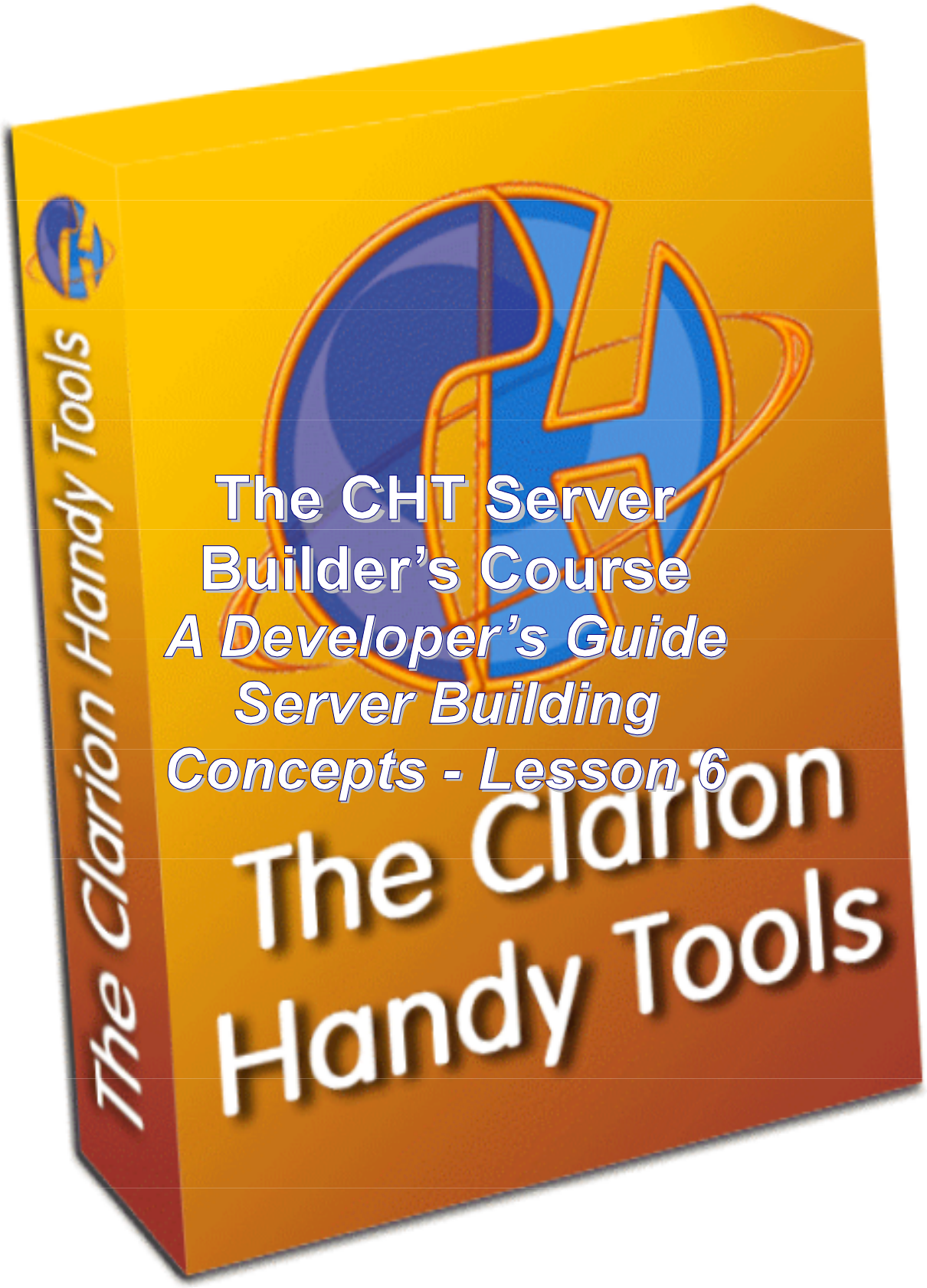

The Clarion Handy Tools
The CHT Server Builder's Course
A Developer's Guide
Server Building
Concepts - Lesson 6
The Clarion Handy Tools

INTRODUCTION

Why Reinvent The Wheel?

This is one of the most frequently asked questions about the Web-Internet-Intranet server development and research work we've been doing. We've replied to this in various ways but here is a list of reasons why you might like to build a CHT server instead of opting for a more traditional solution using IIS (Microsoft Internet Information Server) via your ISP (Internet Service Provider).

- **Convenience**

The convenience of running an office desktop Internet server is unparalleled. With the server local to your office or workspace, making changes to your website or to your website interactive data is a snap compared to the hoops you'll need to go through in order to get your site changes, data changes reflected on a web site based at your ISP. While a simple, static page website is easy enough to maintain using an ISP-based server, have you considered the difficulty of managing a data-interactive website this way? How would you, for instance, create a web site that provides up-to-the-minute order tracking data via the web using an ISP-based server?

- **Control**

CHT server technology is aimed at Clarion Developers. A data-interactive, dynamic-page website, built with CHT server technology totally eliminates "back-end" or "server-side" scripting because your Clarion code, inside the server executable, is your back end script.

Most web servers like IIS are generic servers. They aren't designed to do anything specific such as delivering your up-to-the-minute order processing information from your order processing system to the web. To make IIS perform a specific task, and also to stop it from doing other built-in things that you don't want it to do, accepting rogue scripts from the web, for instance, you must write code. This code is often written in the form of a script using an MS technology called ASP (Active Server Pages). Alternatively, you could opt for any number of other server-side scripting techniques from CGI (an older technology called Common Gateway Interface), to Perl, PHP (Hypertext Preprocessor a lot like ASP), SSI (Server Side Includes), and half a dozen others.

These scripts interface your generic IIS server perhaps to your data base directly, or perhaps to an already-out-of-date extract from your data base that's moved up to your ISP's hardware. They wrap HTML, XML, DHTML, Javascript and CSS around your data, so it will display correctly, and send it off to the visiting web browser. This "wrapper" in itself is another set of scripts that you'll have to write ("front-end" or "client-side" scripts, if you will) which determine the look and feel of your web pages and the placement of data controls. Moving an entry control from one place on your web page to another could involve up to 3 or 4 computer languages to accomplish, depending on the technique selected.

- **Client-Side Scripting Only**

With a CHT server, you only ever write client-side scripts, since your server-side scripting tool is Clarion. Writing the server part of a CHT web application is like writing a Clarion desk-top application. It involves the application of Clarion templates coupled with some Clarion-Language embedding. And the resulting server, just like your desk top application, is specific to your intended end-use. You don't have to worry about this server being waylaid by a rogue back-end script because it ignores such things entirely. In other words, your CHT server is a Clarion application that delivers data to the web - packaged in a way that browsers understand natively, or to a custom web client - again built with Clarion - which your Clarion application understands natively.

- **More Compact Data Requires Less Bandwidth**

Your web page design depends on client-side scripts written in a combination of HTML and Javascript incorporating CSS (Cascading Style Sheets). These scripts are not wrapped around your back end data bloating its size by a factor of 500 percent or more. They're delivered the first time any user visits your site

and are not required to be re-delivered until you make script changes. No big deal if they need to be re-delivered, though, since 3 or 4 script sets of 10-50K with a download time of a few seconds can service quite a complex data interactive website. Data packages are sent from your CHT server in the form of Javascript Data Objects, most, with data validation and entry error checking built right in, eliminating the need for server-side data validation.

- **Security**

There are hundreds of ways to command a generic web server like IIS to do your bidding. That's what it's built to do. And all of the front doors and back doors into such a server are well know and available to anyone with an interest. **CHT servers are single-purpose servers.** The services they provide are decided by the developer based on the specific purpose he/she has in mind. While CHT server classes have the capability to give your server application all kinds of extra powers these powers are turned off by default and cannot be turned on by a script sent up from a remote site. They must be enabled with embed code placed in your server application either by hand embedding or via a CHT extension template.

- **Portability**

To run the CHT Web Group server for example, you need only a DSL or cable internet connection and a computer running the XP or Win2K operating system. No extra web-design tools are necessary, no expensive SQL data engines, no costly web server software and no dual-processor PC's. You don't have to pay for web disk space. The web space available to you is determined by the size of your computer's hard disk. There are literally thousands of end-uses for small to medium web-based functionalities for which many of the available technologies are pure overkill. Rather than a "reinvention" of the wheel, CHT-based servers are a nice alternative for those situations where technical control and cost control are important factors.

YOUR FIRST CHT SERVER (HNDSLFSV.APP)

Our first objective in the server-building part of this course is to have you build a static page web server from scratch. The following discussion revolves around a demonstration application called **HNDSLFSV.APP** available to all CHT subscribers via Live-Update. A pre-compiled, self-installing version of this server is also available from: *http://www.cwhandy.com/pcdemos/hndslfsvdemo.exe*. Before this lesson ends we'll outline how to build a server like HNDSLFSV.APP in a matter of minutes using a CHT Jump Start template. In **Server Building Concepts Lesson 7**, we'll explain how to add and remove server "hooks" or functionalities and trace what happens inside the server from beginning to end of a server request/response cycle.

Before you go any further with this lesson, please make sure that your CHT Clarion code libraries are up-to-date with Live-Update or with WEBLOADER. That will also ensure that you have the latest copy of HNDSLFSV.APP available. You can use C55 or C6.x as either will work fine.

BACKGROUND

A static page web server delivers pre-built web pages from your server computer to visiting browsers. The operative word here is "static". These are web pages that contain relatively fixed content as opposed to user-interactive content.

When someone visits a static page website, they don't have to ask for any specific page or content - although they could. When only your server's IP or URL is entered into a browser command line, the browser receives back a default starting page or "home" page to begin the session. All other pages on the site are linked in various ways to this starting point. CHT's main website is an example of a static page website. Our "base" or "root" page is INDEX.HTML. This is the page you are given when you don't ask to

see a specific page. You may, of course, ask for a specific page, like our purchase page, in the following way: *http://www.cwhandy.com/purchase.htm*.

When a page request comes into a static page server, this can result in one file being sent from the server to your browser, or in several dozen files being sent or even in no files being sent to your browser, depending on how recently you've visited. If you're wondering how one can visit a website and display its pages and have no files sent, review lessons 2 and 3 in the *CHT Script Concepts* part of this course. There we provide a more comprehensive explanation of web browser/web server behaviors as regards the sending of files (or pages) which haven't changed since a previous visit.

How can a single page request result in several files being sent to your browser? This situation arises from embedded extras in web pages, like images and design files. Images embedded in web pages are actually references or links to images sitting on the server. When the HTML interpreter in a visiting web browser encounters a reference to an image, a download request is sent to the server. On arrival of the return image, the browser drops it into a "web cache" and renders it from there, as part of the page. Displaying the CHT home page in your browser requires exactly 7 "hits" or "requests" to the server (at time of writing). These requests are not necessarily sequential, in the case of embedded resources such as images, style files and the like, virtually parallel requests are made. This single, home page request resulted in 7 different socket connections, however brief, between the visiting browser and the server.

Here they are:

```
Request                        Response
GET /index.html                200 OK Content-Length:00558 Time: (12/100 seconds)
GET /$hndmainweb2_content.js   200 OK Content-Length:15064 Time: (44/100 seconds)
GET /cicon13.gif               200 OK Content-Length:01039 Time: (18/100 seconds)
GET /home.htm                  200 OK Content-Length:12546 Time: (41/100 seconds)
GET /hndpages.css              200 OK Content-Length:16653 Time: (47/100 seconds)
GET /htmh2.jpg                 200 OK Content-Length:18703 Time: (56/100 seconds)
GET /chtBox1bfade.jpg          200 OK Content-Length:41326 Time: (76/100 seconds)
```

Strung end to end, the timings provided would make it appear that this web page was entirely processed in just under 3 seconds. Given that some of these timings actually overlap, the web page was probably processed in under two seconds and began to display in the browser in less than one second. The above sequence of events was logged during a first-time visit to the home page.

Here is the log resulting from a second visit to the same home page several minutes later.

```
Request                        Response
GET /index.html                304 Not Modified Time: (9/100 seconds)
GET /$hndmainweb2_content.js   304 Not Modified Time: (9/100 seconds)
GET /cicon13.gif               304 Not Modified Time: (9/100 seconds)
GET /home.htm                  304 Not Modified Time: (9/100 seconds)
GET /hndpages.css              304 Not Modified Time: (9/100 seconds)
GET /htmh2.jpg                 304 Not Modified Time: (9/100 seconds)
GET /chtBox1bfade.jpg          304 Not Modified Time: (9/100 seconds)
```

In this second trial, the server sent no files, only message code **304 Not Modified** indicating to the browser that it should render that particular web page component from its cache. Total turnaround time counting overlap, was very likely under one quarter second. From the provided log values it is also possible to see that this particular server, running on our $700 XP Home computer connected to a 600kbps (upload speed) web connection takes approximately 9/100's of a second from the instant that a socket is opened to receive a request until the socket is closed after having processed the request and having transmitted a minimal **304 Not Modified** response.

Server perceived speed is more than anything, dependent on your web connection's upload speed. Upload speed is king. Since the base turnaround time on this particular setup for a short request and a short

response is 9/100's seconds we can be reasonably certain that in the case of the file named **chtBox1bfade.jpg** with a turnaround time of **76/100**, that **67/100**'s of the time spent was in getting the file uploaded. That's 330,608 bits in just over half a second. Close enough to our connection maximum speed of approximately 600kbps (600,000 bits per second) to lend credence to our assertion. Clearly, the time involved completing any request is heavily dependent on the size of the return data. There's one other element that would not seem immediately relevant but is. That is, the download speeds of the connected browser.

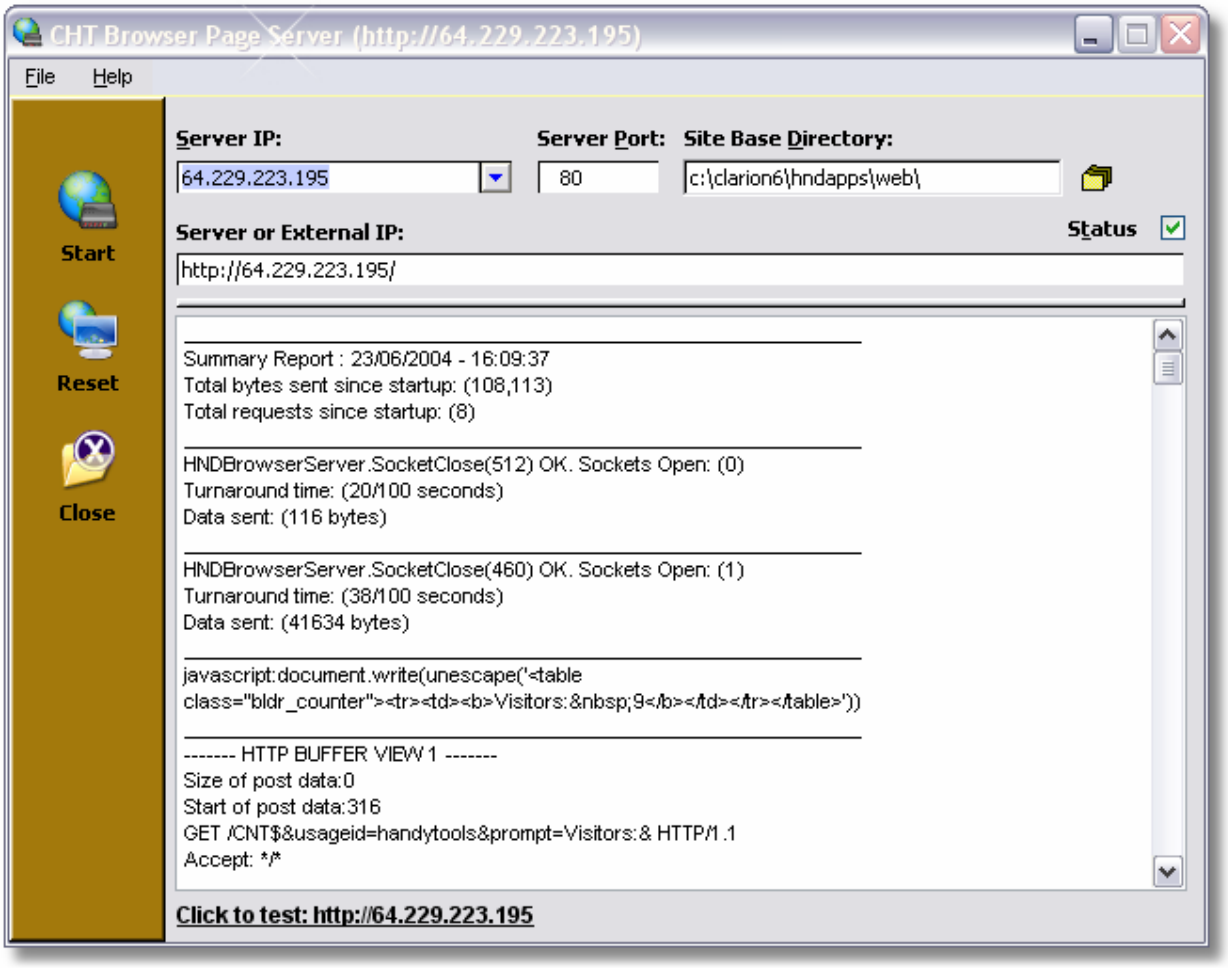


Illustration 6.1

Once the server requests a socket to be closed after it has successfully stuffed the data into the TCP pipeline, it receives an acknowledgement from the browser's TCP/IP components to indicate that the transfer is complete. From a browser using a slow connection, a return acknowledgement takes longer coming back than from a fast connection. The current state of the internet at any time of day has a similar effect on request turnaround. Parts of the internet *do* tend to slow down at certain times of day depending on traffic. To prevent the "transmission" part of a connection interfering or holding up the "receiving" part, these two aspects of a request/response transaction are handled as separate processes by CHT server code.

HNDSLFSV.APP handles request input and request output as two entirely separate operations. The receiving operation must constantly keep an ear to its assigned HTTP port, and when a request comes in, hand it off to another process to decide what is being asked for and to follow up with an appropriate

response. By the time that the output or response process is under way, the listening process is ready and awaiting another request.

If you've watched the HNDSLFSV.EXE server's status control as web requests come in you may already be aware that the turnaround time of most requests is very short. The output, "sender", component that's handed off to by the input, "listener" component spawns a thread, capable of overlapping I/O, to handle background uploading of the requested file to your browser.

Remember too, that this browser-to-server relationship is non-exclusive. While requests from browser "A" are being processed, requests from browsers "B", "C" and "D" may also be under way. The relationship never develops beyond request/response or input/output, with the input socket acting as the output vector back to the browser client. The server never opens a socket connection of its own accord. It does so at the request of a visiting browser.

A visiting browser is allowed to make only one request per open TCP/IP socket. But it can cascade several separate sub-requests - via separate socket connections - to the same server port to complete a transaction such as displaying the CHT home page. This behavior ensures that browser request processing is not happening end-to-end but in a series of quasi-parallel operations.

All sockets have separate "input/output" switches, or vectors, that are controlled independently and only one direction of data travel is allowed at any given time on a given socket. Once the request part of an input/output transaction is complete, that socket is closed to further browser input. That same socket is then used to carry the server's response back to the connected browser. Requests come in on a given socket; responses go out via the same socket.

Output processing is generally, in the form of a file or a notification header indicating that the requested file hasn't changed. Once the return information has been fully transmitted back to the browser, the server closes that TCP/IP socket completely and returns it to the pool of available sockets for re-use.

HTTP HEADERS

All HTTP transactions, be they requests or responses are prefixed with an HTTP header. Some HTTP communications, such as the server's **304 Not Modified** response is only a header without any other data attached. Here is an example of such a return header:

```
1.      HTTP/1.1  304 Not Modified
2.      Last-Modified: Wed, 21 JAN 2004  7:56:00 GMT
3.      Date: Wed, 11 FEB 2004 20:14:19 GMT
4.      Host: http://65.95.88.74
5.      Filename:$HNDMAINWEB2_CONTENT.JS
6.      Socket-Number: 1696
7.      Connection: Close
8.      Content-Type: text/javascript
9.      Content-Length: 0
10.     Server: THE CLARION HANDY TOOLS O8A1.4
```

The important elements in this particular communication are the 1st line, the 5th line and the 9th line which indicate respectively that the file named in line 5 was not changed, and that there is zero data attached to this header. The request that triggered this response was as follows:

```
1.      GET /$hndmainweb2_content.js HTTP/1.1
2.      Accept: */*
3.      Referer: http://svr.cwhandy.ca
4.      Accept-Language: en-ca
5.      Accept-Encoding: gzip, deflate
```

```
6.      If-Modified-Since: Wed, 21 Jan 2004 02:56:00 GMT
7.      User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; .NET CLR 1.1.4322)
8.      Host: svr.cwhandy.ca
9.      Connection: Keep-Alive
```

The important elements in this request are the 1st line, the 6th line and the 7th line. Line 7 describes the type of browser attached. Line 6 indicates that the server should only send the file if it is newer than the date indicated, and line 1 names the requested file.

An incoming request is seen by the server as a stream of bytes. The first part of this data stream is the header which consists of an opening line, prefixed by GET or POST and the name of something being requested or a command intended for the server.

```
GET /$hndmainweb2_content.js HTTP/1.1 (CRLF)
```

Following that are header tags, each terminated by a colon, followed by some information followed by a carriage return line feed.

```
If-Modified-Since: Wed, 21 Jan 2004 02:56:00 GMT (CRLF)
```

When the server makes any kind of response, be it a no-data response, a file, or database data, the information is packaged again as a stream of bytes, prefixed by an HTTP header appropriate for the situation. The breadth of responses appropriate for a browser-server connection is pretty narrow. Here is the entire list used by CHT servers at this point in time.

```
HTPROP:Success                  EQUATE(' 200 OK ')
HTPROP:NewResourceCreated       EQUATE(' 201 New Resource Created')
HTPROP:RequestAcceptedWait      EQUATE(' 202 Request Accepted Please Wait')
HTPROP:NoContent                EQUATE(' 204 No Content ')
HTPROP:ResourceNotModified      EQUATE(' 304 Not Modified')
HTPROP:NotAuthorized            EQUATE(' 401 Not Authorized')
HTPROP:NoExecutablesAccepted    EQUATE(' 403 No Executables Accepted')
HTPROP:AccessDenied             EQUATE(' 403 Access Denied')
HTPROP:FileTooBig               EQUATE(' 403 File Too Big')
HTPROP:ResourceNotFound         EQUATE(' 404 Not Found ')
HTPROP:ServerErrorInternal      EQUATE(' 500 Server Internal Error')
HTPROP:ServerBusy               EQUATE(' 503 Server Busy')
```

Some of these are specific to CHT but the numbering scheme is determined by standard HTTP protocol. All 200 responses are a variation of OK. All 300 responses are branching or conditional responses that cause the browser to behave differently depending on which 300 value. All 400 responses are denial of service related to security issues or simply the non-existence of a requested resource. Everyone who has ever used a browser is by now familiar with **404 Not Found**. All 500 errors are server-related from the standpoint of the server communicating back its inability to fulfill a request for one reason or another. The text portion of these return values are for human consumption. The browser reacts only to the numeric portion.

When the server sends back some actual data the header indicates this using a Content-Length tag:

```
1.  HTTP/1.1 200 OK   (CRLF)
2.  Content-Length: 15064  (CRLF)
3.  Content-Type: text/javascript (CRLF)
4.  Last-Modified: Wed, 21 JAN 2004  2:56:00 GMT (CRLF)
5.  Date: Thu, 12 FEB 2004 15:52:56 GMT (CRLF)
6.  Content-File: $HNDMAINWEB2_CONTENT.JS (CRLF)
7.  Host: http://65.95.88.193 (CRLF)
8.  Socket-Number: 308 (CRLF)
```

```
9.  Connection: Close (CRLF)
10. Server: THE CLARION HANDY TOOLS-O8A1.4 (CRLF)(CRLF)
Data starts here...
```

In the stream of bytes returned to the browser from the server, the HTTP header begins at byte one and continues until a double carriage-return-line-feed combination is encountered. HTTP headers are not fixed length. That would limit their flexibility to communicate a variety of information. The actual data being returned begins after CRLF/CRLF and continues for the number of bytes indicated by **Content-Length:.** The nature of the data is provided by the **Content-Type:** label, in this case, text/javascript. You can also see here that the server was given a **Last-Modified:** date for this file and that the server responded by sending a new copy of the file indicating to the browser that it had found a newer version of the file and giving the file **Date:.**

All HTTP header date transactions are given in GMT (Greenwich Mean Time) so the server must be aware of it's time offset relative to Greenwich England. In earlier servers, we provided a manual server configuration setting for the time offset. Our server code now uses the CHT class called **HNDDates** (HNDDATES.INC/CLW) to read the time offset from your operating system. Be aware that when your computer date/time clock setting is incorrect, it may cause the server to send files to visiting browsers when it really isn't necessary. Or it may indicate to the visiting browser that a file hasn't changed, when in fact, it has, because it's doing the date/time offset math incorrectly based on incorrect, local date/time information as provided by your computer clock.

Return headers are built for you automatically by CHT server code. We discuss them here as a prelude to deeper understanding of what HTTP actually is. When communicating with a browser, the types of return headers used are determined by the general specifications of internet browsers. Browsers issue and expect certain HTTP header tags.

When the transaction is taking place between two CHT applications, a live-update client and a live-update server, for instance, the range of HTTP header information used is significantly broader. Since in that case, we're coding both ends of the transaction, it's possible to include header information of our own design. Suffice to say that in a Browser-Server transaction, CHT servers must play by the same rules already outlined in the specifications describing the behavior of HTTP servers and browsers in general.
For an official list of the most commonly used HTTP headers, consult this website and any other links it may provide: http://www.cs.tut.fi/~jkorpela/http.html.

HTTP headers are what make the HTTP protocol different from FTP, SMTP, and NNTP and so on. Down deep, all of these communications protocols are TCP/IP network communications using sockets.

SERVER INTERFACE

Server IP

A basic static-page HTTP server requires very little by way of an interface since most of what it does is automatic and performed in the background. Four bits of information are required to get the server operational. These are the input controls provided on the server interface.

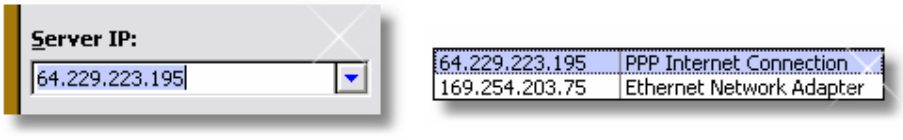


Illustration 6.2

The Server IP control is used to determine which internet IP you will be servicing. Server code searches your computer for available IP's and presents you with a list that includes a generalized description of the type of IP's found.

Obviously, this requires some knowledge on your part in deciding which IP is appropriate. The HNDIPADR.INC/CLW module (**HNDIPAddress** class) handles this part of the server's operation. The base server module HNDBRWSV.INC/CLW (**HNDBrowserServer** class) creates an instance of the HNDIPAddress for its own use which it calls **IP**.

In the demonstration server application HNDSLFSV.APP, early on in the template generated server procedure **JumpStartStaticPageServer()**, a call is made to a HNDIPAddress function to determine a list of available IP's.

```
1. ServerIP                  = Server.IP.GetAdapterIps(IPQ)
2. ?ServerIP{PROP:From}      = IPQ
3. IPItem                    = GETINI('CONFIG','IPItem', 1,INIMgr.FileName)
4. GET(IPQ,IpItem)
5. ?ServerIP{PROP:Selected}  = IpItem
6. ServerIP                  = IPQ.IPAddress
```

Line 1 calls the function **.IP.GetAdapterIPs()** passing it a queue called IPQ which is a queue of type HNDIPQ2 defined in HNDEQU.CLW. This call returns a list of available IP's stored in queue IPQ.

Line 2 assigns the queue to the FROM property of the drop down control ?ServerIP, making the queue visible on the server window.

Line 3 fetches the queue pointer number of the last IP that you selected in the IP drop down. If no entry is found, it defaults to item 1.

Line 4 fetches the queue item selected in a previous server session, or 1, depending on circumstances.

Line 5 sets that item as the default item selected in the drop down control.

Line 6 sets the contents of local variable ServerIP to the IPQ.Address value of the selected IP address.

All this happens before you actually select an IP address and before you start the server, on the assumption that the IP you selected last time is the one you will use again, thus saving the step of having to select the correct IP each time the server is run.

Server Port

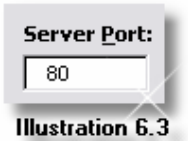


Illustration 6.3

TCP/IP allows for 64K ports on any IP address. The fact that most HTTP servers run on port 80 is purely convention. The same convention that assigns FTP transactions to port 21, SMTP transactions to port 25, POP3 transactions to port 110 and HTTPS transactions to port 443. These are merely conventions like driving on the right or left side of the road. HTTP transactions can take place on any port as long as both ends of the transaction know which one to use. As with the side-of-the-road convention, things go smoothly as long as everyone uses the same set of assumptions.

Only one server can service a given IP/PORT combination. But there's nothing stopping you from running several servers on the same IP each servicing a different port number. In fact, at CHT we have one server computer running our website server on port 80, an alternate website server on port 9000, a Web Group server on port 8000, an alternate Web Group server on port 443, and three Live-Update servers on ports 9001, 9002, and 9003.

Visiting our Web Group server you may be accustomed to entering **http://news.cwhandy.ca**. In fact that URL resolves to **http://www.cwhandy.ca:8000,** which is the same IP address as our website server, but uses port 8000. We're simply using DNS techniques to route a given URL to a specific IP/PORT on our server hardware without the browser user having to remember the IP or the port number in use.

External IP

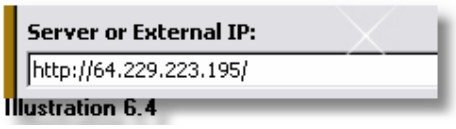


Illustration 6.4

The purpose of this control is easily enough understood if you already understand what routers do. This setting lets you run the CHT server on an IP assigned to your computer by the router while the router is servicing your ISP-assigned internet IP address. Since the router can have several computers attached, it can monitor internet requests and among other things, "route" specific port requests to specific computers.

The following diagram, illustrates two different setups. In the setup on the left, the server computer is hooked directly to the internet or to a hub connected directly to the internet via DSL modem.

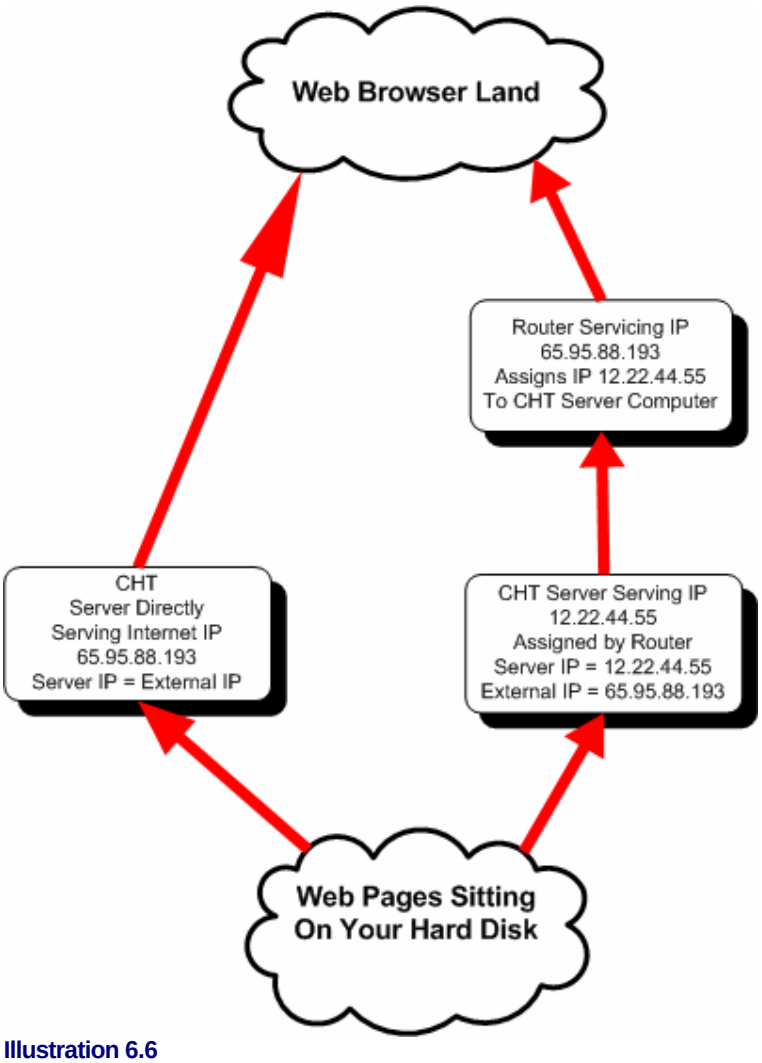


Illustration 6.6

The ISP-assigned internet address is visible on the computer and the CHT server can detect it. In this case the **Server IP = External IP**.

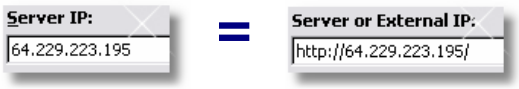


Illustration 6.5

In the setup on the right, the server computer is hooked to the internet via router. The router obtains a connection to the ISP-assigned internet address. It then assigns an IP address to the computer which is different than the actual internet IP. In fact the IP it assigns is not visible on the internet at all. In this case, the CHT server talks to the router, the router talks to the internet. When requests come in, the requesting browser talks to the router, the router talks to the CHT server.

In this case **Server IP <> External IP**. The chief purpose of this setting is to tell the CHT server what IP is required to reach it from the internet so that it can correctly complete HTTP header references and certain other external references which, because they do not pertain to a static page server, we will be discussed in greater detail in a later lesson.

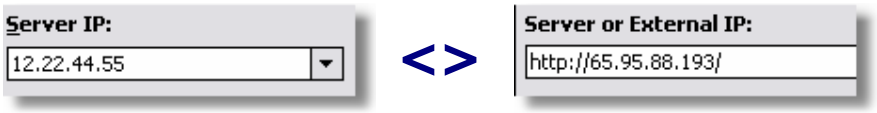


Illustration 6.7

Site Base Directory

Illustration 6.8

The purpose of this control is more immediately obvious than the last one. It tells the server where your pages are located, or at least where your "base" or "root" page, **index.html** is located. For obvious security reasons, as far as CHT servers are concerned, no other directories except this "Site Base Directory" and directories below it exist on your computer. In the case of the demonstration site that comes with HNDSLFSV.APP - available from *Help -> Download Demo Site* - all files including images are located for simplicity and portability in the site base directory. This need not be the case. Images, for instance could be stored in a subdirectory of your site base directory. In that case when an image is referenced in a link, it must be referenced, with the subdirectory included in the image name. Here are several examples:

```
<--- image located in site base directory --->
<IMG SRC="htmh2.jpg" WIDTH="661" HEIGHT="98" BORDER="0">

<--- image located below site base directory in /images/ --->
<IMG SRC="/images/htmh2.jpg" WIDTH="661" HEIGHT="98" BORDER="0">
```

The same principle applies to resource files such as style sheet files and javascript procedure files. Here are several examples:

```
<--- style sheet file located in site base directory --->
<LINK REL="STYLESHEET" HREF="hndpages.css" TYPE="text/css">

<--- style sheet file located below site base directory in /styles/ --->
<LINK REL="STYLESHEET" HREF="/styles/hndpages.css" TYPE="text/css">

<--- a menu which locates other pages in the site base directory --->
<SPAN class="mainmenu">
||<a href="home.htm">Home</a>
||<a href="presentcomponents.htm">About</a>
||<a href="aboutus.htm">Author</a>
||<a href="goodwords.htm">Comments</a>
||<a href="whatsnew.htm">What's New</a>
||<a href="helpfiles.htm">Downloads</a>
||<a href="purchase.htm">Subscribe</a>
||<a href="http://news.cwhandy.ca/" target="_blank">Support</a>
</SPAN>
```

```
<--- a menu which locates other pages below the site base directory /pages/--->
<SPAN class="mainmenu">
||<a href="/pages/home.htm">Home</a>
||<a href="/pages/presentcomponents.htm">About</a>
||<a href="/pages/aboutus.htm">Author</a>
||<a href="/pages/goodwords.htm">Comments</a>
||<a href="/pages/whatsnew.htm">What's New</a>
||<a href="/pages/helpfiles.htm">Downloads</a>
||<a href="/pages/purchase.htm">Subscribe</a>
||<a href="http://news.cwhandy.ca/" target="_blank">Support</a>
</SPAN>
```

Depending on how fussy you are about the location of things, you might conceivable have only **index.html** in the site base directory and all other files neatly tucked into subdirectories. The server doesn't care as long as you remember to prefix such "tucked away" files with the appropriate sub-directory using forward slashes when you do.

If your HTML is on the ball, you will also have noticed that *links or references to files and pages do not need to be on your server at all* . The last menu item called, **Support** in the example menu above, references the home page on the CHT Web Group server. It may not have occurred to you, but this kind of thing happens all the time. A website with a single, home entry point could conceivable bounce the visiting browser between several single-purpose web servers. Whether you do this, or not, is a design consideration.

SERVER SERVICES

We've said this before, in earlier essays, but it bears repeating. **CHT servers are primarily designed to be single-purpose servers**, not generic, general purpose servers. CHT technology is aimed at Clarion developers of whom, we assume a certain level of technical competence with the Clarion application development system. Just as the desktop applications you build are more-or-less single-purpose applications, CHT servers are built as needed, with a specific end use in mind.

To that end we'll discuss here a number of services that HNDSLFSV.APP provides given that it is designated a "static page server" as well as a number of other services which it could provide but which are disabled or not fully realized in this application for the reason that they are not needed for this particular end-use designation.

File Services

It should by now be fairly obvious that a static page server is in fact a specialized file server. It communicates with a browser to send it files on request and it intelligently and conditionally processes the files in such a way as to optimize the amount of uploading required. This particular design permits the following file extensions to be requested:

```
CASE CLIP(UPPER(Exten))
OF '.JPG' OROF '.JPEG' OROF '.GIF' OROF '.PNG' OROF  '.BMP' OROF '.ICO'
OF '.MPEG' OROF '.MPG' OROF '.MP3' OROF '.WAV' OROF '.AU' OROF '.SND' OROF '.SWF'
OF '.HTM' OROF '.HTML' OROF '.XML'
OF '.JS'
OF '.CSS'
OF '.TXT'
OF '.DOC' OROF '.PDF' OROF '.XLS'
OF '.ZIP'
OF '.HNZ' OROF '.HNY' OROF '.HZY'
!DEVELOPER NOTE ----------------------------------------------------------------
!Requesting all of the file types below and all types covered by the ELSE clause
!will result in a not authorized header being sent back.
!If you really need these things returned in an un-authorized fashion, override
!this function inside your server app and allow them through.
```

This code extract is from HNDSUBSV.CLW in a function called: HNDSubscriptionServer.CheckSendFile()

```
!Otherwise, look at the secure file upload/download methods for getting such files
!to and from the server.
!-----------------------------------------------------------------------------
OF '.EXE' OROF '.DLL'
OF '.INI' OROF '.TPS'
ELSE
  !The command line requested something not approvied for send.
  SELF.SendNoContentHeader(SELF.CmdSocket, SELF.SocketIndex, HTPROP:NotAuthorized)
END
```

If you have file delivery requirement beyonds this, open the **HNDSubscriptionServer** class module called HNDSUBSV.CLW and check out a method called **CheckSendFile()**. This is a virtual method. That means you can expand, or completely change the behavior of this method by embedding your own code into it in order to embellish or replace CHT code, inside the server procedure **JumpStartStaticPageServer().**

If you don't know how to do that, please take the time to further your understanding of Clarion OOP and embedding. The concept of embedding to override a virtual method is fundamental to all serious Clarion application development. It is certainly not specific to this situation. Nor is it a CHT-specific concept.

```
Server.CheckSendFile PROCEDURE

! Start of "The Clarion Handy Tools - Handy Method Data Section"
! [Priority 5000]

! End of "The Clarion Handy Tools - Handy Method Data Section"

  CODE
  ! Start of "The Clarion Handy Tools - Handy Method Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  PARENT.CheckSendFile
  ! [Priority 7500]

  ! End of "The Clarion Handy Tools - Handy Method Executable Code Section"
```

Illustration 6.9

Conceptually, all virtual methods are overridden by the ABC template system when you enter the Clarion Embeditor as we have done for the screen capture pictured above. If you embed nothing, the method is not overridden and the standard **CheckSendFile()** code provided in the CHT library file HNDSUBSV.CLW is executed.

The code in a virtual method provided in an OOP library file is always represented in any embed point as **PARENT.SomethingOrOther()**. In this case, that's **PARENT.CheckSendFile()**. If you want to execute some code BEFORE CHT code executes, place it in the embed point before PARENT.CheckSendFile(). If you want to execute some code AFTER CHT code executes, place it after PARENT.CheckSendFile(). If you want to disable CHT code altogether, place a RETURN before PARENT.CheckSendFile(), in which case this server won't do anything unless you've coded it yourself.

Page Counter

Every website deserves a page counter, and this server provides one. To see how it's done, open first a page in the demo website provided with this server called, HOME.HTM. The easiest way to get to any of the demo site pages is to simply start the server, execute the "Click to Test" link. Once the "Home" page appears, right click and select "View Source". Your browser will show you the "source code" for the page that you're viewing. Near the bottom of this file you will find a script that looks like this:

```
<script src="CNT$&usageid=handytools&prompt=Visitors:&" >
</script>
```

When your browser encounters this link it tries to resolve it using the information provided. In this case, instead of resolving to a file name we have added a server command (CNT$ or REQUEST:UsageCounter) to the URL, as well as a couple of parameters with values. The first parameter is "usageid" and its value is "handytools". The second parameter is "prompt" and its value is "Visitors".

The server returns a page count specific to the "handytools" account. If that account doesn't exist it creates a new one starting at 1, if it does exist it ups the current count value by one and returns that value to be displayed on your web page.

We suggest on your server setup that you change the usageid tag to something more appropriate for your website. Remember, though, that once a counter is started under a specific usageid, if you change the usageid, a new account is started with that name which initializes at 1.

To see where server code is intercepting this server command open HNDSUBSV.CLW and search for **REQUEST:UsageCounter**. When you do that you'll find the following:

```
     OF REQUEST:UsageCounter
        SELF.TakeUsageCounterRequest()
        RETURN
```

In other words, when the server receives a CNT$ command it calls a server function called **TakeUsageCounterRequest()**.

A quick search through HNDSUBSV.CLW, leads us to the source code for this procedure.

```
HNDSubscriptionServer.TakeUsageCounterRequest  PROCEDURE ()
!This increments the counter for "CounterID"
!and sends back an HTML string with a hit count.

  CODE
  CounterID = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterID)
  IF NOT CounterID THEN
     SELF.SendResponseHeader(SELF.CmdSocket,  |
     SELF.SocketIndex, ERRORCODE:PostedFileNoIDProvided)
     RETURN False
  END
  CounterSheet = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterSheet)
  CounterText  = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterText)
  IF NOT CounterSheet THEN CounterSheet = 'bldr_counter'.
  IF NOT CounterText  THEN CounterText  = 'Visitors:'.

  NextNumber = GETINI('COUNTERS',CounterID,1,SELF.GetServerLogName())
  PUTINI('COUNTERS',CounterID, NextNumber + 1, SELF.GetServerLogName())
  SendNumber = ScriptStart & CLIP(LEFT(FORMAT(NextNumber, @N10))) & ScriptEnd

  SELF.Disk.SearchReplaceInCString(SendNumber,'$$$$$$$$$$',CLIP(CounterSheet))
  SELF.Disk.SearchReplaceInCString(SendNumber,'!!!!!!!!!!',CLIP(CounterText))

  SELF.TransmitBuffer(SELF.CmdSocket, SendNumber, SELF.SocketIndex, LEN(SendNumber))
  SELF.PostStatusQ(SendNumber)

  RETURN True
```

There are numerous "hook" functions like this in the server, each with a different, intended purpose. In coming lessons we'll point most of important ones out to you with an explanation of how they work and what they're intended to do. At this point we draw your attention to a function being used here called **SELF.ExtactCommandLine()**. Its purpose is to extract from the HTTP header or from the GET/POST line preceding the header, the information attached to a parameter tag.

Take for example, the page counter command illustrated above. It appears to the server as follows:

```
GET /CNT$&usageid=handytools&prompt=Visitors:& HTTP/1.1
Accept: */*
Referer: http://65.95.89.57/home.htm
Accept-Language: en-ca
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: 65.95.89.57
Connection: Keep-Alive
```

The following line of code extracts the value attached to the **usageid** tag sent with the CNT$ command:

```
  CounterID = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterID)
```

You guessed it, TOKEN:HttpUsageCounterID has a defined value in HNDEQUSK.CLW as **usageid**. The value returned from this particular call given the command line under discussion, is **handytools**. There are two other calls to **ExtractCommandLineItem()**.

```
  CounterSheet = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterSheet)
  CounterText  = SELF.ExtractCommandLineItem(TOKEN:HttpUsageCounterText)
```

The first of these extracts the name of a style sheet to be used to display the counter number. Our CNT$ illustration provides no style sheet so the default name, "bldr_counter" is inserted by the server. The second call extracts the prompt (TOKEN:HttpUsageCounterText) that precedes the usage counter numeric value. Our CNT$ illustration provides the word "Visitors".

Ultimately, returned to the web page from this is a section of javascript code that writes HTML to the browser, as follows:

```
javascript:document.write(unescape('<table
  class="bldr_counter"><tr><td><b>Visitors: 1</b></td></tr></table>'))
```

This is rendered by the browser as a section of HTML displaying in bold, the word Visitors: followed by a number all displayed inside a table which follows the style dictated by a style sheet called "bldr_counter". The style sheet named is assumed to be somewhere in your style sheets (.CSS) file for bldr_counter to have any visible effect.

```
<table class="bldr_counter">
 <tr>
  <td>
   <b>Visitors: 1</b>
  </td>
 </tr>
</table>
```

The page counter function, as we've said, is a built-in server function which you can request the server to execute by sending the command CNT$ (REQUEST:UsageCounter) plus two or three relevant parameters. This particular function has not been disabled in HNDSLFSV.APP because it's relevant to the end-use

intended. A list of other built-in commands, which we call REQUESTS can be found in the CHT equates file called HNDEQUSK.CLW. Here is a complete listing (at time of writing):

```
REQUEST:HomePage                    EQUATE('GHP$')  !01
REQUEST:Login                       EQUATE('LOG$')  !02
REQUEST:TakeLogin                   EQUATE('KLG$')  !03
REQUEST:Register                    EQUATE('REG$')  !04
REQUEST:TakeRegister                EQUATE('KRG$')  !05
REQUEST:Query                       EQUATE('QRY$')  !06
REQUEST:TakeQuery                   EQUATE('KQY$')  !07
REQUEST:ShowUpdateForm              EQUATE('SUP$')  !08
REQUEST:TakeWebUpdate               EQUATE('KWU$')  !09
REQUEST:Renewal                     EQUATE('REN$')  !10
REQUEST:TakeRenewal                 EQUATE('KRN$')  !11
REQUEST:PurchasePage                EQUATE('PPG$')  !12
REQUEST:DownLoadPage                EQUATE('DLP$')  !13
REQUEST:RedirectPage                EQUATE('RDP$')  !14
REQUEST:SetMenuText                 EQUATE('SMT$')  !15
REQUEST:SetMenuLink                 EQUATE('SML$')  !16
REQUEST:Exit                        EQUATE('QIT$')  !17
REQUEST:Quit                        EQUATE(REQUEST:Exit)
REQUEST:CloseWindow                 EQUATE('WXX$')  !18
REQUEST:Back                        EQUATE('BAK$')  !19
REQUEST:HelpUrl                     EQUATE('HLP$')  !20
REQUEST:HelpMail                    EQUATE('HML$')  !21
REQUEST:FileUpload                  EQUATE('FUP$')  !22
REQUEST:UnKnown                     EQUATE('UNK$')  !23
REQUEST:ImageFile                   EQUATE('IMG$')  !24
REQUEST:MediaFile                   EQUATE('MED$')  !25
REQUEST:StyleSheet                  EQUATE('CCS$')  !26
REQUEST:Document                    EQUATE('DOC$')  !27
REQUEST:Executable                  EQUATE('EXE$')  !28
REQUEST:TextHTML                    EQUATE('HTM$')  !29
REQUEST:JavaScript                  EQUATE('JAV$')  !30
REQUEST:Change                      EQUATE('CHG$')  !31
REQUEST:TakeEditButton              EQUATE('KED$')  !32
REQUEST:Insert                      EQUATE('INS$')  !33
REQUEST:Delete                      EQUATE('DEL$')  !34
REQUEST:ThinClientFile              EQUATE('TCF$')  !35
REQUEST:ThinClientQuery             EQUATE('TCQ$')  !36
REQUEST:ThinClientPostFile          EQUATE('TCP$')  !37
REQUEST:ThinClientAcceptLogin       EQUATE('TCL$')  !38
REQUEST:ThinClientExit              EQUATE('TCX$')  !39
REQUEST:HTTPDeleteFile              EQUATE('DTF$')  !40
REQUEST:HTTPBrowserPutFile          EQUATE('PTF$')  !41
REQUEST:HTTPGetFile                 EQUATE('GTF$')  !42
REQUEST:LoginDetailsRequest         EQUATE('LDR$')  !43
REQUEST:UserCustomRequest           EQUATE('UCR$')  !44
REQUEST:RefreshQuery                EQUATE('RFQ$')  !45
REQUEST:ClientLogOut                EQUATE('LGX$')  !46
REQUEST:ThinClientReadDir           EQUATE('DIR$')  !47
REQUEST:HTTPPutFileSecure           EQUATE('PSF$')  !48
REQUEST:HTTPGetFileSecure           EQUATE('GSF$')  !49
REQUEST:HTTPSendLogFile             EQUATE('RLF$')  !50
REQUEST:UsageCounter                EQUATE('CNT$')  !51
REQUEST:Help                        EQUATE('HLP$')  !52
REQUEST:GetIP                       EQUATE('GIP$')  !53
REQUEST:RefreshLastBrowse           EQUATE('RLB$')  !54
REQUEST:TakeRegisterExtra           EQUATE('KGE$')  !55
```

```
REQUEST:ChtDemoAppRequest           EQUATE('DAR$')  !56
REQUEST:ChtDemoTutorialRequest      EQUATE('DTR$')  !57
REQUEST:ChtMemberCheckinRequest     EQUATE('MCR$')  !58
REQUEST:ShowWebPreviewForm          EQUATE('WPF$')  !59
REQUEST:UserCustom                  EQUATE('RUC$')  !60
REQUEST:WebMailRequest              EQUATE('WMR$')  !61
```

Not all of these are implemented in HNDSUBSV.CLW with built-in procedures. Many are. All but one other one are ignored by this server implementation. They're not relevant to it.

User Custom Request

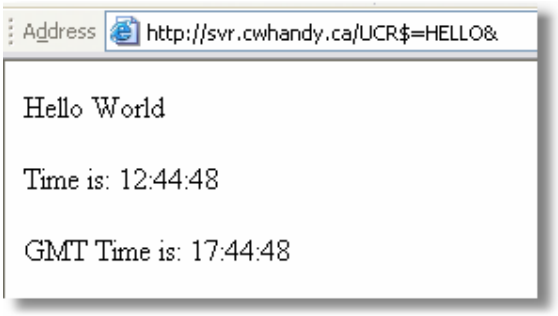
One other implemented command, included as an exercise for your experimentation, is **REQUEST:UserCustomRequest (UCR$)**. The purpose of this command is to cause the server to branch to a built-in server procedure called **ProcessUserCustomRequest()** which in and of itself does nothing. It's simply a place-holder meant to be overridden (embedded into) by the developer. Inside this procedure you're able to intercept parameters in the way we've already illustrated in the page-counter example. What you do with that information and what you return to the browser as a result of it, is wide open. A user custom request is only a hook onto which you can hang some code.

By way of illustration we've provided an small example, embedded into HNDSLFSV.APP:

```
Server.ProcessUserCustomRequest PROCEDURE ()
ReturnValue          BYTE,AUTO
TakenValue           CSTRING(100)
  CODE
  !This is an example of using user custom request to perform a task.
  !This particular command is typed in on the browser
  !as http://www.yoursite.com/UCR$=HELLO&
  IF SELF.LastCustomRequest = 'HELLO'
     !At this point you can send back a file
     !SELF.SendFile('mypage.html')
     !Or send back a hand-built html page (or whatever)
     SELF.ClearBuffer()
     SELF.AddBufferItem('<HTML><HEAD></HEAD><BODY>',2,1)
     SELF.AddBufferItem('Hello World<p>', 3,)
     SELF.AddBufferItem('Time is: ' & FORMAT(CLOCK(),@T4B),3,1)
     SELF.AddBufferItem('<p>GMT Time is: ' & FORMAT(CLOCK() |
            - (Server.GetGMTOffset()),@T4B),3,1)
     SELF.AddBufferItem('<p></BODY></HTML>',2,1)
     SELF.SendBuffer()
     RETURN True
  ELSIF SELF.LastCustomRequest = 'TAKEVALUE'
     !This particular command is typed in on the browser
     !as http://www.yoursite.com/UCR$=TAKEVALUE&VALUEIS=xxxxxx&
     TakenValue = SELF.ExtractCommandLineItem('VALUEIS')
     !TakenValue now contains "xxxxxx"
  ELSE
     !SELF.LastCustomRequest is assumed to be a file name
     !eg: http://www.yoursite.com/UCR$=index.html&
     SELF.SendFile(SELF.LastCustomRequest)
  END
```

Try it for yourself. Start your HNDSLFSV.APP server, then launch your browser. In the browser command line enter the URL or IP of your server as follows: http://xxx.xxx.xxx.xxx/UCR$=HELLO&. Insert your url or IP specific values where the xxx's are.

Don't forget the ampersand character after the upper case word HELLO. Ampersand acts as a standard parameter separator and is used by all browsers to separate name=value pairs. If you did this correctly you should have gotten back a very basic HTML page like the following:



Of course, a command of this type need not be hand-typed into the browser command line area. It can be embedded in a link, or a button as needed. Suffice to say, this one is wide open for you to use or abuse as you see fit. In and of itself, UCR$ does nothing unless you code it into your server. And, unlike IIS, no one else can misuse your server by sending it a script to make it do something abusive. That power is reserved for the server developer himself. Please use your power for good.

Illustration 6.10

Building An Application Like HNDSLFSV.APP From Scratch

We've left this until the end of LESSON 1 because, it's actually the easiest thing of all to build a static page web server from scratch using a Jump Start procedure template. Here are the steps:

1. Start a new Clarion application, no dictionary required.
2. Leave the Main (ToDo) procedure untouched for the time being.
3. Execute Menus *Application -> Global Properties -> Extensions*.
4. Insert Extension Template AACHTControlPanel.
5. On AACHTControlPanel click the Jumpstart Procedures Button.
6. Navigate to the "HTTP Web Servers" tab and click "Static Page HTTP Web Server".
7. Read the information provided on the dialog presented and check "Import Procedure?"
8. On the Procedure Name Clash dialog, click "Replace All".
9. Click OK, OK, OK, OK, until you're back on the application procedure tree.

Three JumpStart Procedures are added to your application: **JumpStartStaticPageServer**, **JumpStartSplash** and **JumpStartHTTPRunHelpMedia**. At this point, the application will compile without errors but is not quite ready to run since we haven't hooked the server procedure to Main().

Follow these steps:

10. Right click JumpStartStaticPageServer and select rename.
11. When the rename dialog appears, strike CtrlC on your keyboard (to copy the name to the clipboard).
12. Click cancel on the rename dialog without changing the name. (Lazy or what?)
13. Right Click Main(ToDo) and select Properties.
14. In the "Select Procedure Type" dialog, select the "Templates" tab followed by "Source Procedure" and click Select.
15. Click OK to return to the procedure tree.
16. Right Click Main(Source) and select Source.
17. In the first embed slot after the word CODE strike CtrlV on your keyboard (to paste the name of your server procedure).
18. Make sure this procedure name is not coloured red like a label by moving it off the left margin.
19. Add a set of parenthesis () to indicate to yourself that this is not a variable but a procedure.

What you have now should look like this:

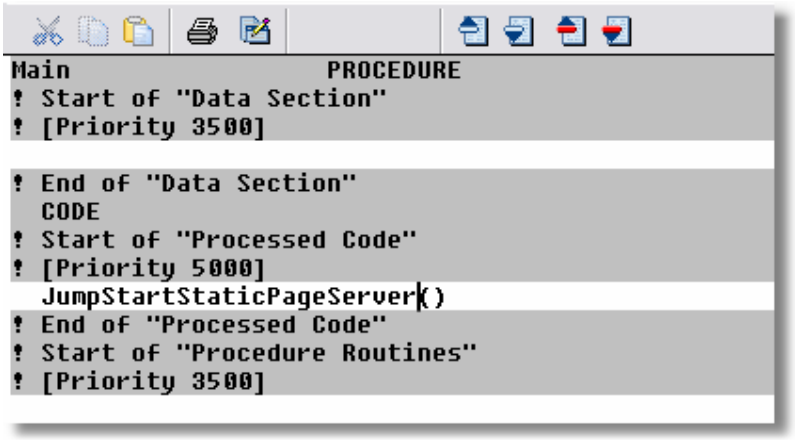


Illustration 6.11

20. Exit Main(Source) and save your code.
21. Right Click Main(Source) and select Procedures.
22. In the list presented, select **JumpStartStaticPageServer**, followed by a click on OK.
23. Compile and Run

Your server is ready to be customized, added to, extended as you see fit. Our sample server executable is compiled using C6 so we've made some screen control adjustments to provide flat entry controls and have set the server status control to transparent. That provides a more up-to-date XP look especially if you also include an XP manifest and some visual indicators using *Application->Global Properties->App Settings*.

LESSON 1 - EXERCISES

10. Use your favorite static page building tool (we use HELP and MANUAL), bash together a few web pages for your scratch-built server and send us the URL or IP to have a look. If you're stuck for time, set up the default web site pages that we've provided and send us the URL or IP to that. The point we're interested in determining is: Did you get the server working correctly?

11. On your home page, implement a page counter that references your server.

12. Spend some time experimenting with the User Custom Request functionality. Send us a command line link in an email that we can click to make something happen on your server via User Custom Request. It doesn't have to be anything elaborate. Just something that illustrates you have understood the concept. Paste your Clarion code into the return email for us to have a look.

13. Open HNDBRWSV.CLW and find the function called **HNDBrowserServer.TakeSocketMessages()**. Study the code and explain in your own words what's going on in this function. Explain, also what **HNDBrowserServer.AcceptThisSocket()** does.

14. Open HNDBRWSV.INC (the header file for HNDBRWSV.CLW). Please count and name all the CHT classes being used by this module either directly by derivation or indirectly by composition. Feel free to provide as much detail as you're able. If you understand the principles involved here, you'll begin to appreciate why CHT is promoted as an "Integrated" tool kit.

15. Open HNDSUBSV.CLW and find the function called **HNDSubscriptionServer.ParseCommand()**. Note that this is the last function that HNDBrowserServer.TakeSocketMessages() calls before it returns. Explain what **ParseCommand()** does as best you can from examining the code.

16. Two templates are used in the Jump Start server procedure called **JumpStartStaticPageServer()** to implement the static page server functionality. Name these templates. Explain briefly what each template does.