



HTTP Client Deux

Developer Documentation v1.1.1

©1998-2002 Deep Sky Technologies, Inc. All Rights Reserved.
Published and Distributed Worldwide by Deep Sky Technologies, Inc.

Deep Sky Technologies, Inc.
P.O. Box 6897
Vero Beach, FL 32961-6897
772.794.9494



Software License and Limited Warranty

Please read this license carefully before using the software. By using the software, you agree to become bound by the terms of this agreement, which includes the software license and warranty disclaimer (collectively referred to herein as the "agreement"). This agreement constitutes the complete agreement between you and Deep Sky Technologies, Inc. If you do not agree to the terms of this agreement, do not use the software and promptly destroy all copies in your possession, physical and electronically.

1. Ownership of Software: The enclosed manual and computer programs ("Software") were developed and are copyrighted by Deep Sky Technologies, Inc. ("DSTi") and are licensed, not sold, to you by DSTi for use under the following terms, and DSTi reserves any rights not expressly granted to you. DSTi retains ownership of all copies of the Software itself. Neither the manual nor the Software may be copied in whole or in part except as explicitly stated below.

2. License: DSTi, as Licensor, grants to you, the Licensee, a non-exclusive, non-transferable right to use this Software subject to the terms of the license as described below:

- a. You may make backup copies of the Software for your use provided they bear the DSTi copyright notice.
- b. You may use this Software in an unlimited number of custom or commercial databases or applications created by the original licensee. No additional product license or royalty is required.

3. Restrictions: You may not distribute copies of the Software to others (except as an integral part of a database or application within the terms of this License) or electronically transfer the Software from one computer to another over a network. You may distribute copies of the Software as an integral part of a development shell or non-compiled commercial database as long as the DSTi copyright notices and documentation remain intact with the distribution. The Software contains trade secrets and to protect them you may not decompile, reverse engineer, disassemble, or otherwise reduce the Software to a human perceivable form. You may not modify, adapt, translate, rent, lease, loan or resell for profit the software or any part thereof.

4. Termination: This license is effective until terminated. This license will terminate immediately without notice from DSTi if you fail to comply with any of its provisions. Upon termination you must

destroy the Software and all copies thereof, and you may terminate this license at any time by doing so.

5. Update Policy: DSTi may create, from time to time, updated versions of the Software. At its option, DSTi will make such updates available to the Licensee.

6. Warranty Disclaimer: The software is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. DSTi does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the software or written materials in the terms of correctness, accuracy, reliability, currentness or otherwise. The entire risk as to the results and performance of the software is assumed by the Licensee. If the software or written materials are defective you, and not DSTi or its dealers, distributors, agents, or employees, assume the entire cost of all necessary servicing, repair or correction. No oral or written information or advice given by DSTi, its dealers, distributors, agents, or employees shall create a warranty or in any way increase the scope of this warranty, and you may not rely on such information or advice. This warranty gives you specific legal rights. You may have other rights, which vary from state to state.

7. Governing Law: This agreement shall be governed by the laws of the State of Florida.

Copyrights and Trademarks

All trade names referenced in this document are the trademark or registered trademark of their respective holder.

BASh, BASh Pro, TCP Deux, TCP Deux Pro, SMTP Deux, SMTP Deux Pro, POP3 Client Deux, POP3 Client Deux Pro, FTP Client Deux, FTP Client Deux Pro, HTTP Client Deux, HTTP Client Deux Pro, eTrans, TCP Server Deux, TCP Server Deux Pro, HTTP Server Deux, HTTP Server Deux Pro, HTTP Log Deux, and HTTP Log Deux Pro are copyright Deep Sky Technologies, Inc.

4th Dimension, ACI, ACI US, 4D Compiler, 4D, 4D Server, 4D Client, and 4D Insider are trademarks of 4D, Inc.

4D Internet Commands plugin provided courtesy, and with permission, of 4D, Inc.

Internet Commands plugin provided courtesy, and with permission, of Christian Quest.

Macintosh and MacOS are trademarks of Apple Computer, Inc.

Windows is a trademark of Microsoft Corporation.

Preface

The HTTP Client Deux component is designed to work in conjunction with many other components. Specifically, the HTTP Client Deux component requires that the BASH component, available for free from DSTi, and TCP Deux component both be installed already in your database structure file.

Make certain that you view the compatibility matrix for components available to make certain you are using compatible versions of the different components required. There is a compatibility matrix available in this manual; the most recent compatibility matrix is available on the DSTi web site.

Acknowledgements

The creation of the HTTP Client Deux component is not directly attributable to any single person. Particular pieces of functionality within the HTTP Client Deux component may be from the direct knowledge and experience of certain developers, but the overall concept and construction of the HTTP Client Deux component has come from all of the developers at Deep Sky Technologies, Inc.

In particular, the tireless efforts of Robert T. McGoye have contributed the most to the HTTP Client Deux component. His ability, and patience, to be able to tolerate the swings in the atmosphere at DSTi, have proven to be invaluable in the development of the HTTP Client Deux component.

Later tweaks and additions to the HTTP Client Deux component have resulted from the training of James A. Crate. Mr. Crate's experience in many different programming environments has provided refreshing insights into the overall structure and organization of the core routines at DSTi, the same core routines which are available in the BASH and HTTP Client Deux components.

Finally, I, Steven G. Willis, might have had something to do with the creation of the HTTP Client Deux component...

Features

HTTP Client Deux is a 4th Dimension component that provides a single set of HTTP request methods for use throughout your 4D projects. Specifically, HTTP Client Deux provides cross-platform compatibility for all HTTP operations, significantly simplifying the need for 4D programmers to request remote HTTP and HTTPS directly from within 4D code. For most requests, it is as simple as a single method call to retrieve the remote object.

HTTP Client Deux provides many automation and management functions for use in writing HTTP request and response processing code. It automatically handles most of the common operations involved in managing HTTP communications in 4D.

HTTP Client Deux works "on top" of TCP Deux, another 4D component available from DSTi. This provides completely transparent operation with 4D Internet Commands v6.7.x, 4D Internet Commands v6.8.x, Internet ToolKit v2.0.x, and Internet ToolKit v2.5.x; any of these plugins can be used without any coding changes. The only exception to this is for HTTPS requests to function properly you must be using a valid copy of ITK Pro v2.5.x.

TCP Deux does require that the BASH 4D component be installed for it to operate correctly. BASH is a 4D component available for free from DSTi and it handles all of the core routines needed to properly run TCP Deux and HTTP Client Deux.

All of the code within the HTTP Client Deux component is ready to use immediately. There are no compiler issues to be concerned about; all of the variables are typed in compiler methods for even the most stringent 4D developers. Just install the component in your structure and start calling the methods within it. It is really that simple!

System Requirements

The HTTP Client Deux component is compatible with both Macintosh and Windows installations of 4th Dimension.

Since it is a component, it does require at least version 6.7 of 4th Dimension or above, including 4D Insider v6.7 or above for installation.

Other than the normal hardware and software requirements for your version of 4th Dimension, there are no other minimum requirements for proper use of this software.

Support

Support is provided for HTTP Client Deux component free of charge for all currently licensed users. Included support services provided for all currently licensed users encompasses all of the online support services available through the DSTi web site (email, FAQ, messaging, etc.). Check the DSTi web site for current direct support options available; we are always working to offer more resources for your needs.

Contact information, including email address(es), phone number(s), and a Contact Us request form, for Deep Sky Technologies, Inc., can be found on the DSTi web site located at:

<http://www.deepskytech.com/>

If there are terms or conventions which you find difficult to understand in relation to the HTTP Client Deux component or TCP networks in general, feel free to contact Deep Sky Technologies, Inc., support. We will be more than happy to help you in any way we reasonably can. And, only through your questions do we know what subjects to include in future versions of this manual.

Components

A component groups various 4D objects (tables, project methods, forms, menu bars, variables, etc.) representing one or more additional functions. Developing a 4D component providing electronic mail functionality is one such example. A component is autonomous and must be able to be installed in any 4D structure.

Components are defined, generated, and installed with the help of 4D Insider. The component definition is based on the cross referencing analysis performed by 4D Insider (target objects and source objects).

Unlike libraries and groups, components embed the idea of security of objects that they compose. During the development phase of the component, each object is attributed an access type, "Public", "Protected" or "Private". This attribute determines whether each object will be visible or modifiable in 4th Dimension and in 4D Insider once the component is installed within a 4D database.

Installing and Updating HTTP Client Deux

Installing HTTP Client Deux or updating an existing version of HTTP Client Deux within a 4D database is performed using 4D Insider. The activity primarily consists of installing the HTTP Client Deux component in a database structure opened with 4D Insider (installing the HTTP Client Deux component in a library is not supported at this time).

4D Insider will manage possible conflict issues within the installation and will inform you as they are detected. Though, with the naming conventions used within the HTTP Client Deux component and the limited number of object names, conflicts should be very rare.

To install or update the HTTP Client Deux component, follow these very simple steps:

Open the uncompiled structure that you wish to install HTTP Client Deux into using 4D Insider.

Choose the "Install/Update..." command in the "Components" menu.

A standard open file dialog box will appear.

Select the HTTP Client Deux component file and click on the "Open" button.

4D Insider parses the HTTP Client Deux component and prepares to integrate it with your open database. 4D Insider will detect if the operation is an installation or an update of the HTTP Client Deux component.

In the event of a new installation, all HTTP Client Deux objects are installed.

In the event of an update, 4D Insider compares the version numbers of both the currently installing HTTP Client Deux component and the already installed HTTP Client Deux component. If the date of the "new" component is older than the already installed component, a dialog box will alert you, allowing you to then "Continue" or "Cancel" the update.

4D Insider replaces old objects with newer objects within the HTTP Client Deux component and adds new objects from the new HTTP Client Deux component. 4D Insider takes into account "public" objects having been modified by you (e.g.

"_ERROR" methods) and will prompt you to either save or replace them. If any other conflicts arise from the installation or update of the HTTP Client Deux component, 4D Insider will prompt you with an appropriate dialog box.

Save the database in 4D Insider.

Place a copy of the Affix HTTPcd document in the 4DX folder.

The Affix HTTPcd document contains essential data and resources for many of the methods within the HTTP Client Deux component. For many of the methods within the HTTP Client Deux component to function properly, the Affix HTTPcd document must be in the 4DX folder for the current structure.

On Macintosh, the Affix HTTPcd document is entitled **Affix_HTTPc_Deux.4DX** (under 4D v6.8.x, there is a carbonized version entitled **Affix_HTTPc_Deux.4CX**) and is located in the Mac4DX directory in the HTTP Client Deux component's archive. The document should be copied into the Mac4DX folder of your current structure. If the HTTP Client Deux component is going to be used in all of your 4D projects, the Affix HTTPcd document can instead be placed within the Mac4DX folder within the 4D folder of your system.

On Windows, the Affix HTTPcd document is actually two documents: **Affix_HTTPc_Deux.4DX** and **Affix_HTTPc_Deux.RSR**. These two documents correspond to the data fork and the resource fork of the Affix HTTPcd document used on Macintosh. These documents are located in the Win4DX directory in the HTTP Client Deux component's archive. These documents should be copied into the Win4DX folder of your current structure. If the HTTP Client Deux component is going to be used in all of your 4D projects, the Affix HTTPcd document can instead be placed within the Win4DX folder within the 4D folder of your system.

For client/server installations in cross-platform environments, both the Macintosh and Windows versions of the Affix HTTPcd document should be installed.

Call the method *INIT_HTTPcd* early in the On Startup database method.

To initialize the HTTP Client Deux component in your code, place a call to the method ***INIT_HTTPcd*** early in your **On Startup** database method. However, this method must be after the calls to ***INIT_BASh*** and ***INIT_TCPd***.

Details about the ***INIT_HTTPcd*** method can be found in the module and method documentation, below.

The HTTP Client Deux component is now installed/updated in your database and is listed on the "Components" page of the 4D explorer.

Managing Installation Conflicts

On very rare occasions, when the HTTP Client Deux component is installed or updated in your 4D database, several questions and conflicts may arise. In the event of an update, 4D Insider will detect that you have modified one of more "Public" objects in HTTP Client Deux after the initial installation. Or, one or more objects of the same type and of the same name may already exist in your database and in the HTTP Client Deux component.

4D Insider detects and solves these conflicts during installation:

Modified public objects (updates only)

In this case, 4D Insider alerts you by a dialog box, allowing you to choose an update mode:

Replace the object

Replace all objects

Do not replace the object

Stop installation

Name conflicts

In this case, 4D Insider stops the HTTP Client Deux installation process, alerts you through a dialog box and saves the list of objects in conflict. This list is stored as a text file in the 4D database folder.

Naming conflicts between logical objects, such as variables, are managed by 4D Insider, in a manner that allows database compilation and avoids conflicts between HTTP Client Deux and other 4D components.

It may be necessary to rename certain objects in your database or in other components in order to be able to install the HTTP Client Deux component.

If any naming conflicts do occur between HTTP Client Deux and other 4D components, please notify Deep Sky Technologies, Inc., immediately.

Affix HTTPcd Document

The Affix HTTPcd document (entitled **Affix_HTTPc_Deux.4DX** on Macintosh; entitled **Affix_HTTPc_Deux.4DX** and **Affix_HTTPc_Deux.RSR** on Windows; under 4D v6.8.x, there is a carbonized version entitled **Affix_HTTPc_Deux.4CX**) contains essential data and resources for many of the methods within the HTTP Client Deux component. For many of the methods within the HTTP Client Deux component to function properly, the Affix HTTPcd document must be in the 4DX folder for the current structure. For distributed and installed versions of your 4D projects, the Affix HTTPcd document must be available in the 4DX folder for the HTTP Client Deux methods to continue to function properly.

Note: it is best to consider the Affix HTTPcd document another plug-in within your 4D project. Though there no actual plug-in calls available within the Affix HTTPcd document, it does contain data and resources essential to the operation of the HTTP Client Deux methods. The HTTP Client Deux component has been designed to find the Affix HTTPcd document correctly in all environments (any platform, any 4DX folder, single user or clients/server, etc.). Since the Affix HTTPcd document is configured similarly to plug-ins, 4D and the HTTP Client Deux component will automatically manage the document for you in all of the possible installations of a 4D project.

4D v6.8.x

With the availability of 4D v6.8.x, the complete 4th Dimension environment is now fully carbonized. 4D as a carbonized application allows for

a single set of tools to function on either MacOS X or MacOS v9.x using CarbonLib.

With the release of 4D v6.8.x, there is now a new plugin architecture available for third party developers. As well, there are some changes which have been made in the actual plugin hierarchy and naming conventions. There have also been changes made to the component architecture within the 4D product line. Reading the release notes for 4D v6.8.x is a great source of information regarding these changes.

HTTP Client Deux is currently available with compatibility for 4D v6.8.x. This comes in the form of new affix documents for use with 4D v6.8.x. The HTTP Client Deux archive contains both 4D v6.7.x and 4D v6.8.x compatible versions of the component and affix documents.

When updating an existing database from 4D v6.7.x to 4D v6.8.x, we have found that installed components will no longer update properly. The first time that a component is updated after upgrading a 4D structure, the component must first be removed from the structure before installing the new version of the component. We have not seen any problems with doing this other than the extra step required to remove the component using 4D Insider.

It is also important that you install the correct affix documents for your environment. Whether you are running under MacOS X or MacOS 9 is not a factor. Rather, whether you are using 4D v6.7.x or 4D v6.8.x is the determining factor. Copy the appropriate documents for your current version of 4D from the component archive for use in your 4D structure. The differences between the 4D v6.7.x and 4D v6.8.x affix documents is simple to determine; the names of the documents compatible with 4D v6.7.x end in ".4DX" and the names of the documents compatible with 4D v6.8.x end in ".4CX".

Since 4D v6.8.x is still currently in beta, it is best to report any issues with components immediately to Deep Sky Technologies, Inc. We can research the issue very quickly and notify 4D, Inc./4D SA of any compatibility and functional problems quickly so that the final release of 4D v6.8.x is as bug free as possible.

Uninstalling HTTP Client Deux

4D Insider allows you to uninstall the HTTP Client Deux component from your 4D database.

To uninstall HTTP Client Deux from your 4D database:

Using 4D Insider, open your database containing the copy of HTTP Client Deux to be uninstalled.

In the "Main" listing window, select the HTTP Client Deux component.

Consider again how great the HTTP Client Deux component is and make certain that you will really no longer need it in your 4D database.

Select the "Uninstall..." command in the "Components" menu.

This command is only active when a component is installed in the database. A dialog box appears allowing you to confirm or cancel the operation. If you uncertain about the previous step then the cancel option is probably your best choice at this time.

Click "OK" to validate the operation.

Remove the Affix HTTPcd document from your 4DX folder.

Remove the call to the method *INIT_HTTPcd* from your On Startup database method.

All objects from the HTTP Client Deux component are deleted from your 4D database. Obviously, you are now very sad to no longer have the HTTP Client Deux component in your 4D database. Crying is allowed...

HTTP Client Deux Conventions

Throughout this manual, and all other documentation and supporting materials, included with the HTTP Client Deux component package, there are different core knowledge which is essential to know and understand. With this knowledge, basically concerning the conventions used in TCP communications and conventions used within the HTTP Client Deux component, you will be able to more easily and efficiently utilize the functionality available within this software package.

If there are other terms or conventions which you find difficult to understand in relation to the HTTP Client Deux component or TCP communications in general, feel free to contact Deep Sky Technologies, Inc., support. We will be more than happy to help you in any way we reasonably can. And, only through your questions do we know what subjects to include in future versions of this manual.

Basics of HTTP

This section explains the basics to understanding HTTP, HyperText Transport Protocol. HTTP is the protocol used to view web pages on the Internet. Understanding it from the perspective of a client, like a web browser, is actually not too difficult. For detailed information to understand HTTP in greater depth, it is recommended that RFC 1945, the RFC for HTTP, be read. RFC 1945 is available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

It is worth stating that this description of the workings of HTTP is greatly simplified. It is by no means comprehensive in any way. Rather, it serves as a good starting point for programmers just becoming familiar with HTTP. For more detailed information about HTTP, reading the RFCs relating to HTTP is highly recommended.

As well, learning specifically about web integration with 4th Dimension can be done by reading David Adams book, "4D Web Companion". Information about Mr. Adams new book can be found on his web site, located at:

<http://www.island-data.com/>

It is worth noting, too, that throughout this section there are examples of HTTP request and HTTP responses. Often, lines in an HTTP request and HTTP response will be delimited by a pairing of bytes, specifically a Carriage Return (ASCII code 13) and Linefeed (ASCII code 10). In this text, a carriage return/linefeed pairing is represented by:

[CRLF]

In places where only a carriage return is within content, the following symbol will be used:

[CR]

Overview

At the simplest level, HTTP is used to request documents from a remote server over TCP. Commonly, this consists of a single request being made for a single document; a single response is then returned from the server to the requesting client. There are clear and quantifiable formats for both the request and the response.

Every HTTP request and response consists of a two sections: the header and the content. The header contains information about the client, the server, the document, the content, and much more. The content section of a request and response commonly contains the bulk of the information being transferred, like the document contents that were requested. It is not uncommon for the content section to be empty in either a request or a response.

There are different request types that can be made by a client. The most common type of request is a simple request to retrieve a remote document. For instance, when you type into your browser the URL:

<http://www.deepskytech.com/downloads.html>

There is a request made for the document entitled “downloads.html” at the root “/” of the remote host “www.deepskytech.com”. The response for this request happens to be an HTML document which describes a web page within Deep Sky Technologies, Inc., web site.

Another example of such a request is:

http://www.deepskytech.com/images/logo_dsti_v3_01.gif

This example is requesting an image, specifically the image named “logo_dsti_v3_01.gif” within the directory “/images/” at the remote host “www.deepskytech.com”. It just happens that the document at this location is an image stored in the GIF format.

Requests can also contain variable information (e.g. search parameters) for a remote host to act upon. An example of such an URL would be:

<http://www.google.com/search?q=4th+dimension>

This example URL is requesting a search using the term “4th dimension” within the remote host “www.google.com”.

The three example requests presented so far are commonly called “GET” requests, or more colloquially as just “GETs”. By far, the large major-

ity of requests made using HTTP are done using GETs. In a GET request, the header section contains all of the transmitted information; the content section is empty.

It is also possible for parameters which are sent within a request to not be displayed within the requested URL. The parameters are still sent to the remote host in the request, but they are placed within another location in the request such that they are not a part of the requested URL. Requests of this type are called “POST” requests, or more commonly “POSTs”. It is common for more complicated forms on a web page to be sent to the remote server as a POST. And, in such a case, it is commonly called POSTing the page to the server. In a post request, the header contains most of the transmitted information, with the exception of the content section which contains the variable parameters which are being sent.

Another type of HTTP request is called the “HEAD” request. Not all HTTP servers support HEAD requests. But, using HEAD requests can help in many instances. In short, a HEAD request will return the same thing as a GET request but without the contents of the request document; instead, only the header of the response, containing vital information about the status and contents of the requested document are returned, without the often much larger full contents of the remote document being returned. HEAD requests are commonly used to confirm the existence, document type, and total size of a remote document in an efficient manner. A HEAD request is fully contained within the header section of the request; the content section of the request is empty.

When a server receives a request, it is the responsibility of the server to send a response. This response may even be just an error of some kind; error response of all kinds are available within the HTTP specification to cover any situation. A response will always have a header section. Almost always, a response will have data within the content section; the most common response which has an empty content section would be some implementations of error response and any response to a HEAD request.

Responses must always contain a response code. The response code is a numeric value which indicates the results of the server processing the request. The status code may indicate such things as success, movement of the request document, errors detected in the request, privilege challenges, document not found, or server errors. The response code is the single most important piece of any HTTP response and should always be the first item processed by a client after making an HTTP request.

GET Requests

GET requests are the most commonly used HTTP request type. A GET request contains all of the information related to the URL requested within the actual URL. Additional lines of information in the header are used for additional information, compatibility, authentication, and transport details. The content of the HTTP GET request should always be empty.

The following is a sample HTTP GET request:

```
GET /downloads.html HTTP/1.0[CRLF]
User-Agent: HTTP Client Deux v1[CRLF]
Host: www.deepskytech.com[CRLF]
Accept: text/html; image/gif; image/jpeg[CRLF]
[CRLF]
```

Notice that the type of HTTP request being made is the very first piece of information sent. The document being requested, “/downloads.html” is then the very next item in the request. The subsequent lines then detail more information about the request being made and the requesting software.

Another sample request is as follows:

```
GET /cgis/bounce.acgi?foo=whatever HTTP/1.0[CRLF]
User-Agent: Mozilla/4.78 (Macintosh; U; PPC)[CRLF]
Host: www.store-secure.com[CRLF]
Accept: image/gif, image/x-bitmap, image/jpeg, */*[CRLF]
[CRLF]
```

This GET contains a single named parameter as part of the request. As well, there are other differences in the details of the request.

In general, the above few sample requests are enough to build a simple template, of sorts, for making HTTP GET requests. And, in actuality, this is how the routine **HTTPcd_GET_Simple** functions. The template used for this method is as follows:

```
GET ^02 HTTP/1.0[CRLF]
User-Agent: HTTP Client Deux v1[CRLF]
Host: ^01[CRLF]
Accept: text/html; image/gif; image/jpeg[CRLF]
[CRLF]
```

The document delimiter “/”, document name, and any direct and search arguments are substituted for the value of “^02” in the above template. The host is substituted for into the template for the value of “^01”.

In case this template needs to be modified, it is stored in the resource fork of the Affix HTTP Client Deux document, in 'TEXT' resource ID 29000.

For more details about HTTP GET requests, read RFC 1945 available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

HEAD Requests

HEAD requests differ from GET requests only in the actual request type. All other values in the request are almost always the exact same as an equivalent GET request.

The following is a sample HEAD request:

```
HEAD /cgis/bounce.acgi?foo=whatever HTTP/1.0[CRLF]
User-Agent: Mozilla/4.78 (Macintosh; U; PPC)[CRLF]
Host: www.store-secure.com[CRLF]
Accept: image/gif, image/x-bitmap, image/jpeg, */*[CRLF]
[CRLF]
```

The following is the template used for the method ***HTTPcd_HEAD_Simple*** when making requests:

```
HEAD ^02 HTTP/1.0[CRLF]
User-Agent: HTTP Client Deux v1[CRLF]
Host: ^01[CRLF]
Accept: text/html; image/gif; image/jpeg[CRLF]
[CRLF]
```

The document delimiter “/”, document name, and any direct and search arguments are substituted for the value of “^02” in the above template. The host is substituted for into the template for the value of “^01”.

In case this template needs to be modified, it is stored in the resource fork of the Affix HTTP Client Deux document, in 'TEXT' resource ID 29002.

For more details about HTTP HEAD requests, read RFC 1945 available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

POST Requests

An HTTP POST request differs from a normal GET request in that it is commonly used for POSTing form data (data filled in on a form by a user on a previous page). The POSTed data is always formatted as name value pairs in the content of the request.

The following is a simple example POST request:

```
POST /4dz.acgi$freeware_search HTTP/1.0[CRLF]
User-Agent: HTTP Client Deux v1[CRLF]
Host: www.4dzine.com[CRLF]
Accept: text/html; image/gif; image/jpeg[CRLF]
Content-Type: application/x-www-form-urlencoded[CRLF]
Content-Length: 73[CRLF]
[CRLF]
append_wildcard=&prepend_wildcard=&search_type=AND&search_terms=component[CRLF]
[CRLF]
```

As you can see, a POST request is a little more complex than a GET request. But, in general, the header of the POST request is very similar to a GET request. The primary difference is that there is content to the request beyond the header, content which is often used directly by the server to process the request. The above example obviously does a Free-ware search within the 4D Zine web site for the word "component".

The following is the template used for the header of the request within the method ***HTTPcd_POST_Simple***:

```
POST ^02 HTTP/1.0[CRLF]
User-Agent: HTTP Client Deux v1[CRLF]
Host: ^01[CRLF]
Accept: text/html; image/gif; image/jpeg[CRLF]
Content-Type: application/x-www-form-urlencoded[CRLF]
Content-Length: ^03[CRLF]
[CRLF]
```

The document delimiter "/", document name, and any direct and search arguments are substituted for the value of "^02" in the above template. The host is substituted for into the template for the value of "^01". The length of the content area is substituted for the value of "^03" in the above template.

In case this template needs to be modified, it is stored in the resource fork of the Affix HTTP Client Deux document, in 'TEXT' resource ID 29001.

For more details about HTTP POST requests, read RFC 1945 available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

Responses

HTTP responses are all structured the same regardless of the request type. The sole mentionable exception is that the content of a HEAD request is always empty. Otherwise, an HTTP response follows similarly to an HTTP request, containing both a header section and a content section.

The following is an example HTTP response:

```
HTTP/1.1 200 OK[CRLF]
Server: WebSTAR/4.4(SSL) ID/73166[CRLF]
Date: Mon, 30 Jul 2001 01:57:03 GMT[CRLF]
Connection: close[CRLF]
MIME-Version: 1.0[CRLF]
Content-type: text/html[CRLF]
Pragma: no-cache[CRLF]
[CRLF]
<HTML>[CR]
<HEAD>[CR]
  <TITLE>Deep Sky Technologies, Inc.</TITLE>[CR]
</HEAD>[CR]
[CR]
<BODY>[CR]
[CR]
This is my web page with lots of HTML on it...[CR]
[CR]
</BODY>[CR]
[CR]
</HTML>
```

It is a simple matter to read the response and understand much of it without any explanation. Of particular interest should be the first line of the response. In it there is a response code, “200”, which is used to indicate the request was fulfilled successful by the server. There are more details about response codes in the follow section of this documentation.

The following is another example of an HTTP response:

```
HTTP/1.0 200 Success[CRLF]
Server: NetLink/4D[CRLF]
MIME-Version: 1.0[CRLF]
Content-type: text/html[CRLF]
[CRLF]
<HTML>[CR]
<HEAD>[CR]
  <TITLE>4D Zine</TITLE>[CR]
</HEAD>[CR]
[CR]
```

```

<BODY>[CR]
[CR]
This is my web page with lots of HTML on it...[CR]
[CR]
</BODY>[CR]
[CR]
</HTML>[CR]

```

Again, this sample shows the simplicity of most HTTP responses.

It is important to note that that header of a response is delimited from the content of the response by a double pairing of carriage return/line-feed, four bytes in a row equating to 0x0C0A0C0A. The method ***HTTPcd_Parse_Header_f_Response*** uses this convention for HTTP responses to split an HTTP response into the discrete header and content sections contained within it.

Just for an example, the following is another sample HTTP response:

```

HTTP/1.1 404 File Not Found[CRLF]
Server: WebSTAR/4.4(SSL) ID/73166[CRLF]
Date: Mon, 30 Jul 2001 02:09:20 GMT[CRLF]
Connection: close[CRLF]
MIME-Version: 1.0[CRLF]
Content-type: text/html[CRLF]
Pragma: no-cache[CRLF]
[CRLF]
<HTML>[CR]
<HEAD>[CR]
<TITLE>ERROR: File Not Found</TITLE>[CR]
</HEAD>[CR]
<BODY BGCOLOR="#ffffff">[CR]
<H1>File Not Found</H1><P>[CR]
The file you have requested does not exist on this server.[CR]
</BODY>[CR]
</HTML>

```

This example shows an error which was reported by a server in response to a request. Specifically, a response code of “404” was given, indicating the requested page was not found on the server.

For more details about HTTP responses, read RFC 1945 available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

Response Codes

All HTTP Client Deux request methods return the HTTP response code received from the remote server. Other values for response codes may be returned by the HTTP Client Deux methods; commonly these alternative values indicate an error detected within the code or parameters, and these alternative values will always be negative. Since all valid HTTP response codes are positive, it is a simple matter to distinguish valid values returned from a remote host and values which are do to coding errors.

The following table lists all of the common HTTP response codes which an HTTP host may return:

Response Code	Description
200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

Clearly, just because you receive a positive response code, there is no reason to think that a request was responded to as you might expect. Remote hosts can return responses that are indications of errors of all sorts. Commonly, if a response is fulfilled successfully by a remote host, the response code returned will be “200”.

For more details about HTTP response codes, read RFC 1945 available at:

http://www.deepskytech.com/rfcs/rfc_1945.txt

HTTP Client Deux Constants

There are a few custom constants included with the HTTP Client Deux component package. These constants are grouped into convenient constant groups for easier referencing and organization.

Where appropriate, it is highly recommended that the custom constants included with the HTTP Client Deux component be utilized within your code; this will considerably simplify future feature enhancements to the core code within HTTP Client Deux.

HTTPh Selectors

The HTTPh Selectors constants group is a group of constants provided with the HTTP Client Deux component. This group of constants provides a compact identification mechanism for the common titles of HTTP header values. A different constant defines each of the different values available within the header of an HTTP request and/or response, including a few others for describing different header values when formatted in a particular way.

The following table lists all of the HTTPh Selector constants, their value, and the common name associated with each value:

Constant	Value	Common Name
HTTPh_kfor	1	search arguments
HTTPh_user	2	user name
HTTPh_pass	3	password
HTTPh_Direct	4	direct parameter
HTTPh_addr	5	client address
HTTPh_svm	6	server name
HTTPh_svpt	7	server port
HTTPh_snm	8	script name
HTTPh_ctyp	9	content type
HTTPh_refr	10	referrer
HTTPh_Agent	11	user agent (browser)
HTTPh_Kact	12	action name
HTTPh_Kapt	13	action path
HTTPh_post	14	post arguments
HTTPh_meth	15	HTTP method
HTTPh_Kcip	16	client IP address
HTTPh_Kfrq	17	full request
HTTPh_Kcid	18	transport connection ID
HTTPh_DIRE	19	ID of root folder of host
HTTPh_COOK	20	cookie(s)
HTTPh_CNTN	21	connection

Constant	Value	Common Name
HTTPH_HOST	22	host (domain or IP address)
HTTPH_ACHT	23	accept character set
HTTPH_ACPT	24	accept selector
HTTPH_ACLG	25	accept language
HTTPH_ACEN	26	accept encoding
HTTPH_RURL	27	relative URL
HTTPH_AUTH	28	www authenticate
HTTPH_CTLN	29	content length
HTTPH_CTBD	30	content type boundary
HTTPH_FILE	31	file name requested
HTTPH_FPTH	32	file path requested

Specifically within HTTP Client Deux, only a few of these selector values are used. But, in many other components available from Deep Sky Technologies, Inc., components which function “on top” of HTTP Client Deux, these constants are used.

None of these constants are needed to properly use the HTTP Client Deux component.

Modules

All of the code within the HTTP Client Deux component is organized into modules. Each module is designated by a three (3) to six (6) character module prefix. All of the module prefixes are used within the name of every object within the module (methods names, variable names, semaphore names, etc.). This allows for the easy identification of any object within the HTTP Client Deux component.

Each module contains a set of methods which can be used to easily implement a web server once the HTTP Client Deux component is installed. Method names all begin with the module prefix followed by an underscore (“_”) characters. The remainder of the method name then describes the function of the method.

HTTPcd Module

ENV_Get_HTTPcd_HardName_Long

ENV_Get_HTTPcd_HardName_Long => Long Hard Name

ENV_Get_HTTPcd_HardName_Long
=> Long Hard Name : Text

	Parameter	Type	Description
	<i>Long Hard Name</i>	Text	Full, hard coded name of HTTP Client Deux component including versioning information

The method **ENV_Get_HTTPcd_HardName_Long** returns the full, hard coded name of the HTTP Client Deux component, including versioning information.

Long Hard Name is the full, hard coded name of the HTTP Client Deux component. As of this release, this will always return the value "HTTP_Client_Deux_v1.1.1".

Note: this method was added as of HTTP Client Deux v1.0.1.

ENV_Get_HTTPcd_HardName_Short

ENV_Get_HTTPcd_HardName_Short => Short Hard Name

ENV_Get_HTTPcd_HardName_Short
=> Short Hard Name : Text

	Parameter	Type	Description
	<i>Short Hard Name</i>	Text	Short hard coded name of HTTP Client Deux component; does not include versioning information

The method **ENV_Get_HTTPcd_HardName_Short** returns a shortened, hard coded name of the HTTP Client Deux component.

Short Hard Name is the shortened, hard coded name of the HTTP Client Deux component. As of this release, this will always return the value "HTTP_Client_Deux".

Note: this method was added as of HTTP Client Deux v1.0.1.

HTTPCd_ERROR

HTTPCd_ERROR (*HTTPcd Error Number; Special Error Text; Calling Method Name*)

HTTPCd_ERROR

```
(
    -> HTTPcd Error Number : Longint
    -> Special Error Text : Text
    -> Calling Method Name : Text
)
```

	Parameter	Type	Description
	<i>HTTPcd Error Number</i>	Longint	Internal HTTPcd error number
	<i>Special Error Text</i>	Text	Special text to describe the exact error instance
	<i>Calling Method Name</i>	Text	Name of the method that the error condition occurred in

The method **HTTPCd_ERROR** acts as a callback method from within the HTTPcd module for errors that may occur. Any time an error condition is detected within the HTTPcd module, a call to the method **HTTPCd_ERROR** is made.

The internal *HTTPcd Error Number* is passed to this method as the first parameter. The *Special Error Text* parameter will contain any relevant error text which is specific to the error which occurred. It is not uncommon for the *Special Error Text* value to be empty. The *Calling Method Name* will always contain the name of the HTTPcd method which call the **HTTPCd_ERROR** method.

The **HTTPCd_ERROR** method has been implemented as a source for a consistent interface and/or error tracking mechanism to be available while using the HTTP Client Deux component. This method can be modified to suit the needs of

the database in which the HTTP Client Deux component has been installed.

HTTPcd_Extract_ContentType

HTTPcd_Extract_ContentType (*Referenced HTTP Response*) => *Content Type*

HTTPcd_Extract_ContentType

```
(
    -> Referenced HTTP Response : Pointer
)
=> Content Type : Text
```

	Parameter	Type	Description
	<i>Referenced HTTP Response</i>	Pointer	Referenced BLOB containing HTTP response
	<i>Content Type</i>	Text	HTTP response code extract from referenced response

The method ***HTTPcd_Extract_ContentType*** will extract the content type from a referenced HTTP response.

Referenced HTTP Response is a pointer to a BLOB containing either a full HTTP response or the header of an HTTP response.

Content Type is the content type contained in *Referenced HTTP Response*. The content type of an HTTP response is the contents of the first line in the response header which begins with the string "Content-type:".

HTTPcd_Extract_ResponseCode

HTTPcd_Extract_ResponseCode (*Referenced HTTP Response*) => *HTTP Response Code*

HTTPcd_Extract_ResponseCode

```
(
```

)
 -> *Referenced HTTP Response* : Pointer
 => *HTTP Response Code* : Longint

	Parameter	Type	Description
	<i>Referenced HTTP Response</i>	Pointer	Referenced BLOB containing HTTP response
	<i>HTTP Response Code</i>	Longint	HTTP response code extract from referenced response

The method ***HTTPcd_Extract_ResponseCode*** will extract the HTTP response code from a referenced HTTP response.

Referenced HTTP Response is a pointer to a BLOB containing either a full HTTP response or the header of an HTTP response.

HTTP Response Code is the response code contained in *Referenced HTTP Response*. If the first line of *Referenced HTTP Response* is not a well formed HTTP response or HTTP response header line then *HTTP Response Code* will be set to NULL.

HTTPcd_GET_Simple

HTTPcd_GET_Simple (*URL* ; *Referenced HTTP Response* { ; *Additional Request Header Lines* { ; *Request Timeout* } }) => *HTTP Response Code*

HTTPcd_GET_Simple

(
 -> *URL* : Text
 -> *Referenced HTTP Response* : Pointer
 { -> *Additional Request Header Lines* : Text
 { -> *Request Timeout* : Longint } }
)
 => *HTTP Response Code* : Longint

	Parameter	Type	Description
	<i>URL</i>	Text	Fully formatted URL to retrieve

	Parameter	Type	Description
	<i>Referenced HTTP Response</i>	Pointer	Pointer to a BLOB to contain the complete HTTP response
	<i>Additional Request Header Lines</i>	Text	Additional information to be put into the HTTP request header
	<i>Request Timeout</i>	Longint	Request timeout, in seconds
	<i>HTTP Response Code</i>	Longint	HTTP response code returned

The method ***HTTPcd_GET_Simple*** will run a simple HTTP GET request for a specified URL and return the response from the remote server.

The header used for the request is contained in the Affix HTTP Client Deux document, resource of type 'TEXT', resource ID # 29000. The header is a PTEXT value which has placeholders for the host name and path with parameters.

URL is the fully formatted URL to retrieve. This method does not support usernames and passwords in the HTTP request.

Referenced HTTP Response is a pointer to a BLOB which will be set to the complete response from the remote server.

Additional Request Header Lines is additional information, such as cookie information or custom header fields, to be included in the HTTP request header. *Additional Request Header Lines* must be prepended with **[CRLF]**, and each line separated by **[CRLF]**.

Request Timeout is the time in seconds to allow for inactivity before aborting the request and returning to the calling method.

HTTP Response Code is the response code contained in *Referenced HTTP Response*. If the first line of *Referenced HTTP Response* is not a well formed HTTP response or HTTP response header line then *HTTP Response Code* will be set to NULL. If there is an error in this method before retrieving the remote *URL*, then *HTTP Response Code* will be set to negative one (-1).

HTTPcd_HEAD_Simple

HTTPcd_HEAD_Simple (*URL* ; *Referenced HTTP Response* { ; *Additional Request Header Lines* { ; *Request Timeout* } }) => *HTTP Response Code*

HTTPcd_HEAD_Simple

```
(
    -> URL : Text
    -> Referenced HTTP Response : Pointer
    { -> Additional Request Header Lines : Text
    { -> Request Timeout : Longint } }
)
=> HTTP Response Code : Longint
```

	Parameter	Type	Description
	<i>URL</i>	Text	Fully formatted URL to retrieve
	<i>Referenced HTTP Response</i>	Text	Pointer to a BLOB to contain the complete HTTP response
	<i>Additional Request Header Lines</i>	Text	Additional information to be put into the HTTP request header
	<i>Request Timeout</i>	Longint	Request timeout, in seconds
	<i>HTTP Response Code</i>	Longint	HTTP response code returned

The method **HTTPcd_HEAD_Simple** will run a simple HTTP HEAD request for a specified URL and return the response from the remote server.

The header used for the request is contained in the Affix HTTP Client Deux document, resource of type 'TEXT', resource ID # 29002. The header is a PTEXT value which has placeholders for the host name and path with parameters.

URL is the fully formatted URL to retrieve. This method does not support usernames and passwords in the HTTP request.

Referenced HTTP Response is a pointer to a BLOB which will be set to the complete response from the remote server.

Additional Request Header Lines is additional information, such as cookie information or custom header fields, to be included in the HTTP request header. *Additional Request Header Lines* must be prepended with **[CRLF]**, and each line separated by **[CRLF]**.

Request Timeout is the time in seconds to allow for inactivity before aborting the request and returning to the calling method.

HTTP Response Code is the response code contained in *Referenced HTTP Response*. If the first line of *Referenced HTTP Response* is not a well formed HTTP response or HTTP response header line then *HTTP Response Code* will be set to NULL. If there is an error in this method before retrieving the remote *URL*, then *HTTP Response Code* will be set to negative one (-1).

HTTPcd_Parse_Header_f_Response

HTTPcd_Parse_Header_f_Response (*Referenced Full HTTP Response* ; *Referenced HTTP Header*)

HTTPcd_Parse_Header_f_Response

(
 -> *Referenced Full HTTP Response* : Pointer
 -> *Referenced HTTP Header* : Pointer
)

	Parameter	Type	Description
	<i>Referenced Full HTTP Response</i>	Pointer	Referenced BLOB containing complete HTTP response
	<i>Referenced HTTP Header</i>	Pointer	Referenced BLOB to contain HTTP header within Referenced Full HTTP Response

The method ***HTTPcd_Parse_Header_f_Response*** will parse a complete HTTP response into discrete header and content areas.

Referenced Full HTTP Response is a pointer to a BLOB containing a complete HTTP response. After this method runs, this value will be only the HTTP response content.

Referenced HTTP Header is a pointer to a BLOB which will be set to the HTTP header parsed from *Referenced Full HTTP Response*.

HTTPcd_POST_Simple

HTTPcd_POST_Simple (*URL ; Referenced Names ; Referenced Values ; Referenced HTTP Response { ; Additional Request Header Lines { ; Request Timeout } })*
 => HTTP Response Code

HTTPcd_POST_Simple

```
(
  -> URL : Text
  -> Referenced Names : Pointer
  -> Referenced Values : Pointer
  -> Referenced HTTP Response : Pointer
  { -> Additional Request Header Lines : Text
    { -> Request Timeout : Longint } }
)
```

=> HTTP Response Code : Longint

	Parameter	Type	Description
	<i>URL</i>	Text	Fully formatted URL to retrieve
	<i>Referenced Names</i>	Pointer	Referenced text array containing POST names
	<i>Referenced Values</i>	Pointer	Referenced text array containing POST values
	<i>Referenced HTTP Response</i>	Pointer	Pointer to a BLOB to contain the complete HTTP response
	<i>Additional Request Header Lines</i>	Text	Additional information to be put into the HTTP request header
	<i>Request Timeout</i>	Longint	Request timeout, in seconds
	<i>HTTP Response Code</i>	Longint	HTTP response code returned

The method **HTTPcd_POST_Simple** will run a simple HTTP POST request for a specified URL and return the response from the remote server.

The header used for the request is contained in the Affix HTTP Client Deux document, resource of type 'TEXT', resource ID # 29001. The header is a PTEXT value which has placeholders for the host name, path with parameters, and content length.

URL is the fully formatted URL to retrieve. This method does not support usernames and passwords in the HTTP request.

Referenced Names is a pointer to a text array containing the names of search arguments to post to the remote HTTP server. Values in the referenced array are paired with values of the same index in *Referenced Values*. All values in *Referenced Names* will be URL encoded and formatted for a proper HTTP POST request.

Referenced Values is a pointer to a text array containing the values of search arguments to post to the remote HTTP server. Values in the referenced array are paired with values of the same index in *Referenced Names*. All values in *Referenced Values* will be URL encoded and formatted for a proper HTTP POST request.

Referenced HTTP Response is a pointer to a BLOB which will be set to the complete response from the remote server.

Additional Request Header Lines is additional information, such as cookie information or custom header fields, to be included in the HTTP request header. *Additional Request Header Lines* must be prepended with **[CRLF]**, and each line separated by **[CRLF]**.

Request Timeout is the time in seconds to allow for inactivity before aborting the request and returning to the calling method.

HTTP Response Code is the response code contained in *Referenced HTTP Response*. If the first line of *Referenced HTTP Response* is not a well formed HTTP response or HTTP response header line then *HTTP Response Code* will be set to NULL. If there is an error in this method before retrieving the remote *URL*, then *HTTP Response Code* will be set to negative one (-1).

HTTPcd_Retrieve_Object

HTTPcd_Retrieve_Object (*Host Name* ; *Referenced Request Header* ; *Referenced Request Content* ; *Remote Port* ; *Local Port* ; *Local IP Address* ; *Protocol* ; *Timeout* ; *Referenced HTTP Response*) => *HTTP Response Code*

HTTPcd_Retrieve_Object

```
(
-> Host Name : String[80]
-> Referenced Request Header : Pointer
-> Referenced Request Content : Pointer
-> Remote Port : Longint
-> Local Port : Longint
-> Local IP Address : Longint
-> Protocol : Longint
-> Timeout : Longint
-> Referenced HTTP Response : Pointer
)
=> HTTP Response Code : Longint
```

	Parameter	Type	Description
	<i>Host Name</i>	String[80]	Host name of remote HTTP server
	<i>Referenced Request Header</i>	Pointer	Referenced BLOB containing complete HTTP request header
	<i>Referenced Request Content</i>	Pointer	Referenced BLOB containing complete HTTP request content
	<i>Remote Port</i>	Longint	Remote port to connect to
	<i>Local Port</i>	Longint	Local port to connect through
	<i>Local IP Address</i>	Longint	Local IP address to connect from
	<i>Protocol</i>	Longint	Protocol to use for connection, HTTP or HTTPS
	<i>Timeout</i>	Longint	Timeout value for HTTP session in seconds
	<i>Referenced HTTP Response</i>	Pointer	Pointer to a BLOB to contain the complete HTTP response
	<i>HTTP Response Code</i>	Longint	HTTP response code returned

The method **HTTPcd_Retrieve_Object** gives you the most control over sending an HTTP request. It also is the most complex, because you must build the HTTP header and request content yourself.

Host Name is the full domain name or IP address of the remote host to connect to.

Referenced Request Header is a pointer to a BLOB containing the complete HTTP request header to send to *Host Name*.

Referenced Request Content is a pointer to a BLOB containing the complete HTTP request content to send to *Host Name*. This value can be NULL if there is no HTTP request content to send (e.g. HEAD or GET requests).

Remote Port is the remote port to connect to.

Local Port is the local port to connect through. This value is not used if Internet Commands is the current TCP plugin being used by the TCP Deux component.

Local IP Address is the local IP address on the local host to connect from. If this value is NULL then the primary IP of the local host will be used. This value is not used if Internet Commands is the current TCP plugin being used by the TCP Deux component.

Protocol is session protocol value to indicate the protocol to be used for retrieving the object. In general, the only options here which make sense are HTTP and HTTPS sessions. *Protocol* values are available from the TCPd_Protocols constant group in the TCP Deux 4D component package.

Timeout is the number of second to use as the timeout value for the HTTP connection being made.

Referenced HTTP Response is a pointer to a BLOB which will be set to the complete response from the remote server.

HTTP Response Code is the response code contained in *Referenced HTTP Response*. If the first line of *Referenced HTTP Response* is not a well formed HTTP response or HTTP response header line then *HTTP Response Code* will be set to NULL. If there is an error in this method before retrieving the remote *URL*, then *HTTP Response Code* will be set to negative one (-1).

INIT_HTTPcd

INIT_HTTPcd

INIT_HTTPcd

	Parameter	Type	Description
	<i>none</i>	n/a	n/a

The method ***INIT_HTTPcd*** initialises the HTTP Client Deux component. A single call to this method should be made early in the **On Startup** database method in your 4D application. Make certain the call to this method follows the initialization call to the BASH and TCP Deux components but before any other calls to the HTTP Client Deux component package.

Version History

The following is a brief version history of the HTTP Client Deux component. It details release notes, bug fixes, and changes for each version publicly available.

HTTP Client Deux v1.1.1

released Friday, December 6th, 2002

Changes:

Corrected bug in Windows Affix document for 4D v6.8.x in which the '4BNX' resource was conflicting with the ID in the Affix BASH document (it was set incorrect in HTTPcd).

Modified all request templates to now have an Accept: value of */*.

Corrected name of '4BNX' resource on Windows to no longer be Affix BASH.

Corrected resource ID conflicts within platforms of BASH Affix documents; made certain all affix documents coincide with the new plugin ID for HTTPcd (29610).

Made Affix document on Mac under 4D v67x a true plugin stub.

Corrected issue with bug in 4D v68x on Windows in which resource names can cause conflicts.

Added full timeout control and respect for response reception in method **HTTPcd_Retrieve_Object**.

Modified method **HTTPcd_Parse_Header_f_Response** to use case sensitive search to be faster.

Added support for returning from **HTTPcd_Retrieve_Object** when a properly formatted HTTP response has been fully received, regardless of what else might be on the stream.

HTTP Client Deux v1.1.0

released Monday, April 15th, 2002

Changes:

Added support for authorization username and password across all simple request methods.

Added NVP parameter to all simple headers in Affix document to support additional request header lines, as needed.

Added optional parameter to all simple request methods for including extra header lines in request.

Added optional parameter to all simple request methods for request timeout in seconds.

Increased default timeout in all simple request routines from 30 to 60 seconds.

Changed name of Mac affix document from “Affix_HTTPc_Deux” to “Affix_HTTPc_Deux.4DX”.

Updated routine for finding correct Affix document for this component. Now uses generic routines in BASH for handling under 4D v6.8.x and MacOS X.

Added support for new 4D plugin architecture under 4D v6.8.x in which plugins can be located next to the current application in any environment.

Created a carbon affix document entitled “Affix_HTTPc_Deux.4CX”.

HTTP Client Deux v1.0.2

released 20020105

Changes:

Fixed bug in ***HTTPcd_Extract_ResponseCode*** where if descriptive text contained "e" (i.e. "HTTP/1.1 302 Object Moved"), the response code returned would be zero.

Changed name of affix document on Windows from "Afx_HTTPcd" to "Affix_HTTPc_Deux".

HTTP Client Deux v1.0.1

released 20010906

Changes:

Modified the initialization routines so that it checks if the TCP Deux version is greater than or equal to 1.0.1; if not, then the ***HTTPcd_ERROR*** method is called and the component is not initialized.

Fixed a bug in the routine ***HTTPcd_Extract_ContentType*** where the content-type was not being returned properly; it was looking for "Content-types:" and not "Content-type:".

Added the methods ***ENV_Get_HTTPcd_HardName_Long*** and ***ENV_Get_HTTPcd_HardName_Short***.

HTTP Client Deux v1.0.0

released 20010728

Changes:

First full release of the component. No changes made since v1.0.0b02.

HTTP Client Deux v1.0.0b02

released 20010628

Changes:

Added the method ***HTTPcd_Extract_ContentType***.

Fixed a bug in the method ***HTTPcd_Retrieve_Object*** in which when sending HTTP content in the request when using IC the request sending would fail and generate an 2903902 error in ***HTTPcd_ERROR*** (thanks to Marc Conner for identifying this bug).

Fixed a bug in which the local port setting would cause the streams to fail to be opened if the port was currently in use on the local machine; made it so that the plugin chooses automatically the local port for all simple API commands in this component.

Fixed problem in the method ***HTTPcd_Retrieve_Object*** in which the server closing the TCP stream could possibly generate an error in the HTTPcd component even though the complete response was successfully received; now, the method will not return an error in this case though the initial asynchronous receive over TCP will have a catch, and notification, if an error occurs, though subsequent TCP receives will not generate an error.

HTTP Client Deux v1.0.0b01

released 20010605

Changes:

First public release of the component.

Errors

The listing of error codes and conditions is obviously a continuously updated process. With practically every new version of HTTP Client Deux, the error codes and conditions can and do change. Though it has been a long time coming, we are now documenting much of the error conditions that can occur in HTTP Client Deux.

Different methods in the HTTP Client Deux component can generate errors when used incorrectly or when used under the wrong circumstances. When an error condition is encountered, the methods within the HTTP Client Deux component will call the “HTTPcd_ERROR” method. The “HTTPcd_ERROR” method is documented above. The first parameter sent to the “HTTPcd_ERROR” method is the error code identifying the unique error condition that occurred.

With future versions of the HTTP Client Deux component (and other components to come), the management and handling of error conditions will become much better documented, much clearer to understand, and easier to handle in your code. For now, though, reading this manual thoroughly is the best single source of understanding for the error conditions that can arise in using the HTTP Client Deux component.

Error Codes

When the “HTTPcd_ERROR” method is called in the HTTP Client Deux component, the first parameter is always an error code. This error code is always a 7 digit integer indicating the unique error condition which was detected in the code.

These 7 digit numbers actually consist of two pieces for uniquely identifying the error code and condition. The first five digits is an internal code for the module which an error occurred within. The last two digits identifies the unique error condition from within a module that has been detected.

The following is a quick listing of all of the error codes, grouped by the module which the error codes are from. Each error code includes the textual description of the error condition.

HTTPcd: 29039 series

2903901	Failed to open TCP stream
2903902	Failed to send request content
2903903	Failed to receive response from host
2903904	Failed to send request header
2903905	Failed to establish connection to host
2903906	TCP Deux component is too old to work with this version of HTTP Client Deux
2903907	Hard long name of Affix document is not correct length

Method Listing

The following is a listing of all of the methods within the HTTP Client Deux component (COMPILER methods are not included in this listing), including all private methods which are not directly available in the API when the HTTP Client Deux component is installed. Following each method is a list of the error codes which each method can generate.

```

ENV_Get_HTTPcd_HardName_Long
ENV_Get_HTTPcd_HardName_Short
ENV_Get_HTTPcd_RF_FullPath
HTTPcd_ERROR
HTTPcd_Extract_ContentType
HTTPcd_Extract_ResponseCode
HTTPcd_GET_Simple
HTTPcd_HEAD_Simple
HTTPcd_Parse_Header_f_Response
HTTPcd_POST_Simple
HTTPcd_qi_Response_Complete
HTTPcd_Retrieve_Object
    2903901
    2903902
    2903903
    2903904
    2903905
INIT_HTTPcd
    2903906
    2903907
RES_Open_HTTPcd

```