>Section One</H3></A>
...
```

the resulting page, after conversion by ProWeb and saving in the temporary directory under a new name, will contain something like:

```
...
<A HREF="/ProWeb/HTML/#section_one">Section One</A>
...
```

which gives an "HTTP/1.0 403 Access Forbidden" error when clicked upon.

Chapter 22 - Cascading Style Sheet Support

This chapter documents how to incorporate a cascading style sheet into your ProWeb application.

An Example

The following program will generate an HTML page containing its own built-in cascading style sheet:

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [     html,
                      head,
                      style(type='text/css'),
                      StyleSheet,
                      /style,
                      /head,
                      body(bgcolor=white),
                      form,
                      proweb(form),
                      /form,
                      /body,
                      /html
               ]
             ) :-
  ( write(`H1 {color:red}`),
    nl,
    write(`H2 {font-style:italic}`),
    nl,
    write(`H3 {font-family:serif}`),
    nl,
    write(`P {font-family:sans-serif}`)
  ) ~> StyleSheet.

proweb_form( my_form,     [         h1, 'This is red text', /h1,
                                    h2, 'This is italic text', /h2,
                                    h3, 'This is serif text', /h3,
                                    p, 'This is sans-serif text', /p
                                ]
                    ).
```

The source code of the generated page is as follows:

```
<HTML>
<HEAD>
<TITLE>
</TITLE>
<STYLE TYPE="text/css">
H1 {color:red}
H2 {font-style:italic}
H3 {font-family:serif}
P {font-family:sans-serif}
</STYLE>
</HEAD>
<BODY BGCOLOR="white">
<FORM METHOD=POST ACTION="/ProWeb/proweb.exe">
<INPUT TYPE=HIDDEN NAME="proweb_data_uco" VALUE="[25]">
<INPUT TYPE=HIDDEN NAME="proweb_data_page" VALUE="[1]">
<PROWEB FORM>
<H1>This is red text</H1>
<H2>This is italic text</H2>
<H3>This is serif text</H3>
<P>This is sans-serif text</P>
</FORM>
</BODY>
</HTML>
```

Template Page Example

You can also place a line such as the following:

```
<LINK REL="stylesheet" TYPE="text/css" HREF="mystyle.css">
```

into the <HEAD> section of a ProWeb template HTML page.

Chapter 23 - Java Support

This chapter documents how to combine ProWeb and Java.

Developing a Java Front End for your ProWeb-Based Application

A Java applet is an application which runs within an HTML page within a client's web browser. Should you wish to embed a Java applet within a ProWeb-generated HTML page with each application passing a message to the other, the following "Hello" example program will prove of assistance.

The ProWeb-based program's main goal is *hello/0*; this predicate has two clause definitions. On running the program for the first time, the line *proweb_returned_input(text,Hello)* in the first clause fails and so the second clause, which creates the HTML page containing the Java applet, is executed:

```
hello :-
  proweb_returned_input( text, Hello ),
  !,
  proweb_send_form( time_form ).

hello :-
  create_applet_element ~> SelectString,
  proweb_post_reply( my_java_applet, unencoded @ SelectString ),
  proweb_send_form( hello_form ).
```

The program has two forms, *hello_form* and *time_form*; their user-defined *proweb_page/2* and *proweb_form/2* clauses are as for a normal ProWeb application:

```
proweb_page( [ Form ],
          [ include('hello/head.html'),
            Title,
            include('hello/body.html'),
            proweb(Form),
            include('hello/foot.html')
          ]
        ) :- Form = hello_form,
             Title = `Hello`.

proweb_form( hello_form, [??my_java_applet] ).
```

For a more complex example, an alternative to the above *proweb_form/2* clause is to use a template HTML file and place a <PROWEB REPLY="my_java_applet"> element at the location in the page you wish the applet to appear.

```
proweb_page( [time_form], [] ).
```

```
proweb_form( time_form, [TimeString] ) :-
  time(Hrs,Mins,Secs,_),
  (  write( 'time=' ),
     write( Hrs ),
     write( ':'),
     write( Mins ),
     write( ':'),
     write( Secs ),
     nl
  ) ~> TimeString.
```

The <APPLET>element to be embedded within the HTML page is defined within the user-defined *create_applet_element/0* clause. The URL for the Java applet itself is http://<server_name>/proweb/examples/hello/hworld.class; in order for the applet to 'talk' to the ProWeb program, the URL required to launch the ProWeb program must be passed to the applet in the form of one or more <PARAM> elements:

```
create_applet_element :-
   proweb_returned_input( eg, EG ),
   java_HTTP( EG, HTTP ),
   proweb_setting( base_url, Home ),
   ( write(Home), write('/'), write('proweb.exe' ) ) ~> URL,
   html_print_nl( applet(code='hworld.class',
                         codebase='examples/hello/',
                         width=100,
                         height=100)
                ),
   html_print_nl( param(name=eg,value=HTTP) ),
   html_print_nl( param(name=proweb_path,value=URL) ),
   html_print_nl( param(name=name,value='Hello') ),
   html_print_nl( /applet ).
```

Note: the <APPLET> element has been deprecated in HTML 4.0 and may eventually be replaced by the <OBJECT> element.

The following user-defined clauses translates a string into HTTP format:

```
java_HTTP( EG, HTTP ) :-
   write( EG ) ~> EGstr,
   strchr( EGstr, Bytes ),
   java_to_HTTP( Bytes, NewBytes ),
   strchr( HTTP, NewBytes ).

java_to_HTTP( [], [] ).

% Control and other characters are presented as
% #NN where NN is the hex representation of the character.
```

```
java_to_HTTP( [Char|Rest], [0'#,C1,C2|Rest2] ) :-
  ( Char < 32
  ; member( Char, "<>!\/[]}{#~+()*&^%$£""" )
  ),
  !,
  java_to_hex( Char, C1, C2 ),

java_to_HTTP( Rest, Rest2 ).

% Space characters are replaced by the 0'+ character.

java_to_HTTP( [32|Rest], [0'+|Rest2] ) :-
  !,
  java_to_HTTP( Rest, Rest2 ).

% Other characters are returned as is.

java_to_HTTP( [C|Rest], [C|Rest2] ) :-
  java_to_HTTP( Rest, Rest2 ).

% Translate a character into two hex characters.

java_to_hex( C, C1, C2 ) :-
  A1 is C // 16,
  A2 is C mod 16,
  java_hex_char( A1, C1 ),
  java_hex_char( A2, C2 ).

java_hex_char( A, C ) :-
  (  A > 9
  -> C is 0'A + (A mod 10)
  ;  C is 0'0 + A
  ).
```

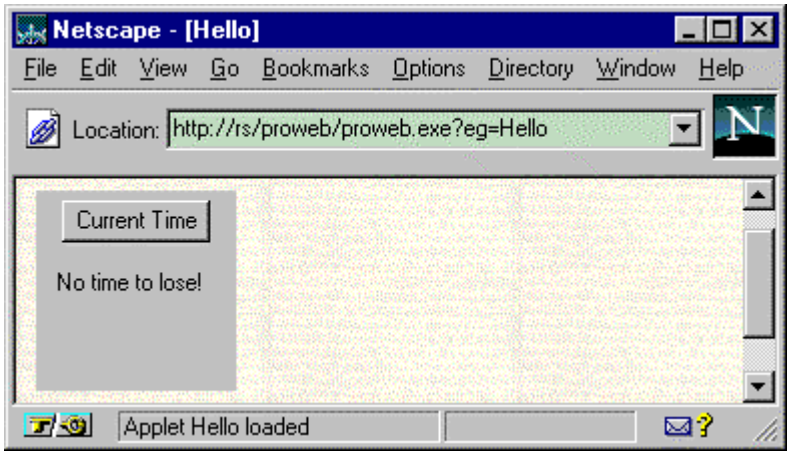
The source code of the Java applet is as follows:

```
import java.io.*;
import java.util.*;
import java.awt.*;
import java.net.*;
import java.applet.*;
```

```
public class hworld extends Applet
{  String pathname;
   String timeText = "No time to lose!";
   String examplename;
   Button submit;

   // Initialization code
   public void init()
   { examplename = getParameter( "name" );
     pathname = getParameter( "proweb_path" );
     submit = new Button( "Current Time" );
     add(submit);
   }

   // Respond to button actions
   public boolean action( Event evt, Object obj )
   { URL urlJava;
     Object target = evt.target;
     if( target.equals( submit ) )
     { String Stri = "http://"
                  + this.getCodeBase().getHost()
                  + pathname
                  + "?"
                  + "eg="
                  + examplename
                  + "&text=%5BHello%5D";
       repaint();

       // Try to fetch data from the URL connection
       try
       { URLConnection conJava;
         InputStream rawInput;
         urlJava = new URL( Stri );
         conJava  = urlJava.openConnection();
         rawInput = conJava.getInputStream();
         DataInputStream DataIn = new DataInputStream(rawInput);
```

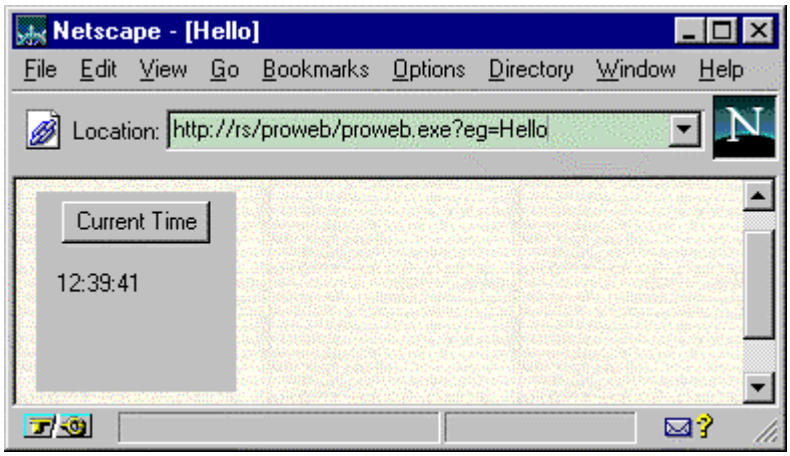
```
            // Find the text from the received input
            for( int i = 1; i < 130 ;  i++ )
            {
              String Value = DataIn.readLine();

              // Create a new "edges" collection from the received data
              if( Value.startsWith( "time=" ) )
              { timeText = Value.substring(5);
                repaint();
                break;
              }
            }
          }
          catch( IOException e )
          { System.out.println("an IO error has been raised. Error: " + e );
          }
        }
        else
          return super.action(evt,obj);
        return( true );
      }

      // Repaint the graphics
      public void paint( Graphics g )
      { g.drawString( timeText, 10, 50 );
      }
}
```

Upon running the ProWeb-based program, the first HTML page, consisting of *hello_form* and the Java applet, is displayed:



Clicking on the applet's Current Time button will cause the applet to 'talk' to the ProWeb-based program by constructing and executing the URL necessary (i.e. http://<server_name>/proweb/proweb.exe?eg=Hello&text=Hello) to get the ProWeb-based program's second form. The additional 'eg=Hello' attribute/value pair in the URL line simply indicates that this example was launched from a ProWeb 'launch page' that is capable of launching one of many ProWeb-based applications and the ProWeb

overlay file possesses the necessary code to handle this. Upon receiving the HTML page from the ProWeb program, the applet will search for 'text=' and extract its value, whereupon it will repaint itself; the text 'No time to lose!' will be changed to the current time:



Chapter 24 – Session Management

Getting The Unique Conversation Ordinal (UCO)

The Unique Conversation Ordinal (UCO) can be picked up by your own application code as ProWeb asserts a *proweb_data_uco/1* fact into Prolog's memory; the following ProWeb program demonstrates this:

```
main_goal :-
  proweb_send_form( first_form  ),
  proweb_send_form( second_form ).

proweb_page( first_form,  [] ).

proweb_page( second_form, [] ).

proweb_form( first_form,
           [ p,
             `The Unique Conversation Ordinal is `,
             UCO,
             /p,
             p @ input(type=submit)
           ]
         ) :-
  proweb_data_uco( UCO ).

proweb_form( second_form,
           [ `The Unique Conversation Ordinal is `,
             UCO
           ]
         ) :-
  proweb_data_uco( UCO ).
```

Chapter 25 - Security

This chapter discusses a number of security issues that you need to be aware of.

Introduction

This user guide documents the process of placing all ProWeb files into a single C:\INETPUB\PROWEB directory accessible via a single /PROWEB alias with both READ and EXECUTE access rights. This causes a security breach as certain executable-only files can be downloaded and effectively "stolen".

Downloading Files They Shouldn't

If you were to execute the following URL:

http://<server_name>/proweb/proweb.sys

where <server_name> is the name given to your server, your web broswer would throw up a save as dialogue box, whereupon you could click on the 'Save' button and download PROWEB.SYS from the server. If you can do it, so can everyone else!

Splitting The ProWeb Files Over Two Aliases

If security is an issue, a better approach would be to have two aliases, say /PWS_EXEC and /PWS_DATA; /PWS_EXEC would have EXECUTE only access and be for executable file only, whilst /PWS_DATA would have READ-ONLY access and be for read-only files only.

The directory pointed to by the /PWS_EXEC alias would contain PROWEB.EXE, PROWEB.SYS, PROWEB.OVL and PROWEB.INI. If your ProWeb application makes reference to any of Prolog's EXAMPLES, LIBRARY OR SYSTEM sub-directories, then these sub-directories must be placed in this directory as well.

Your PROWEB.INI file would contain settings such as:

```
[PROWEB]
BASE_URL=http://localhost/pws_exec/
TEMP_URL=http://localhost/pws_data/temp/
TEMP_PATH=c:\inetpub\pws_data\temp\
HTML_PATH=c:\inetpub\pws_exec\html\
```

The directory pointed to by the /PWS_DATA alias would contain just the read-only files such as graphics (i.e. GIF, JPEG etc) files. Each IMG tag would need to point directly to this alias.

ProWeb's temporary directory would need to be placed under the alias with read access (i.e. /PWS_DATA) as ProWeb may generate hyperlinks to temporary files stored there as in a frameset-based ProWeb application.

The Apache Server

The Apache Server forces the separation of the 'read-only' and 'execute' aliases anyway, so you could not, for example, refer a URL to an HTML page in the ProWeb executable directory.

1) download latest stable apache MSI windows installer : 2.2.15 as of 6 April 2010

Don't need SSL so get the httpd-2.2.15-win32-x86-no_ssl.msi

2) Run the installer (after checking correct SHA1, use prolog to check..) .

3) on server information use "only for the Current User on Port 8080, when started Manually" enter other on that page as you see fit.

Select typical for the setup type.

CHANGE the destination install directory to say c:\apache this will save you bother with permissions later.

4) Run the "Start apache in console" from the "Control Apache" menu program item.

5) Run a browser, enter the address : http://localhost:8080/ hit return, you should see a web page saying "It Works!" .

6) If you want to quit apache just close the console.

You now have a web server for the local machine.

Setting up PROWEB/WEBFLEX

1) Configuration

from a dos box cd to c:\apache (or wherever you installed it).

The directory htdocs is the equivalent of IIS wwwroot.

The directory conf contains the configuration files, cd into that and edit the file httpd.conf .

Go to the section <IfModule alias_module>

comment out the line at the bottom of that section : ScriptAlias /cgi-bin/ ...

by putting a # at the start of the line (like prolog %)

and add the two lines :

ScriptAlias /pws_exec "C:/apache/pws/exec/"

Alias /pws_data "C:/apache/pws/data/"

Comment out the section

<Directory "C:/apache/cgi-bin">

…

</Directory>

Add the following important use '/' not '\' :

<Directory "C:/apache/pws/exec" >

AllowOverride None

Options None

Order allow,deny

Allow from all

</Directory>

<Directory "C:/apache/pws/data" >

AllowOverride None

Options None

Order allow,deny

Allow from all

</Directory>

Save the file .

2) Set up proweb/webflex

Inside the C:\apache directory create a pws directory then install proweb\webflex as you

would in IIS setting paths appropriately in the .ini files.

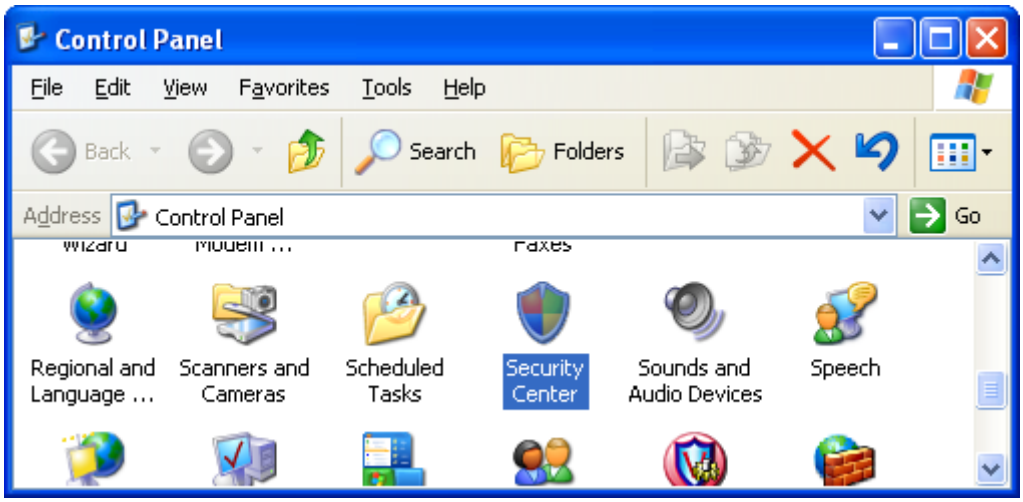
Add kick-off pages into the htdocs folder (replace the default) .

3) Start apache in console, then from the browser navigate to the appropriate page http://localhost:8080/etc…

Should work now!

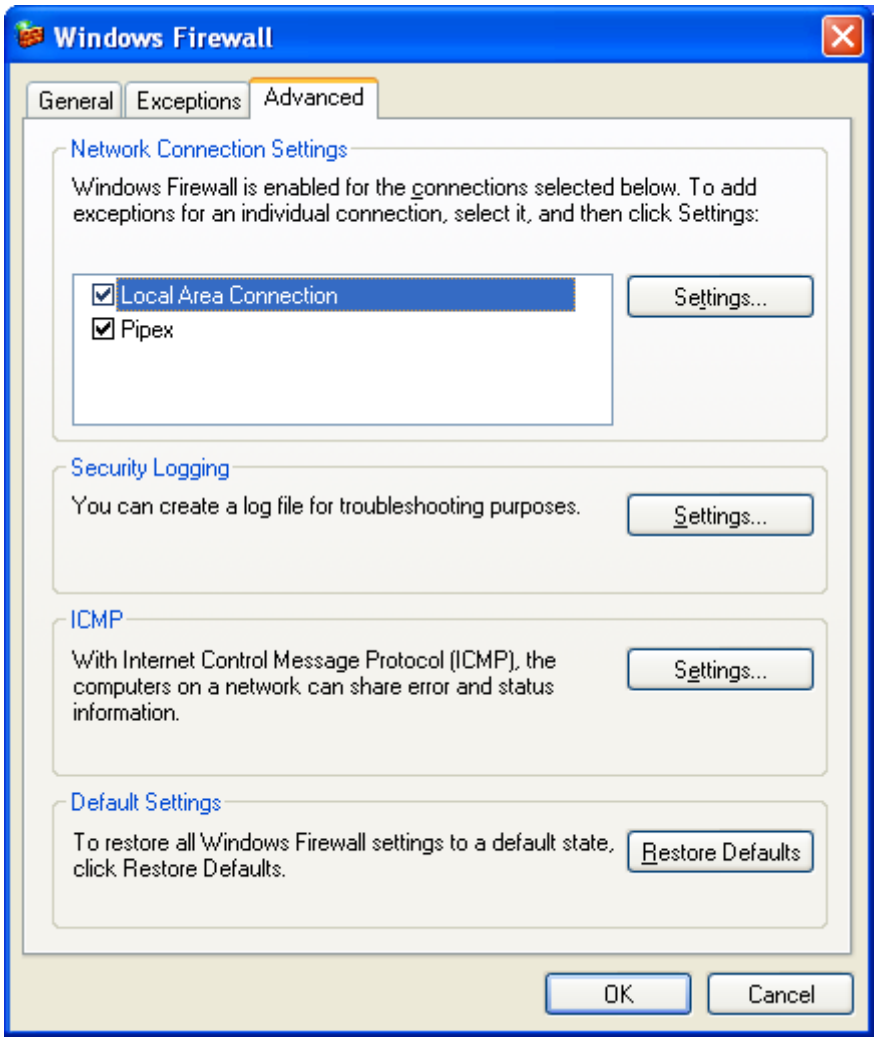
Windows XP Professional and Service Pack 2

A firewall, which is on by default, is included in Service Pack 2 for Windows XP Professional. You will need to modify its settings to allow other intranet and/or Internet users to access your HTTP server. Go to Control Panel and double-click on Security Center:

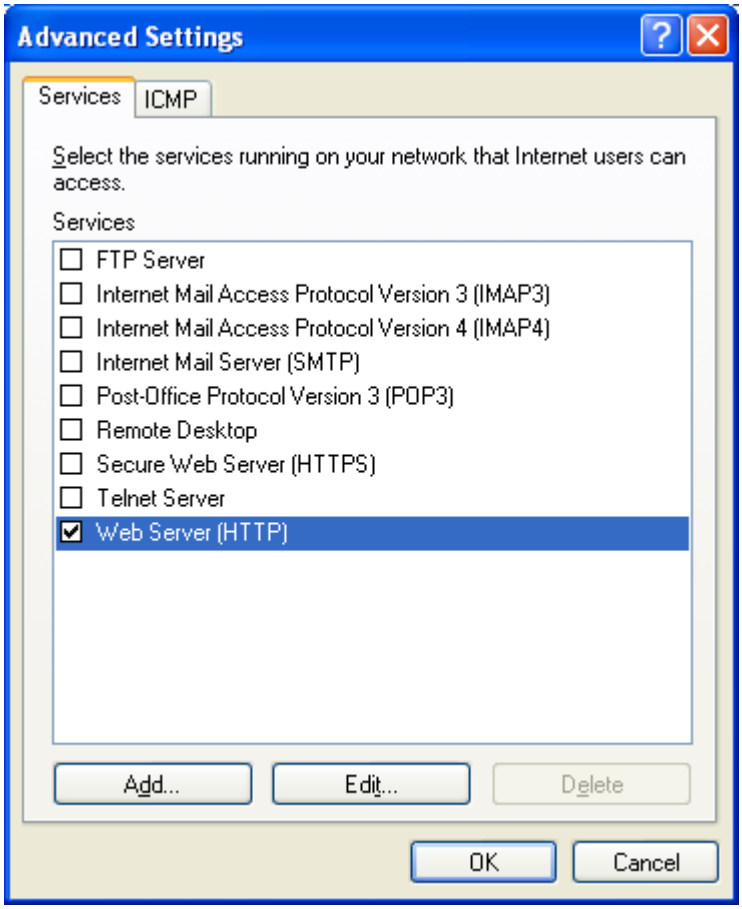


In the Windows Security Center dialog, click on Windows Firewall at the bottom of the dialog:

In the Windows Firewall dialog, you will see Local Area Connection and your Internet connection (not necessarily Pipex). Select Local Area Connection and click on the Settings button:



In the Advanced Settings dialog, select the Web Server (HTTP) option:



Do the same again for your Internet connection as required.

Chapter 26 - Initialisation File

This chapter describes all the ProWeb initialisation settings.

Introduction

ProWeb gets the initial values for its settings (1) from proweb_setting/2 calls (2) from its PROWEB.INI file, or (3) by using their default values.

Microsoft Windows, owing to its MS-DOS origin, uses the back-slash character as a directory separator. The Internet itself, owing to its Unix origin, uses the (forward) slash character to separate directories. However, Windows is able to handle the Unix notation in addition to its own. ProWeb, when constructing a pathname itself, uses the Unix notation.

Note: As some Internet server software, such as Netscape's FastTrack Server, is case sensitive, it is best to standardise on the case you wish to use for pathnames etc.

A setting is in the [PROWEB] section of the .INI file unless stated otherwise.

Overview

Version 4.31 **PROWEB.INI Section and Setting, Type and Default**	**Description**
SETTING: DATE (date_format) TYPE: string DEFAULT: '[dd]/[mm]/[yyyy]'	The format used when printing or inputting date values.
SETTING: TIME (time_format) TYPE: string DEFAULT: '[hh24]:[mm]:[ss]'	The format used when printing or inputting time values.
SETTING: CURRENCY_SIGN (currency_sign) TYPE: atom DEFAULT: signed	Method by which negative/positive currency values are printed.

SETTING: CURRENCY_SYMBOL (currency_symbol) TYPE: string DEFAULT: `$`	The character(s) used when printing a currency value.
SETTING: CURRENCY_THOUSAND (currency_thousand) TYPE: string DEFAULT: `,`	The character(s) used, normally a comma, when printing thousands.
SETTING: CURRENCY_DECIMAL (currency_decimal) TYPE: string DEFAULT: `.`	The character(s) used, normally a period, immediately prior to the decimal part.
SETTING: LOAD_GOAL (load_goal) TYPE: term DEFAULT: true	Application's pre-goal.
SETTING: MAIN_GOAL (main_goal) TYPE: term DEFAULT: true	A term which includes the name of the predicate which constitutes the main goal of the user's ProWeb application.
SETTING: POST_GOAL (post_goal) TYPE: term DEFAULT: true	Names the predicate which constitutes the post goal after termination of the main goal.
SETTING: MAX_FORMS_PER_PAGE(max_forms_per_page) TYPE: integer >= 0 DEFAULT: 0	Specifies the maximum number of forms sent (by the Server) on a single HTML page to the Client.

SETTING: HELP (help) TYPE: on/off DEFAULT: off	Allows in-line help buttons to be attached to HTML questions.
SETTING: CACHE_HELP (cache_help) TYPE: on/off DEFAULT: off	Cache the help text associated with a question. For example, the text may have been retrieved from an ODBC data source.
SETTING: CACHE_HTML (cache_html) TYPE: on/off DEFAULT: off	Cache the Prolog term representation of an HTML file; for instance, when deployed with static HTML template files, there is no reason to re-compute the mapping into Prolog terms.
SETTING: HTML_PATH (html_path) TYPE: atom DEFAULT: 'C:/HTML/'	The HTML path specifies the directory where template HTML files are kept.
SETTING: TEMP_PATH (temp_path) TYPE: atom DEFAULT: 'C:/TEMP/'	The TEMP path specifies the local directory where conversation database files (.WWW) and the temporary files making up a frameset are kept. This setting should be set to the pathname equivalent of temp_url.
SETTING: BUFFER_HEAP (buffer_heap) TYPE: integer DEFAULT: 1	The maximum number of buffers thus limiting the number of simultaneous invocations.
SETTING: BUFFER_SIZE (buffer_size) TYPE: integer DEFAULT: 65535	Specifies the size of the shared buffer. This setting is only applicable to live mode; it has no relevance to the DDE link mode or the copy and paste method.
SETTING: BUFFER_ROOT (buffer_root) TYPE: string DEFAULT: `proweb`	Specifies the root name of the shared buffer between PROWEB.EXE and PROWEB.SYS. This setting is only applicable to live mode where it is used by PROWEB.EXE; it has no relevance to the DDE link mode or the copy and paste method.

SETTING: MUTEX_TIMEOUT (mutex_timeout) TYPE: integer > 0 DEFAULT: 60	The number of seconds PROWEB.SYS waits for the mutex to become available.
SETTING: START_TIMEOUT (start_timeout) TYPE: integer > 0 DEFAULT: 60	Imposes the maximum time allowed (in seconds) before PROWEB.EXE abandons starting PROWEB.SYS.
SETTING: QUERY_TIMEOUT (query_timeout) TYPE: integer > 0 DEFAULT: 60	Imposes the maximum time allowed (in seconds) to execute the main goal and issue a response.
SETTING: TEMP_TIMEOUT (temp_timeout) TYPE: integer > 0 DEFAULT: 3600	Imposes the maximum time allowed (in seconds) for the Client to respond to the last HTML page (within a particular conversation) sent by the Server; after which the conversation database may be discarded.
SETTING: IDLE_TIMEOUT (idle_timeout) TYPE: integer > 0 DEFAULT: 3600	Imposes the maximum time allowed (in seconds) for the executable to remain idle, after which it kills itself.
SETTING: TEMP_MAXSIZE (temp_maxsize) TYPE: integer DEFAULT: 3000	Maximum size of a .LOG file in the temp path.
SETTING: CGI_FILE_EXTENSION (cgi_file_extension) TYPE: atom DEFAULT: 'EXE'	Extension for PROWEB.EXE
SETTING: error_head	The standard system error page
SETTING: error_body	The standard system error page

SETTING: BASE_URL (base_url) TYPE: atom DEFAULT: 'HTTP://BASE_URL/'	The URL of ProWeb's home directory. The value of the BASE_URL setting forms the beginning of the ACTION attribute of the <FORM> tag. If your ProWeb alias is '/PROWEB' (as specified in the Internet Service Manager), it is best to set BASE_URL to 'http://<my_server_name>/proweb/'. The leaf name of the ProWeb executable should not be stated; '/proweb.exe' is added automatically.
SETTING: TEMP_URL (temp_url) TYPE: atom DEFAULT: 'HTTP://TEMP_URL/'	URL of ProWeb's temporary directory. This setting should be set to the URL equivalent of temp_path.
SETTING: SRC (conversation_src) TYPE: server/client DEFAULT: server	Tells ProWeb where to store the details of the conversation; determines whether the conversation database is stored on the Server or Client machine.
SECTION: [PRO386W] SETTING: COMMAND TYPE: - DEFAULT: -	Equivalent of the WIN-PROLOG command line. WIN-PROLOG switches and goals may be placed here.

The load_goal Setting

The value of load_goal is a Prolog term that you want to execute.

```
[PROWEB]
LOAD_GOAL= ensure_loaded( examples(pw_demo) )
```

The main_goal Setting

The value of main_goal can be a single Prolog term that you want to execute. Such a Prolog term would be the main entry point into your ProWeb application.

```
[PROWEB]
MAIN_GOAL = main_goal
```

You can, however, include other goal terms in the main_goal setting. For example, you may want to load a file (e.g. my_loader.pl) which defines a different 'main_goal' (e.g. my_main_goal):

```
[PROWEB]
MAIN_GOAL = ensure_loaded( prolog(my_loader) ), my_main_goal
```

The above example is effectively the same as:

```
[PROWEB]
LOAD_GOAL = ensure_loaded( examples(pw_demo) )
MAIN_GOAL = main_goal
```

The post_goal Setting

Between completion of the 'main goal' and an HTML page being sent to the client, a 'post goal' can be run. A 'post goal' can be used to modify the HTML page about to be sent to a client. The value of post_goal contains the name of a Prolog predicate with zero arity in the source code of the ProWeb-aware application that ProWeb will execute once the main goal has finished executing:

```
[PROWEB]
POST_GOAL=post_goal
```

or

```
[PROWEB]
POST_GOAL=true
```

Rather than have a default 'post goal', ProWeb allows us to specify any predicate with zero arity we like:

```
?- proweb_setting( post_goal, my_post_goal ). <enter>
```

The following program ear-marks the *main_goal_form* for sending to the client. Prior to sending the HTML page containing this form, *post_goal/0* is executed and succeeds, resulting in the *post_goal_form* being appended to the HTML page. Upon submission of this page by the client, *main_goal/0* and *post_goal/0* are executed from the beginning again. Although *main_goal/0* succeeds, the *proweb_unreturned_form(main_goal_form)* line in *post_goal/0* fails, thereby preventing the *post_goal_form* from being appended to a page in this conversation ever again:

```
main_goal :-
        proweb_resend_form(main_goal_form).

post_goal :-
        proweb_unreturned_form(main_goal_form),
        proweb_send_form(post_goal_form).

proweb_page( _, [] ).

proweb_form(        main_goal_form,
                    [ input(type=submit), p ]
                ).
```

```
proweb_form(        post_goal_form,
                    [ `Click the above button to submit this page!` ]
                  ).

proweb_friends( main_goal_form, post_goal_form ).
```

The max_forms_per_page Setting

The max_forms_per_page setting specifies the maximum number of forms that can be sent to a client within a single HTML page.

A *max_forms_per_page* value of:

0 (the default value) means process associated forms together with no maximum number of forms per page;

1 means process a single form at a time; and

n means process associated forms together with a maximum of n forms per page.

The help Setting

The help setting toggles on or off whether in-line help buttons can be attached to questions. You will get '???<name>' in any ProWeb-generated HTML page if this Help setting is missing from your .INI file.

The cache_help Setting

The cache_help setting toggles on or off whether help text associated with questions is to be cached or not. This is especially useful if the text has been retrieved from an ODBC data source.

The cache_html Setting

The cache_html setting toggles on or off whether the Prolog term representation of an HTML file is to be cached or not. This is especially useful when static HTML template files are being used as it saves having to re-compute the mapping into Prolog terms.

The base_url Setting

The base_url setting is the URL of the HOME machine. The value of the BASE_URL setting forms the beginning of the ACTION attribute of a <FORM> tag. If your ProWeb alias is '/PROWEB', it is best to set BASE_URL to 'http://<my_server_name>/proweb/'.

```
BASE_URL = http://localhost/proweb/
```

The temp_url Setting

The temp_url setting is the URL of ProWeb's temporary directory. Even if your ProWeb alias is '/PROWEB' and ProWeb's temporary directory is relative to this, , it is best to set TEMP_URL to 'http://<my_server_name>/proweb/temp/'.

TEMP_URL = http://localhost/proweb/temp/

The html_path Setting

Whenever an HTML page is partly-created from an HTML template file on disk, only its leaf name needs to be stated in the Prolog source code; this is done via an *include/1* call within a *proweb_page/2* or *proweb_form/2* clause. The pathname of such HTML template files is obtained from the ProWeb setting, *html_path*. The value of html_path points to the directory containing your HTML template files. If you wanted *html_path* to point to C:\INETPUB\PROWEB\HTML, allowable values would be:

'C:\INETPUB\PROWEB\HTML\'

If you wanted to use the template HTML file, C:\INETPUB\PROWEB\HTML\HEAD.HTM, you would set *html_path* to 'C:\INETPUB\PROWEB\HTML\':

proweb_setting(*html_path*, 'c:\inetpub\proweb\html\').

and use the line:

include('HEAD.HTM')

within your Prolog source code.

The temp_path Setting

The value of temp_path points to ProWeb's temporary directory. If you wish to set C:\INETPUB\PROWEB\TEMP as your ProWeb temporary directory, possible values for temp_path are:

'C:\INETPUB\PROWEB\TEMP\'

If writing the path using DOS notation, always include a '\' at the end, otherwise you might end up with a file whose path is 'C:\INETPUB\PROWEB\TEMP/00000001.WWW'; although this work at the moment in both *WIN-PROLOG* and Windows, it may cause an error in the future.

The buffer_size Setting

The buffer_size setting specifies the size of the shared buffer between PROWEB.EXE and PROWEB.SYS.

The buffer_root Setting

The buffer_root setting specifies the root name of the shared buffer between PROWEB.EXE and PROWEB.SYS. If you were to have two ProWeb setups on the same server, each setup would need a unique buffer root name to prevent conversations moving from one setup to the other.

The mutex_timeout Setting

The mutex_timeout setting specifies the number of seconds PROWEB.SYS will wait for the mutex to become available. A mutex guarantees that two or more operations are mutually exclusive.

The start_timeout Setting

The start_timeout setting specifies the number of seconds PROWEB.EXE abandons trying to start PROWEB.SYS.

The query_timeout Setting

The query timeout setting imposes the maximum number of seconds allowed to execute the main goal. Whenever PROWEB.EXE instructs PROWEB.SYS (the renamed PRO386W.EXE file) to execute a goal, it will only wait the period of time as per the query_timeout setting, afterwhich it will issue a 'time out' page. As PROWEB.SYS can only execute one client's goal at a time, this setting prevents other clients from being 'locked out' for too long. You can give the query timeout setting a value greater than zero, the default being 60 (1 minute). Obviously, this setting can not be given a value less than the time it takes to complete your application's longest process.

When loading a large database such as WordNet into a ProWeb application, the allocation of memory to ***WIN-PROLOG*** itself needs to be large (hence the *xinit/9* directive), but you must also ensure that ProWeb is given enough time to load the WordNet files. The two ProWeb settings that need to be increased are the start_timeout and the query_timeout. These only need to be of a large size when ProWeb starts up, after which they can return to normal. The following WordNet loader for ProWeb temporarily increases these two ProWeb settings, loads the WordNet files and then returns the two ProWeb settings back to normal:

```
:- xinit( 250, 250, -1, 16000, 17535, 56000, -1, -1, -1 ).

main_goal :-
  load_wordnet,
  …
```

```
load_wordnet :-
  proweb_setting( start_timeout, OldStartTimeoutSetting ),
  proweb_setting( query_timeout, OldQueryTimeoutSetting ),
  proweb_setting( start_timeout, 25000 ),
  proweb_setting( query_timeout, 25000 ),
  forall( member( File, [ 'wordnet\wn_g.pl',
                          'wordnet\wn_s.pl',
                          'wordnet\wn_hyp.pl',
                          'wordnet\wn_sim.pl',
                          'wordnet\wn_fr.pl',
                          'wordnet\wn_mm.pl',
                          'wordnet\wn_ant.pl',
                          'wordnet\wn_per.pl',
                          'wordnet\wn_mp.pl',
                          'wordnet\wn_sa.pl',
                          'wordnet\wn_at.pl',
                          'wordnet\wn_vgp.pl',
                          'wordnet\wn_ent.pl',
                          'wordnet\wn_cs.pl',
                          'wordnet\wn_ppl.pl',
                          'wordnet\wn_ms.pl'
                        ] ),
        ensure_loaded( examples(File) )
      ),
  proweb_setting( start_timeout, OldStartTimeoutSetting ),
  proweb_setting( query_timeout, OldQueryTimeoutSetting ).
```

The temp_timeout Setting

The temp_timeout setting imposes the maximum number of seconds allowed for a client to respond to the server.

Every interaction of a conversation generates a conversational database file; should a client respond in the time allowed, the relevant file will be retrieved from the hard disk. If the client does not respond in the time allowed, the relevant file may have been deleted, resulting in a 'time out' page being issued. You can give the temp timeout setting a value greater than zero, the default being 3600 (1 hour). If the required hard disk space is not available to store such conversational database files, this setting will have to be given a lower value, and should more clients visit your web site a choice will have to be made as to whether to lower it still further or get a bigger hard disk.

Following each client-server interaction, a fresh timer is activated. If a running timer reaches the value of temp_timeout, the user-defined *proweb_temp_timeout/1* clause (if it exists) will be called. Such a user-defined *proweb_temp_timeout/1* clause can, therefore, be called periodically to allow your ProWeb-based application to clear expired items from its own wastebasket. The following example deletes ProWeb's proweb.log file:

```
proweb_temp_timeout( Seconds ) :-
  File = 'c:\inetpub\proweb\temp\proweb.log',
  dir( File, -55, X ),
  (  X \= []
  -> del( File )
  ;  true
  ).
```

Note: the value of temp_timeout must be less than the value of idle_timeout, otherwise the user-defined *proweb_temp_timeout/1* clause will never be called.

The idle_timeout Setting

The idle_timeout setting specifies the maximum number of seconds allowed for PROWEB.EXE and PROWEB.SYS (the renamed PRO386W.EXE file) to remain idle; after which time they will automatically shut themselves down. You can give the idle timeout setting a value greater than zero, the default being 3600 (1 hour). As PROWEB.EXE and PROWEB.SYS take a while to load, this setting should be set high enough that ProWeb is not constantly timing out and having to be reloaded and low enough that you are not waiting ages for ProWeb to time out should something go wrong.

The command Setting

Many of ***WIN-PROLOG***'s default settings can be altered by adding a command line switch such as /b4096 to the ***WIN-PROLOG*** shortcut. As it is not possible for PROWEB.EXE to launch PROWEB.SYS (PRO386W.EXE renamed) via a shortcut, such command line switches are placed in PROWEB.INI instead. Such command line switches will be picked-up by PROWEB.SYS from the PROWEB.INI file and added at startup time.

To set Prolog's backtrack stack to 4096K, for example, you would manually add an appropriate COMMAND value to the PRO386W section of your PROWEB.INI file (there may already be other items in the PRO386W section):

```
[PRO386W]
COMMAND=/b4096
```

ProWeb has the ability to output tracing information to a file named PROWEB.LOG in ProWeb's temporary directory. Such information is surpressed if ***WIN-PROLOG***'s 'y' switch is set to 0. To enable such output, set ***WIN-PROLOG***'s 'y' switch to an integer greater than 0 by adding the following to the COMMAND value in the PROWEB.INI file:

```
[PRO386W]
COMMAND=/y1
```

It is not possible to set the value of switches using *proweb_setting/2*.

The conversation_src Setting

Whilst a client-server conversation is taking place, ProWeb maintains a conversation database; each conversation database, a unique one for each client, stores the following information:

- The name of the last page sent,

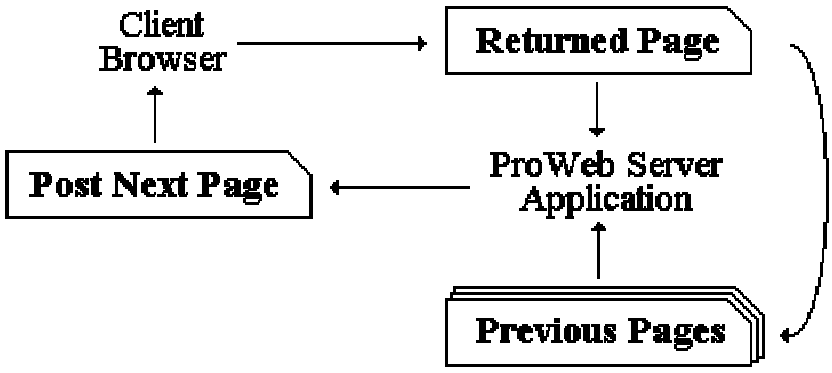
- The names of all previous pages sent,

- Details of the data entered into each submitted form, and

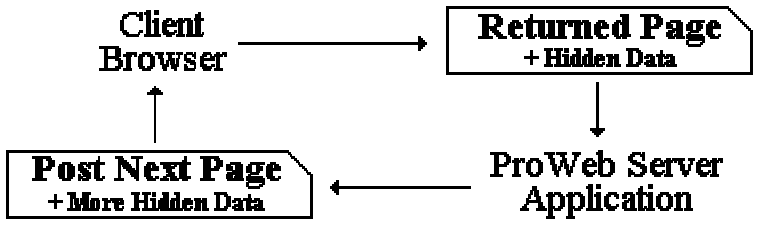
- Details of every assertion or retraction made.

ProWeb is able to store all the conversation databases in one of two locations, either locally or remotely:

Local Conversation Database - Associated with every HTML page sent by a ProWeb server to a client is a hidden pointer. When a page is submitted by that client and returned to the server this pointer is used to reference a unique, local conversation database which recorded the conversation leading up to the page being sent. Thus, when the information extracted from a returned page is added to this prior information the complete picture of the conversation so far is readily available.



Remote Conversation Database - Associated with every HTML page sent by a ProWeb server to a client is a collection of hidden question-answer pairs. These correspond to the information submitted by the client in all of the preceding HTML pages. When this page is submitted by the client all of the hidden information is also returned to the server.



The ProWeb *conversation_src* setting dictates which of the above methods is actually used; a value of *server* indicates that the conversation databases will be stored locally (i.e. saved as a temporary .WWW file on the server's hard disk), whilst a value of *client* indicates that the conversation databases will be stored remotely (i.e. embedded within each HTML page sent to the client). Throughout this user guide the default method, *server*, is assumed.

Saving the data for a conversation in this manner is a very elegant solution as very little information needs to be saved to each file and, as a result, it only takes a very short time to restore the Prolog environment between clients.

The date Setting

The date setting is defined in the PROWEB.INI file as:

```
[PROWEB]
DATE = [WW], the [DD] of [MM], [yyyy]
```

The corresponding *proweb_setting/2* call would be:

```
proweb_setting(date_format,`[WW], the [DD] of [MM],[yyyy]`).
```

The time Setting

The time setting is defined in the PROWEB.INI file as:

```
[PROWEB]
TIME=[hh12]:[mm]:[ss] [ampm]
```

The corresponding *proweb_setting/2* call would be:

```
proweb_setting(time_format,`[hh12]:[mm]:[ss] [ampm]`).
```

The currency_sign Setting

Example: CURRENCY_SIGN=bracketed

The currency_symbol Setting

Example: CURRENCY_SYMBOL=£

The currency_thousand Setting

Example: CURRENCY_THOUSAND=,

The currency_decimal Setting

Example: CURRENCY_DECIMAL=.

Chapter 27 - ProWeb Predicates

This chapter documents all the ProWeb predicates.

html_print/1

html_print(Tag)

+Tag	<atom> or <term>	

Outputs real HTML code from ProWeb's pseudo-HTML code.

See Also: *html_print_nl/1*

html_print_nl/1

html_print_nl(Tag)

+Tag	<atom> or <term>	

Outputs real HTML code from ProWeb's pseudo-HTML code, followed by a new line.

See Also: *html_print/1*

proweb_aborted/0

proweb_aborted

A *proweb_aborted/0* clause can be user-defined; this will be called whenever your ProWeb application aborts.

See Also: *proweb_shutdown/0*

proweb_answered/1

proweb_answered(Question)

+Question	<term>	A question which appears in a previously sent form

proweb_asked/1

proweb_asked(Question)

+Question	<term>	A question which appears in a previously

sent form

proweb_assert/1

proweb_assert(Fact)

+Fact	<goal>	Fact to be asserted

The predicate, *proweb_assert/1*, asserts a fact into Prolog's internal database. If you wish to make an assertion which you intend to use later as a reply, then it is better to use *proweb_post_reply/2*.

A *proweb_unique_assert/1* clause could be defined as:

```
proweb_unique_assert( X ) :-
  (     proweb_call(X)
  ->    true
  ;     proweb_assert(X)
  ).
```

See Also: *proweb_asserta/1*, *proweb_assertz/1*, *proweb_call/1*, *proweb_dynamic/1*, *proweb_retract/1*, *proweb_retractall/1*

proweb_asserta/1

proweb_asserta(Fact)

+Fact	<goal>	Fact to be asserted

The predicate, *proweb_asserta/1*, asserts a fact into Prolog's internal database. If you wish to make an assertion which you intend to use later as a reply, then it is better to use *proweb_post_reply/2*.

See Also: *proweb_assert/1*, *proweb_assertz/1*, *proweb_call/1*, *proweb_dynamic/1*, *proweb_retract/1*, *proweb_retractall/1*

proweb_assertz/1

proweb_assertz(Fact)

+Fact	<goal>	Fact to be asserted

The predicate, *proweb_assertz/1*, asserts a fact into Prolog's internal database. If you wish to make an assertion which you intend to use later as a reply, then it is better to use *proweb_post_reply/2*.

See Also: *proweb_assert/1*, *proweb_asserta/1*, *proweb_call/1*, *proweb_dynamic/1*, *proweb_retract/1*, *proweb_retractall/1*

proweb_call/1

proweb_call(Fact)

?Fact	<goal>	Asserted fact

Retrieves a fact asserted using *proweb_assert/1*, *proweb_asserta/1* or *proweb_assertz/1*.

See Also: *proweb_assert/1*, *proweb_asserta/1*, *proweb_assertz/1*, *proweb_dynamic/1*

proweb_dynamic/1

proweb_dynamic(Predicate/Arity)

+Predicate	<atom>	Name of dynamic predicate
+Arity	<integer>	Its arity

The *proweb_dynamic/1* predicate can be used to declare a ProWeb predicate as dynamic, as the following program demonstrates:

```
main_goal :-
  proweb_dynamic( my_proweb_dynamic_pred/1 ),
  proweb_assert( my_proweb_dynamic_pred(dummy) ),
  proweb_call( my_proweb_dynamic_pred(_) ),
  !,
  proweb_send_form( my_form(`A my_proweb_dynamic_pred/1 clause exists!`)
).

main_goal :-
    proweb_send_form( my_form(`No my_proweb_dynamic_pred/1 clause
exists!`) ).

proweb_page( _, [] ).

proweb_form( my_form(Text), [ Text ] ).
```

The line, *proweb_send_form(my_form(`A my_proweb_dynamic_pred/1 clause exists!`))*, in the first *main_goal/0* clause will be executed if *proweb_assert(my_proweb_dynamic_pred(dummy))* succeeds. The second *main_goal/0* clause will be executed only if the line, *proweb_assert(my_proweb_dynamic_pred(dummy))*, in the first *main_goal/0* clause fails.

See Also: *proweb_assert/1, proweb_asserta/1, proweb_assertz/1, proweb_call/1, proweb_retract/1, proweb_retractall/1*

proweb_executable/3

proweb_executable(Path, Name, Ext)

-Path	<atom>	The path of the executable
-Name	<atom>	The name of the executable
-Ext	<atom>	The extension of the executable

The predicate, *proweb_executable/3*, can be used to check or get the name of the ProWeb executable.

See Also:

proweb_expansion/3

proweb_expansion(Term, Form, Fields)

+Term	<term>	Any term
+Form	<term>	User-defined ProWeb form
-Fields	<list>	List of expanded fields

Prior to a form being sent, a term may be encountered during the expansion stage. To expand the term, ProWeb will search for a user-defined *proweb_expansion/3* clause where the first argument matches the term and the second matches the form. The following program will expand *get_nodes* into *[a,b,c,d]*:

```
main_goal :-
  proweb_send_form( my_form ).

proweb_page( [my_form], [] ).

proweb_form( my_form, get_nodes ).

proweb_expansion( get_nodes, my_form, verbatim @ Nodes ) :-
  setof( Node, A^linked(Node,A), Nodes ).
```

```
linked(X,Y) :-
  arc(X,Y);arc(Y,X).

arc(a,c).
arc(b,c).
arc(b,d).
```

See Also: *proweb_form/2*

proweb_forget_answer/1

proweb_forget_answer(Answer)

+Answer <atom> or <term>

The ProWeb predicate, *proweb_forget_answer/1*, can be used to force ProWeb to forget an answer 'picked-up' via a *proweb_returned_answer/2* call.

See Also: *proweb_question/2, proweb_returned_answer/2*

proweb_forget_forms_sent/0

proweb_forget_forms_sent

The predicate, *proweb_forget_forms_sent/0*, is called within the user's SERVER program when the application needs to re-start itself allowing the new versions of previously sent forms to be sent again!

See Also:

proweb_forget_preset/1

proweb_forget_preset(Preset)

+Preset <atom> or <term>

The ProWeb predicate, *proweb_forget_preset/1*, can be used to force ProWeb to forget a preset answer set via a *proweb_preset_answer/2* call.

For example, executing the following from the ***WIN-PROLOG*** console window:

?- proweb_forget_preset(start_node).<enter>

will forget the preset answer for *start_node* set earlier with

proweb_preset_answer(start_node, b).

See Also: *proweb_question/2, proweb_preset_answer/2*

proweb_form/2

proweb_form(Form, Fields) [:- Body]

+Form	<atom> or <term>	User-defined ProWeb form
-Fields	<atom> or <list>	List of ProWeb fields
-Body	<clause_body>	

See Also: *proweb_expansion/3, proweb_friends/2, proweb_help/3, proweb_page/2, proweb_question/2, proweb_resend_form/1, proweb_send_form/1, proweb_send_unique_form/1*

proweb_frameset/2

proweb_frameset(Forms, Page)

+Forms	<list>	The forms on this page
-Page	<list> or <term>	HTML page hosting the forms

See Also: *proweb_form/2*, *proweb_page/2*

proweb_friends/2

proweb_friends(Form1, Form2)

+Form1	<atom> or <term>	User-defined ProWeb form
-Form2	<atom> or <term>	User-defined ProWeb form

If your ProWeb application is particularly complex with many forms on as many pages, you may like to allow the client to receive all the forms in one go, or within as few pages as possible. With ProWeb, a pair of forms can be declared "friendly", and when the opportunity arises, ProWeb will combine the forms together within the same HTML page. For example, running the following program will return the first page; and when submitted, the second page:

```
main_goal :-
  proweb_send_form(first_form),
  proweb_send_form(second_form).

proweb_form(        first_form, [ `Submit first form: `,
                          input(type=submit, value=`Submit`),
                          p
                        ] ).

proweb_form(        second_form, [ `Submit second form: `,
                            input(type=submit, value=`Submit`)
                          ] ).
```

but adding the clause:

```
proweb_friends( first_form, second_form ).
```

where each argument is the name of a user-defined ProWeb form will cause the two forms to be combined on the same HTML page.

To allow more than one form to be sent per page, the ProWeb *max_forms_per_page* setting will need to be set appropriately:

```
?- proweb_setting( max_forms_per_page, 2 ).<enter>
```

Although this will not normally cause you a problem, it is important to note that the resulting HTML page only has one set of <FORM> elements, both forms being placed within the same <FORM> element.

See Also: *proweb_form/2*

proweb_help/3

proweb_help(Question, HelpButton, HelpString)

+Question	<term>	User-defined ProWeb question
–HelpButton	<term>	Name of the help button
-HelpString	<term>	Associated help string

See Also: *proweb_form/2*

proweb_operators/0

proweb_operators

This predicate defines the Prolog operators used by Proweb. When creating/modifying your Proweb programs within the ***WIN-PROLOG*** development environment, it is advisable to call *proweb_operators* first:

```
:- proweb_operators.
```

proweb_page/2

proweb_page(Form, Page)

+Form	<atom> or <list>	The form(s) on this page
-Page	<list> or <term>	HTML page hosting the form(s)

See Also: *proweb_form/2, proweb_resend_form/1, proweb_send_form/1, proweb_send_unique_form/1*

proweb_posted_reply/2

proweb_posted_reply(Reply, Value)

?Reply	<atom> or <term>	Reply to be put into a form
?Value	<atom> or <term>	Value of the reply

The predicate, *proweb_post_reply/2*, stores a reply in Prolog's internal database. You can check all such replies posted to Prolog's internal database by executing *proweb_posted_reply/2*:

```
?- proweb_posted_reply( Reply, Value ). <enter>
Reply = solution ,
Value = verbatim @ [a,c,e,f,g,i] ;
no
?-
```

See Also: *proweb_post_reply/2, proweb_post_unique_reply/2*

proweb_post_reply/2

proweb_post_reply(Reply, Value)

+Reply	<atom> or <term>	Reply to be put into a form
+Value	<atom> or <term>	Value of the reply

The predicate, *proweb_post_reply/2*, stores a reply, which persists for the current interaction only, in Prolog's internal database. Such an assertion will replace a corresponding <PROWEB REPLY="..."> tag when encountered in a template HTML page.

See Also: *proweb_posted_reply/2, proweb_post_unique_reply/2*

proweb_post_unique_reply/2

proweb_post_unique_reply(Reply, Value)

+Reply	<atom> or <term>	Reply to be put into a form
+Value	<atom> or <term>	Value of the reply

See Also: *proweb_post_reply/2, proweb_posted_reply/2*

proweb_preset_answer/2

proweb_preset_answer(Question, Answer)

+Question	<atom> or <term>
+Answer	<atom> or <term>

See Also: *proweb_forget_preset/1, proweb_question/2*

proweb_question/2

proweb_question(Question, Parameters) [:- Body]

+Question	<atom> or <term>	User-defined ProWeb question
-Parameters	<list>	Valid question parameters
-Body	<clause_body>	

See Also: *proweb_form/2*

proweb_resend_form/1

proweb_resend_form(Form)

+Form	<atom> or <term>	User-defined ProWeb form

See Also: *proweb_form/2, proweb_send_form/1, proweb_send_unique_form/1*

proweb_retract/1

proweb_retract(Fact)

+Fact	<goal>	Asserted fact

See Also: *proweb_assert/1, proweb_asserta/1, proweb_assertz/1, proweb_dynamic/1, proweb_retractall/1*

proweb_retractall/1

proweb_retractall(Fact)

+Fact	<goal>	Asserted fact(s)

See Also: *proweb_assert/1, proweb_asserta/1, proweb_assertz/1, proweb_dynamic/1, proweb_retract/1*

proweb_returned_answer/2

proweb_returned_answer(Question, Answer)

+Question	<atom> or <term>	A question which appears in a previously sent form
?Answer	<term> or <variable>	The answer given for that particular question

See Also: *proweb_forget_answer/1, proweb_question/2*

proweb_returned_input/2

proweb_returned_input(Name, Input)

+Name	<atom>	
-Input	<variable>	

See Also: *proweb_assert/1, proweb_call/1*

proweb_returned_form/1

proweb_returned_form(Form)

+Form	<atom> or <term>	User-defined form	ProWeb

See Also: *proweb_send_form/1, proweb_unreturned_form/1*

proweb_send_content/1

proweb_send_content(Content)

+Content	<string>	Actual content returned

The predicate, *proweb_send_content/1*, is called within the user's SERVER program and immediately returns the content string directly to the client:

The command:

proweb_send_content(`This is some text`).

would result in the following page being sent to the client:

CGI Error

The specified CGI application misbehaved by not returning a complete set of HTTP headers. The headers it did return are:

This is some text

proweb_send_form/1

proweb_send_form(Form)

+Form	<atom> or <term>	User-defined form	ProWeb

See Also: *proweb_form/2, proweb_resend_form/1, proweb_send_unique_form/1*

proweb_send_unique_form/1

proweb_send_unique_form(Form)

+Form	<atom> or <term>	User-defined ProWeb form

See Also: *proweb_form/2, proweb_resend_form/1*, *proweb_send_form/1*

proweb_set_answer/2

proweb_set_answer(Question, Answer)

+Question	<term>	A question which appears in some form
+Answer	<term>	The preset answer for that particular question

proweb_setting/2

proweb_setting(Setting, Value)

?Setting	<atom> or <variable>	A valid ProWeb setting
?Value	<atom>, <integer> or <variable>	A valid setting value

See Also:

proweb_shutdown/0

proweb_shutdown

If a *proweb_shutdown/0* clause has been user-defined, it will be executed automatically during the shutdown phase of your live ProWeb application (i.e. everytime PROWEB.SYS closes down). A *proweb_shutdown/0* clause, which utters three beeps, could be user-defined as follows:

```
proweb_shutdown :-
   beep(260,250),
   beep(230,250),
   beep(200,250).
```

See Also: *proweb_aborted/0*

proweb_startup/0

proweb_startup

See Also:

proweb_string/2

proweb_string(Category, String).

?Category	<atom> in the domain {title, version, build, copyright}	A ProWeb string category
?String	<string>	The string itself

See Also:

proweb_temp_timeout/1

proweb_temp_timeout(Value)

-Value	<integer>

See Also:

proweb_text/2

proweb_text(Name, Value)

+Name	<atom>
+Value	<string>

The '&&' notation allows you to insert text, defined in a user-defined *proweb_text/2* fact, at a specific place within an HTML page.

```
:- multifile proweb_form/2.
:- multifile proweb_page/2.

main_goal :-
  proweb_send_form( my_form ).

proweb_page( _, [] ).
```

```
proweb_form( my_form,
        [ &&(canned_text1),
          br,
          &&(canned_text2)
        ]
      ).

proweb_text( canned_text1, `This is some canned text` ).
proweb_text( canned_text2, `This is some more canned text` ).
```

proweb_trace/1

proweb_trace(Trace)

+Trace	<atom>, <string> or <term>	

See Also:

proweb_unreturned_form/1

proweb_unreturned_form(Form)

+Form	<atom> or <term>	User-defined ProWeb form

See Also: *proweb_returned_form/1*, *proweb_send_form/1*, *proweb_send_unique_form/1*

Declarations

At the top of each ProWeb application's source code file, you will need to declare that the ProWeb predicates you are using in that file are being defined in more than one file; for example:

```
:- multifile( proweb_aborted / 0 ).
:- multifile( proweb_expansion / 3 ).
:- multifile( proweb_form / 2 ).
:- multifile( proweb_frameset / 2 ).
:- multifile( proweb_friends / 2 ).
:- multifile( proweb_help / 3 ).
:- multifile( proweb_page / 2 ).
:- multifile( proweb_question / 2 ).
:- multifile( proweb_shutdown / 0 ).
:- multifile( proweb_startup / 0 ).
:- multifile( proweb_temp_timeout / 1 ).
```

Useage

ProWeb Predicate	User Defined Predicate?
html_print/1	No
html_print_nl/1	No
proweb_aborted/0	Yes
proweb_answered/1	No
proweb_asked/1	No
proweb_assert/1	No
proweb_asserta/1	No
proweb_assertz/1	No
proweb_call/1	No
proweb_dynamic/1	No
proweb_executable/3	No
proweb_expansion/3	Yes
proweb_forget_answer/1	No
proweb_forget_forms_sent/0	No
proweb_forget_preset/1	No
proweb_form/2	Yes
proweb_frameset/2	Yes
proweb_friends/2	Yes
proweb_help/3	Yes
proweb_operators	No
proweb_page/2	Yes
proweb_posted_reply/2	No
proweb_post_reply/2	No

proweb_post_unique_reply/2	No
proweb_preset_answer/2	No
proweb_question/2	Yes
proweb_resend_form/1	No
proweb_retract/1	No
proweb_retractall/1	No
proweb_returned_answer/2	No
proweb_returned_input/2	No
proweb_returned_form/1	No
proweb_send_content/1	No
proweb_send_form/1	No
proweb_send_unique_form/1	No
proweb_setting/2	No
proweb_shutdown/0	Yes
proweb_startup/0	Yes
proweb_string/2	No
proweb_temp_timeout/1	Yes
proweb_text/2	Yes
proweb_trace/1	No
proweb_unreturned_form/1	No

Chapter 28 - Error Messages

This chapter provides explanations of and solutions for the various *WIN-PROLOG* error messages and ProWeb CGI server driver error messages.

WIN-PROLOG Error Messages

Error Message	Solution
34 (File Access Denied)	Have you shared the directory?

ProWeb CGI Server Driver Error Messages

Acronym	Meaning
BRM	Bad "REQUEST_METHOD" env variable
CFM	CreateFileMapping() failed
CLH	CloseHandle() failed
CMX	CreateMutex() failed
ECP	ConPut() failed during ENV processing
ERP	RawPut() failed during ENV processing
ESP	StrPut() failed during ENV processing
GLB	GlobalAlloc() failed
GMF	GetModuleFileName() failed
GUN	GetUserName() failed Make sure you logged onto your computer correctly.
ICP	ConPut() failed during CGI processing
ISP	StrPut() failed during CGI processing
MPF	Main procedure failed
MTO	Mutex Timeout
MVF	MapViewOfFile() failed
NCL	No "CONTENT_LENGTH" env variable

NRM	No "REQUEST_METHOD" env variable
OSB	Output string too big
OSE	Output string error Make sure you have created a temp directory and that it is pointed to correctly by the TEMP_URL setting in the PROWEB.INI file.
QER	Query ProWeb error
QFA	Query ProWeb failure
QTO	Query Timeout
REX	Stop "WinExec() failed" (not actually called)
RFM	Stop CreateFileMapping() failed
RGB	Stop command not "!"
RGC	Stop argument count incorrect
RGV	Stop command value incorrect
RST	Stop sequence completed successfully
RVF	Stop MapViewOfFile() failed
VER	Version Report
WER	WinExec ProWeb error
WEX	WinExec() failed
WFA	WinExec ProWeb failure
WTO	WinExec timeout
XCL	Excessive "CONTENT_LENGTH" env variable
XQS	Excessive "QUERY_STRING" env variable

Acronym in Version 4.31	**Explanation and Solution**
	Completed Query

GMF	Failure in GetModuleFileName()
CMX	Failure in CreateMutex() If you are running two ProWeb setups at the same time on the same server, make sure they have different values for their ProWeb buffer_root settings respectively.
	WAIT_ABANDONED after CreateMutex()
	WAIT_FAILED after CreateMutex()
	WAIT_TIMEOUT after CreateMutex() (1) Increase the value of the ProWeb mutex_timeout setting. (2) When IIS is set up for "anonymous" login, any number of users can access its website simultaneously, and all are logged in as a special user called "IUSR_nnn" where "nnn" is typically the local name of the server. Because each (real) user is simply sharing the single (virtual) user "IUSR_nnn", any resources loaded by one (such as PROWEB.SYS etc), effectively belong to all, so successive people can use the same physical instance of PROWEB. The latter takes care of separating one user's activities from another's. When "anonymous" login is *disabled*, each user logs on as a separate entity; each own his/her own processes (such as PROWEB), and the operating system partitions resources accordingly. The problem that arises is that a MUTEX, such as ProWeb uses, to ensure that two or more instances of PROWEB do not try to run simultaneously, is a GLOBAL resource. If, on a given server, user FRED creates a mutex called "FOO", then another user, MARY, can't see it until FRED releases it. This is what we want. Unfortunately the same is NOT true (by default at least) for a memory mapped file. If user FRED creates it, MARY is prevented access until FRED deletes it. The latter only happens when PROWEB, which FRED loaded into memory, terminates. To recap: when "anonymous" login is enabled, every actual person is seen by the server as the same, single user. When it is disabled, each person is a different user. And file mappings are (in the current version of ProWeb) locked on a per-user access basis.
CFM	Failure in CreateFileMapping() When IIS is set up for "anonymous" login, any number of users can access its website simultaneously, and all are logged in as a special user called "IUSR_nnn" where "nnn" is typically the local name of the server. Because each (real) user is simply sharing the single (virtual) user "IUSR_nnn", any resources loaded by one (such as PROWEB.SYS etc), effectively belong to all, so successive people can use the same physical

	instance of PROWEB. The latter takes care of separating one user's activities from another's. When "anonymous" login is *disabled*, each user logs on as a separate entity; each own his/her own processes (such as PROWEB), and the operating system partitions resources accordingly. The problem that arises is that a MUTEX, such as ProWeb uses to ensure that two or more instances of PROWEB do not try to run simultaneously, is a GLOBAL resource. If, on a given server, user FRED creates a mutex called "FOO", then another user, MARY, can't see it until FRED releases it. This is what we want. Unfortunately the same is NOT true (by default at least) for a memory mapped file. If user FRED creates it, MARY is prevented access until FRED deletes it. The latter only happens when PROWEB, which FRED loaded into memory, terminates. To recap: when "anonymous" login is enabled, every actual person is seen by the server as the same, single user. When it is disabled, each person is a different user. And file mappings are (in the current version of ProWeb) locked on a per-user access basis.
MVF	Failure in MapViewOfFile()
WEX	Failure in WinExec() PROWEB.SYS (a copy of PRO386W.EXE renamed) can not be found. Ensure PROWEB.SYS is in the same directory as PROWEB.EXE. Ensure that it has only one extension and not PROWEB.SYS.EXE or some such.
	"start_timeout" after WinExec() The ProWeb start_timeout setting is set too low for PROWEB.SYS to load completely; increase the ProWeb start_timeout setting in your PROWEB.INI file. Does PROWEB.SYS appear in the Windows Task Manager? If so, is it actually allocated the memory you have asked it to have? Have you made a copy of PRO386W.EXE and renamed it PROWEB.SYS and placed it in the same directory as PROWEB.EXE? Make sure the file really is named 'PROWEB.SYS' and not something like 'PROWEB.SYS.EXE', which can happen if you have file extensions hidden. PROWEB.OVL may be missing; ensure PROWEB.OVL is in the same directory as PROWEB.EXE. If you can see the Prolog console window, has an error message appeared? You may need to be logged on to Windows as the IUSR_<something> user to see the Prolog console window. You could try running PROWEB.SYS directly from the Windows command

	line; make a copy of PROWEB.SYS, PROWEB.OVL and PROWEB.INI, renaming them, say, FRED.EXE, FRED.OVL and FRED.INI respectively.
NRM	Missing "REQUEST_METHOD" environment variable
NCL	Missing "CONTENT_LENGTH" environment variable
	Invalid "REQUEST_METHOD" environment variable
RST	Prolog forced to stop by request You have either asked ProWeb to terminate by issuing a URL such as http://<server_name>/proweb/proweb.exe?!+001 or the syntax of the URL line is incorrect and PROWEB.EXE has interpreted it as a stop request.
QTO	"query_timeout" in Prolog The query being executed in Prolog has timed-out, in accordance with the ProWeb query_timeout setting, before it was finished. Increase the value of the ProWeb query_timeout setting.
OSB	HTML string exceeds the size of the FileMap buffer This error message means that the generated HTML code cannot be passed from PROWEB.SYS to PROWEB.EXE because the shared buffer is too small. Increase the value of the ProWeb buffer_size setting.
	Doomsday scenario The circumstances that generate this error message should NEVER be encountered.
	Error trying to generate a ProWeb built-in error page This error message is issued in preference to going into an infinite loop. Check that you have a ProWeb temporary directory and that its path is correctly stated in the ProWeb temp_path setting. Check that you have not run out of hard disk space.

Appendix A – Undocumented Features Of ProWeb Version 4.31

This appendix lists the undocumented features of ProWeb version 4.31. These will be documented further in a later release of this user guide.

<PROWEB DEBUG>

<proweb debug> indicates where debugging trace is inserted.

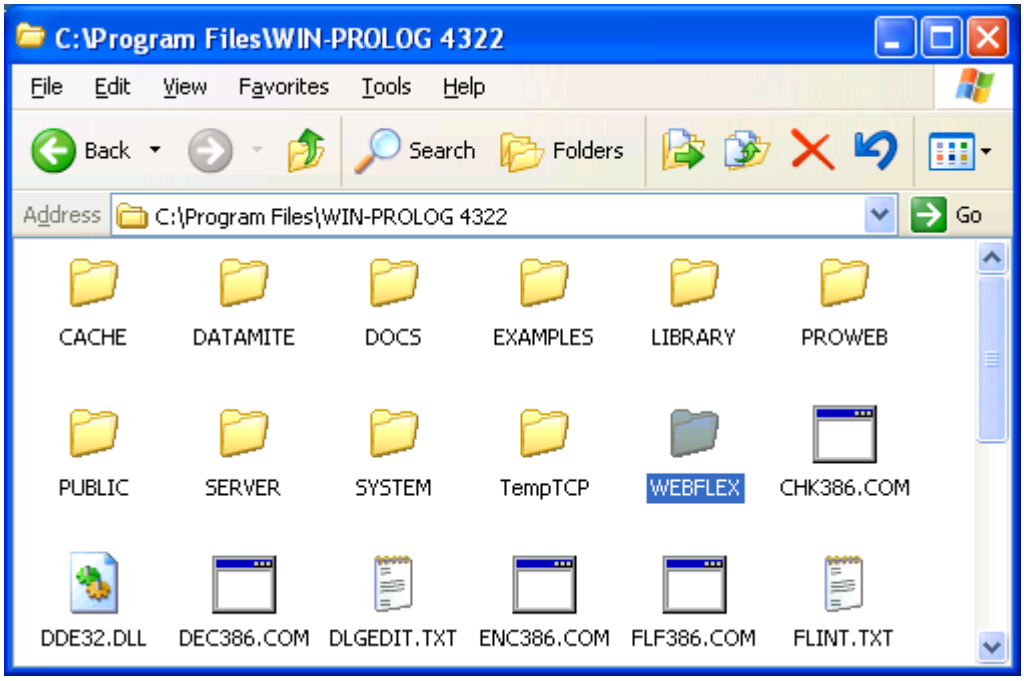
Appendix B – WebFlex

Introduction

The LPA web-based expert system, WebFlex, is basically an instance of a ProWeb application.

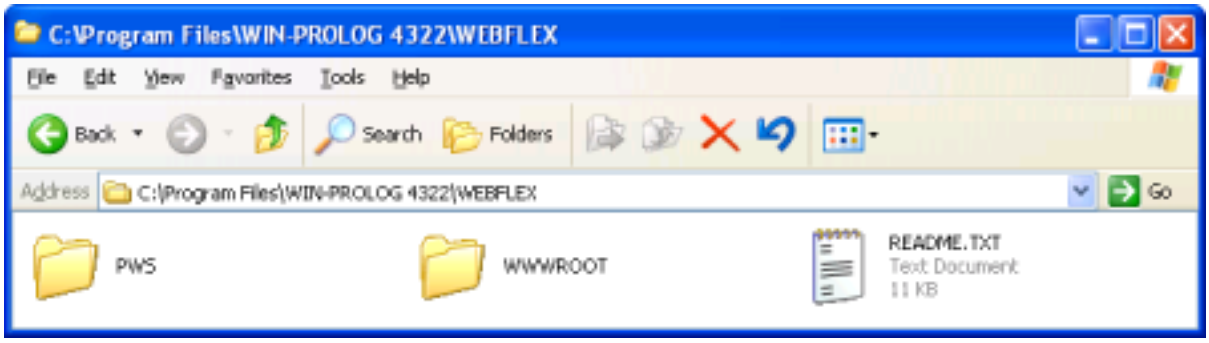
Installing WebFlex

WebFlex has been installed if you have a WebFlex directory within your ***WIN-PROLOG*** directory.



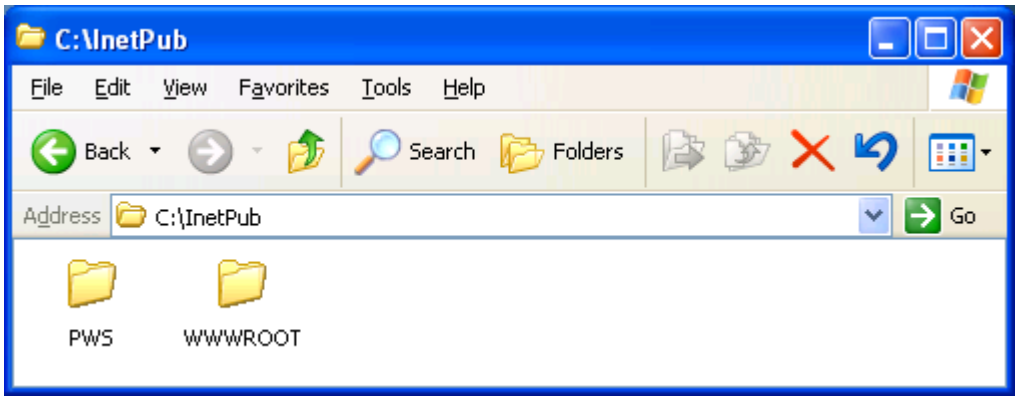
The WebFlex Directory

The WebFlex directory consists of two subdirectories - PWS and WWWROOT.



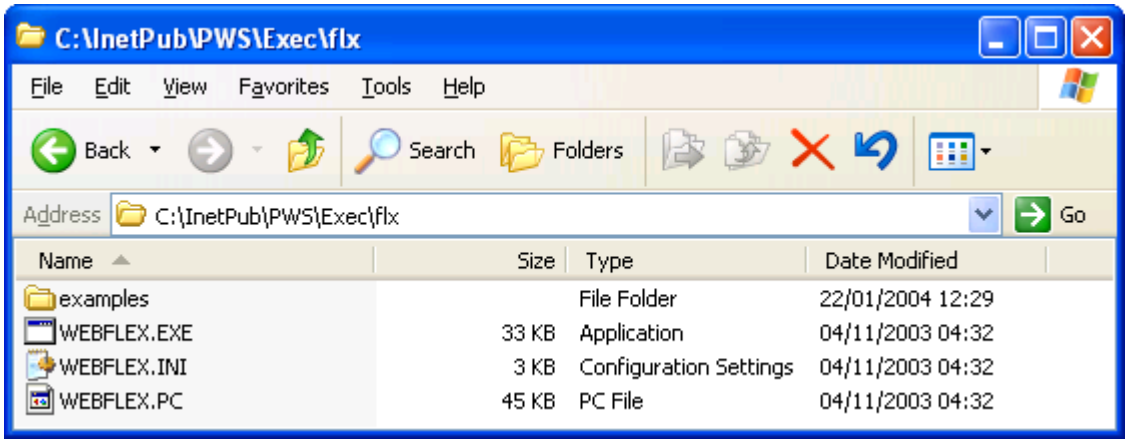
Copying The WebFlex Directory

You need to copy both the PWS and WWWROOT directories to your C:\INETPUB directory. You may already have a WWWROOT directory.

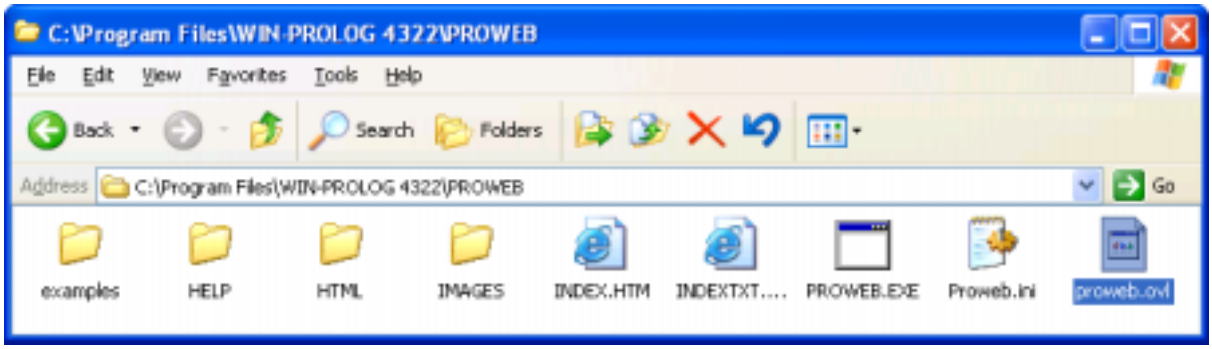


Copying The ProWeb Files

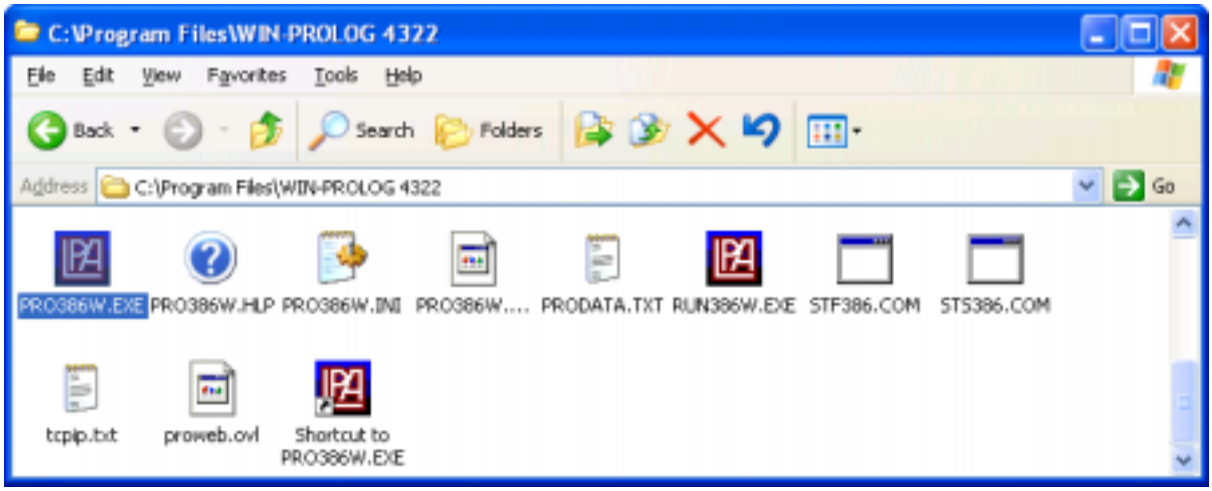
You now need to copy two files into the C:\INETPUB\PWS\EXEC\FLX directory.



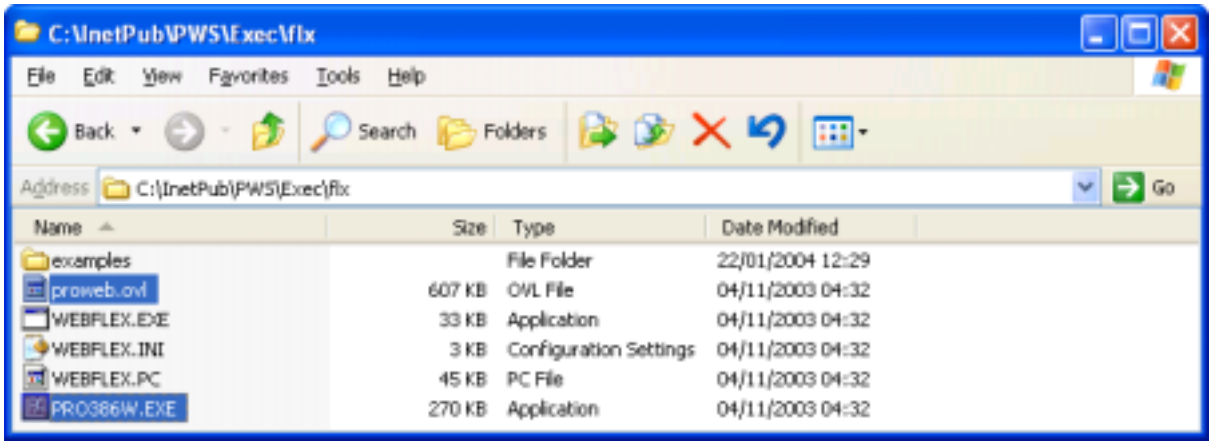
Firstly, you need to copy PROWEB.OVL from your C:\PROGRAM FILES\WIN-PROLOG 4322\PROWEB directory and place it into the C:\INETPUB\PWS\EXEC\FLX directory.



Secondly, you need to copy PRO386W.EXE from your C:\PROGRAM FILES\WIN-PROLOG 4322 directory and place this too into the C:\INETPUB\PWS\EXEC\FLX directory.



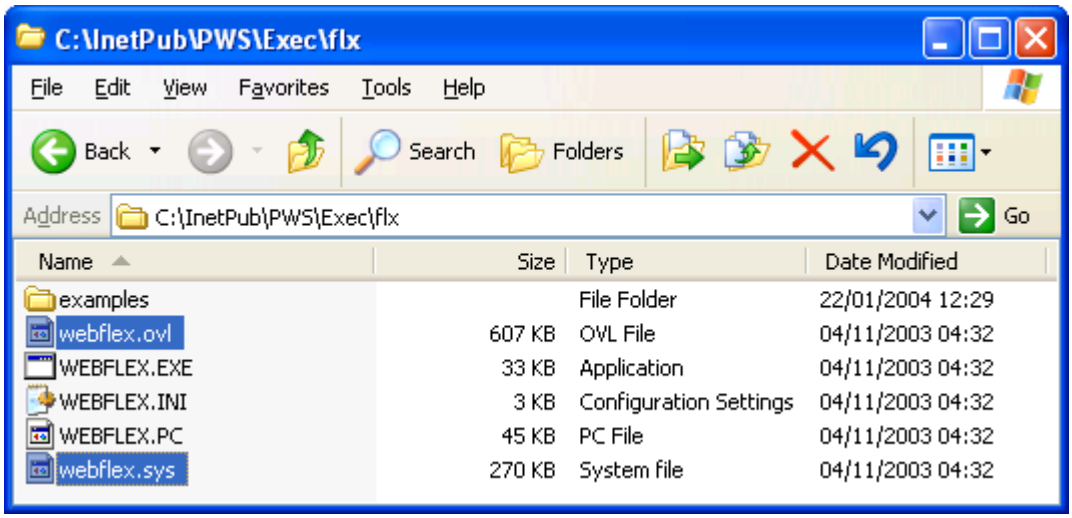
Your C:\INETPUB\PWS\EXEC\FLX directory should now look like this:



Thirdly, you need to rename PRO386W.EXE in the C:\INETPUB\PWS\EXEC\FLX directory to WEBFLEX.SYS.

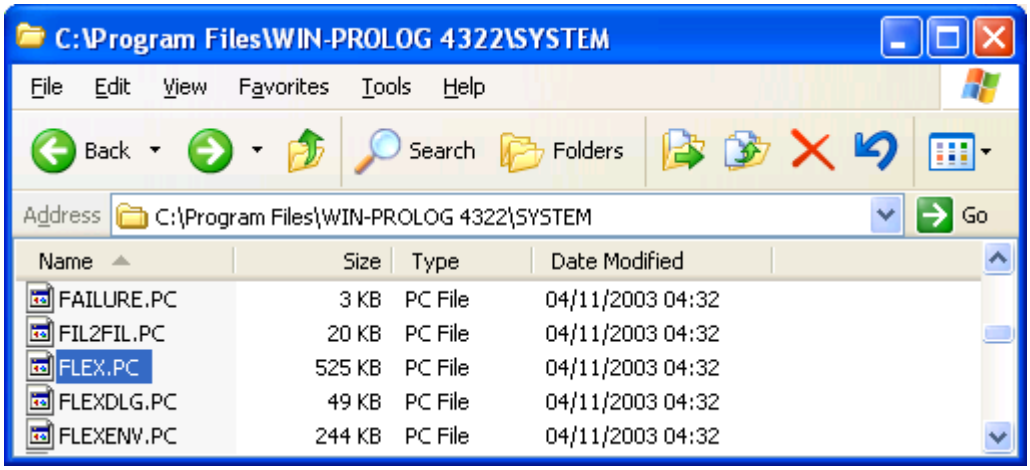
Fourthly, you need to rename PROWEB.OVL in the C:\INETPUB\PWS\EXEC\FLX directory to WEBFLEX.OVL.

Your C:\INETPUB\PWS\EXEC\FLX directory should now look like this:

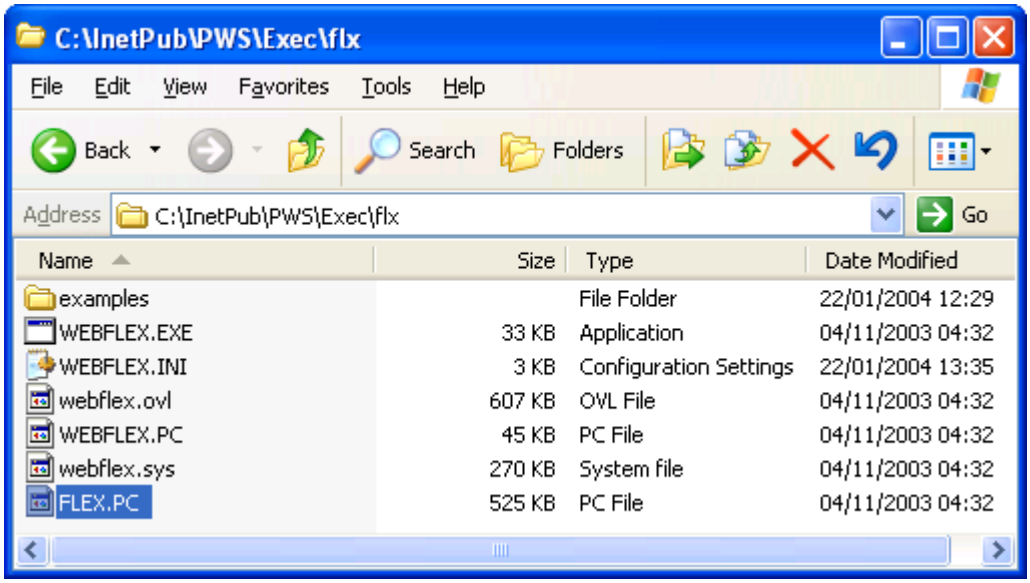


Copying The FLEX.PC File

You now need to copy FLEX.PC from the C:\PROGRAM FILES\WIN-PROLOG 4322\SYSTEM directory and place it into the C:\INETPUB\PWS\EXEC\FLX directory.



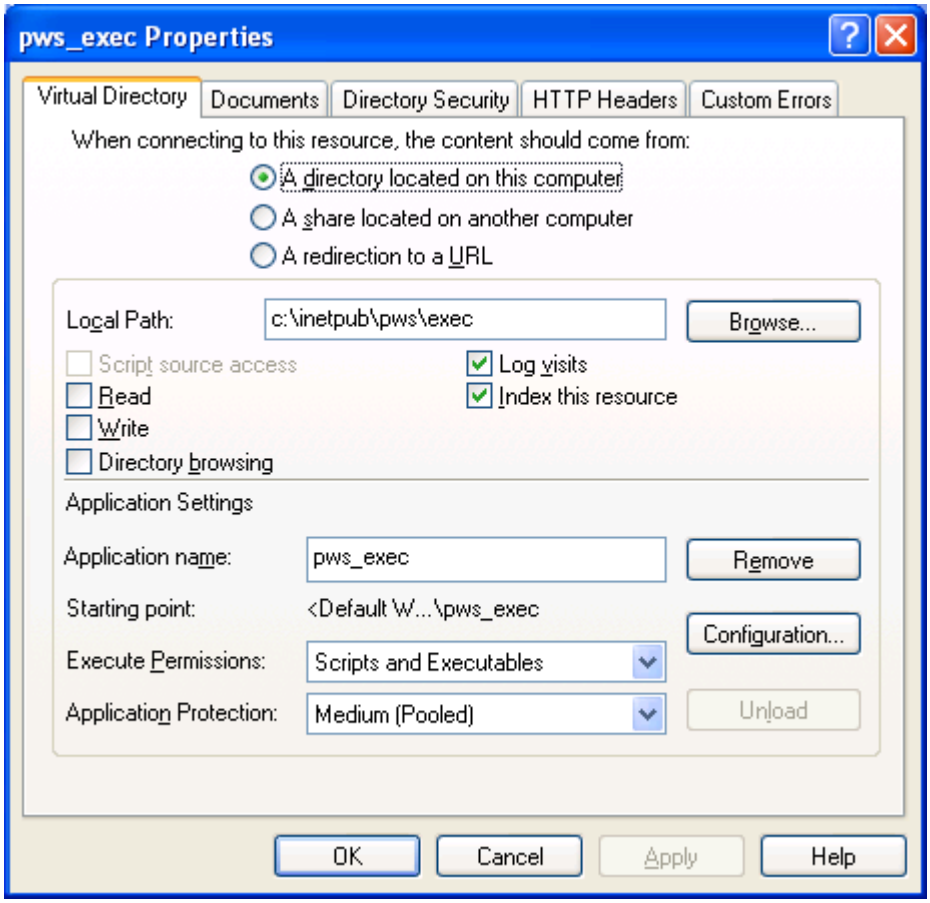
Your C:\INETPUB\PWS\EXEC\FLX directory should now look like this:

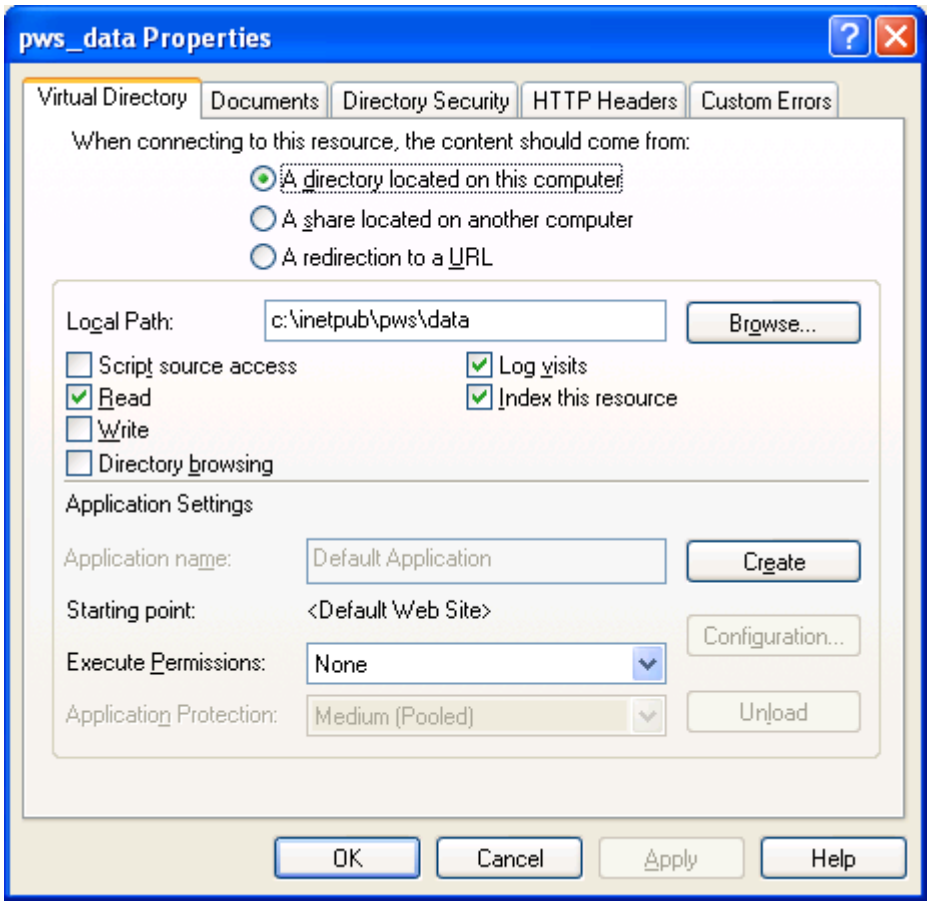


Setting Up The Aliases

The files making up WebFlex are split over three distinct areas, each requiring its own alias to be set up within Personal Web Server or IIS.

Directory	Alias Name	Permissions
C:\INETPUB\WWWROOT	This is IIS's 'home directory'	read
C:\INETPUB\PWS\EXEC	pws_exec	execute
C:\INETPUB\PWS\DATA	pws_data	read


pws_exec Properties
Virtual Directory
Documents
Directory Security
HTTP Headers
Custom Errors
When connecting to this resource, the content should come from:
A directory located on this computer
A share located on another computer
A redirection to a URL
Local Path:
c:\inetpub\pws\exec
Browse...
Script source access
Read
Write
Directory browsing
Log visits
Index this resource
Application Settings
Application name:
pws_exec
Remove
Starting point:
<Default W...\pws_exec
Configuration...
Execute Permissions:
Scripts and Executables
Application Protection:
Medium (Pooled)
Unload
OK
Cancel
Apply
Help

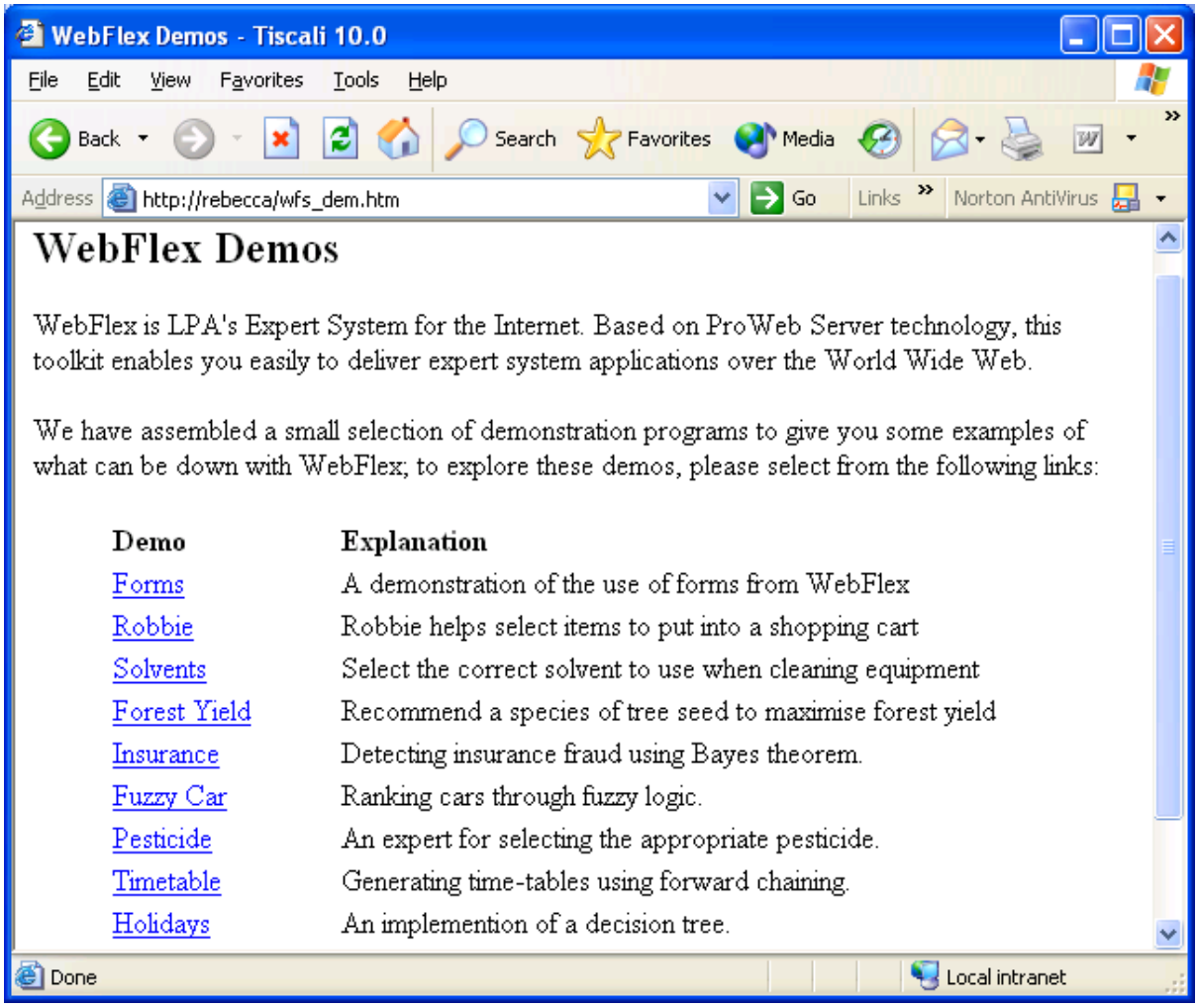


Viewing The WebFlex Launch Pages

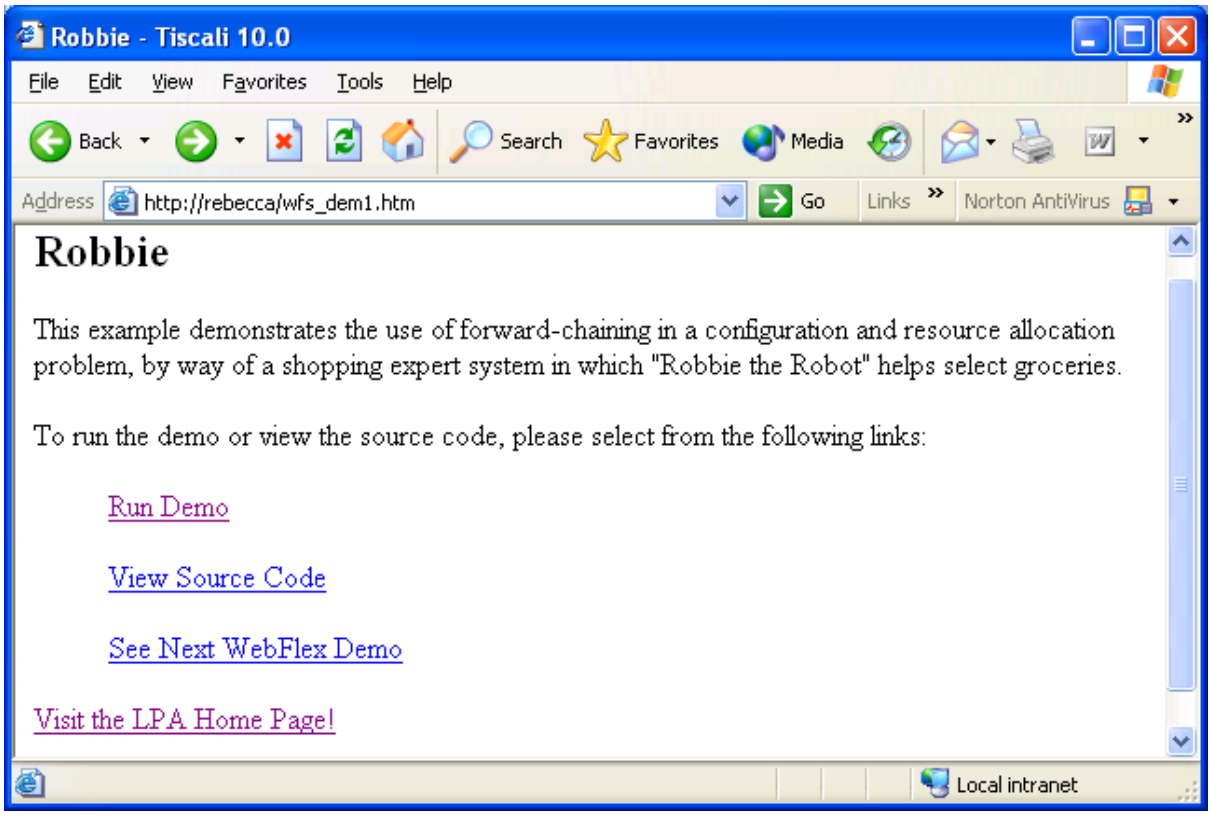
Go into your web browser and execute the following URL:

http://<computer_name>/wfs_dem.htm

replacing '<computer_name>' with the name of your computer.



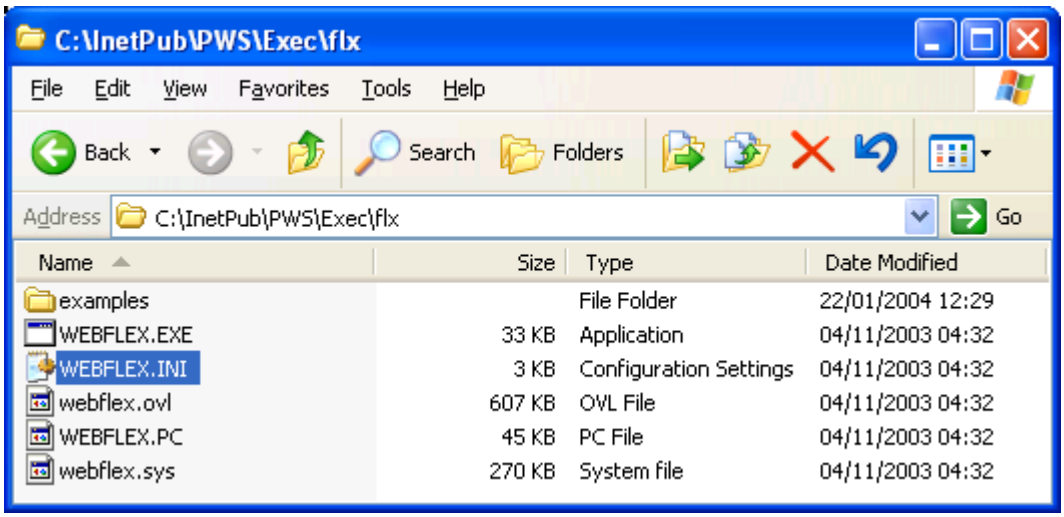
Click on the 'Robbie' hyperlink:



Before we can click on the 'Run Demo' hyperlink, we need to modify WEBFLEX.INI.

Changing WEBFLEX.INI Settings

You now need to modify some of WebFlex's settings within the WEBFLEX.INI file. This file can be found in the C:\INETPUB\PWS\EXEC\FLX directory:



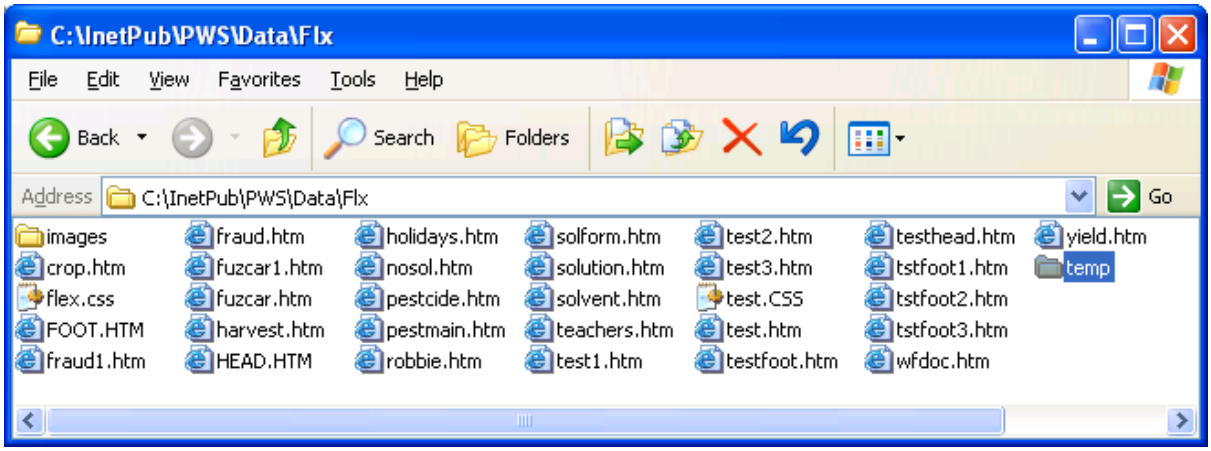
Load WEBFLEX.INI into a text editor (e.g. Notepad) and modify the following settings accordingly:

Original Setting	Change To	Example

TEMP_PATH=c:\inetpub\pws\data\flx\temp\	Leave as supplied but see next section about creating a TEMP directory.	-
BASE_URL=http://mysite/pws_exec/flx/	BASE_URL=http://<computer_name>/pws_exec/flx/	BASE_URL=http://rebecca/pws_exec/flx/
TEMP_URL=http://mysite/pws_data/flx/temp/	TEMP_URL=http://<computer_name>/pws_data/flx/temp/ See next section about creating a TEMP directory.	TEMP_URL=http://rebecca/pws_data/flx/temp/

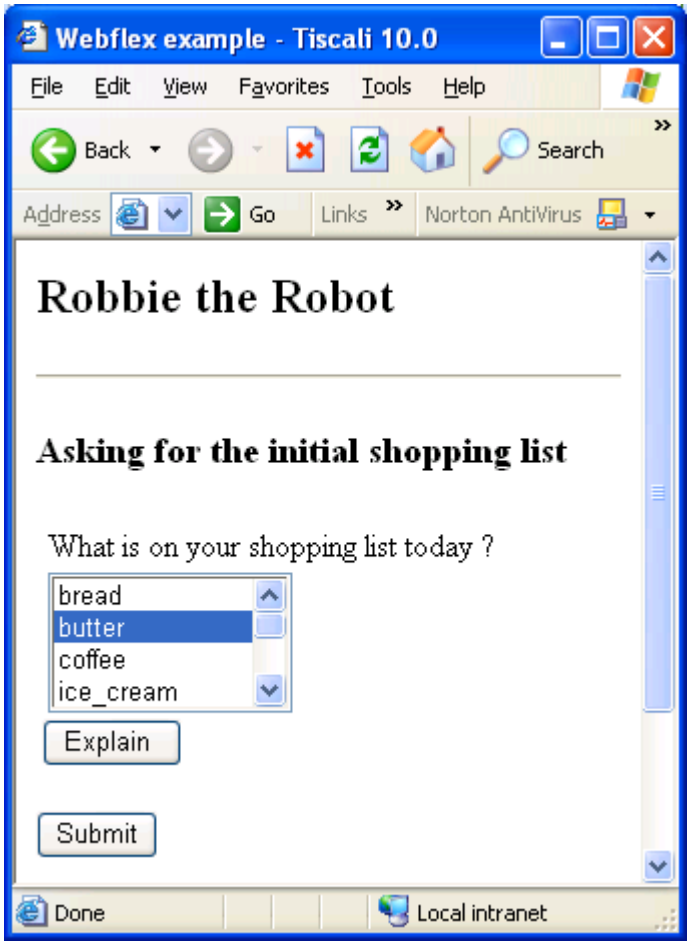
Creating a Temp Directory

Two of the WEBFLEX.INI settings above require a TEMP directory to be present in the C:\INETPUB\PWS\DATA\FLX directory. Create a new directory called TEMP within the C:\INETPUB\PWS\DATA\FLX directory:



Running WebFlex

You are now in a position to return to your web browser window and click on the 'Run Demo' hyperlink and run WebFlex's Robbie The Robot example:



Trouble Shooting

HTML Page Displayed	Solution
ProWeb Server Copyright (c) 1996-2003 Logic Programming Associates Ltd 4.31 - 03 Mar 03 **Main program terminated with an error!** Error #31 File Not Found ensure_loaded / 1	You have forgotten to copy the FLEX.PC file.
System Error (OSE:001:0) Please try again…	You need to create a TEMP directory and point the WEBFLEX.INI settings to it correctly.

Problems

There is a clash of operators between ProWeb and Flex. The nested use of the @ operator in ProWeb (e.g. li @ b @ `this is some text`) can not be used with WebFlex.

Index