Seminole Developer's Guide

Seminole Developer's Guide

Table of Contents

List of Figures

List of Tables

List of Examples

Introduction

This manual is intended for anyone who will be including Seminole in another system, *porting* Seminole to a new platform or *RTOS*, or extending Seminole in some way. This guide is intended as a detailed reference rather than a tutorial. Beginners are encouraged to read the *Getting Started Guide* and work through the examples before taking on more complex projects.

Once familiar with the basics this reference guide should be used when writing code that uses the Seminole *API* to its fullest. Each class is documented with a general summary followed by its public interfaces in excruciating detail. This document is the best way to understand the Seminole *API* when there is no desire to "look under the hood."

Should this document prove insufficient our support department will be happy to help you with further questions. Also, comments and corrections concerning this documentation are welcome, and may be sent via Internet mail to `<support@gladesoft.com>`.

Chapter 1. Overview

About Seminole

Seminole is an embedded webserver toolkit. It is not designed to run as a standalone webserver although it is capable of doing so. Instead, Seminole is designed to be embedded into other software. Such software is typically the firmware of an embedded system although application software can embed Seminole as well. It is written using a subset of C++ with an eye towards portability as well as modularity. The services of the underlying operating system are abstracted with a few simple functions and `typedef`s.

Because Seminole is designed to be embedded in other software it has a small code footprint (especially for embedded systems) and a small heap appetite. Another difference between traditional webservers and Seminole is the interface to external code. Seminole allows application code to execute within Seminole rather than an external process. This is especially important since many real-time operating systems have no concept of a process. This also allows easier application programming with high-level C++ interfaces rather than traditional “gateway” applications.

Most emebedded webservers are not used for serving static content. Although embedded devices can have on-line user manuals via HTTP the primary purpose of web-enabling a device is to provide a user interface. To that end, unlike a traditional web server, Seminole does not require a file system and the core server is not “file centric”. Instead, objects derived from the `HttpdHandler` class lay claim to various portions of the URL space. When a request comes in Seminole finds the appropriate handler and dispatches the request to it. A default handler class, `HttpdFileHandler`, is provided for serving up files from a filesystem abstraction. In addition, both a native *ROM* filesystem and an interface to a POSIX file system are included for more traditional web serving tasks. This default framework is suitable for testing and development purposes on POSIX-oriented systems such as UNIX® or for production use on embedded operating systems providing such a file system interface.

When Seminole is started, an instance of the class `Httpd` is created for each configured port. For each possible URL prefix, a derivative of the abstract base class `HttpdHandler` is inserted into a list within the `Httpd` instance. Therefore, `Httpd` is a container for one or more `HttpdHandler` instances.

When an incoming request is made, the Seminole instance will create a new *thread* (depending on the services provided by the host platform, “task” or “job” may be the appropriate concept) to process the request. The request handler will then create a new instance of `HttpdRequest`.

`HttpdRequest` will read in the HTTP request and *MIME* headers and perform some basic parsing. Most of the request processing centers around calling methods provided by the `HttpdRequest` class.

The handler code in `Httpd` will then call each of the registered `HttpdHandler`s in series giving them the URL and checking to see if they want it. They can either accept the request -- in which case the request object is destroyed after the selected `HttpdHandler` processes the request; or they can reject the request, in which case, the next handler in the chain is called. If no more handlers are present default error processing is performed.

Each `HttpdHandler` is associated with a “prefix string”. This string represents the first `N` characters that a URI must begin with for that handler to be considered “responsible” for that request. The `Httpd` class maintains a linked list sorted by order of prefix length.

Example 1.1. Handler Mapping Precedence

`/web/dynamic/`	Handler 1

`/cgi-bin/`	Handler 2
`/web/`	Handler 3
`/`	Handler 4

Requests will always try the most specific URL prefix first. So a request for `/web/dynamic/foo.html` would be passed to handler 1 in this example, whereas a request for `/web/foo.html` would be passed to handler 3.

Performance

Seminole is designed to scale well. On one end of the performance spectrum it can be configured to consume few resources with adequate performance. On the other end of the spectrum Seminole can be configured to handle a very high volume of requests. Getting the best performance from Seminole does require tuning parameters in both Seminole and the target platform.

Seminole requires few amenities from the target platform. In fact even threads are not required — requests will simply be processed serially. However this configuration should only be used for targets with the most limited resources where performance is not an issue. This is because without threads the persistent connection feature of HTTP can't be used.

When Seminole is using threads it is agnostic to how those threads are managed. On some platforms creating a thread is a very efficient operation. On other platforms thread creation is an expensive and time-consuming process. For these platforms a pool of worker threads can be created in advance and wait for requests. Most of the provided portability layers offer thread pooling as an option.

Regardless of how threads are managed (pooling or on-the-fly creation) the portability layer also needs to limit the maximum number of threads Seminole uses. When the number of active threads exceeds the maximum the portability layer can either return immediate failure or block waiting for a period of time for a thread to free up. The former approach is best for small systems that have at most a handful of users performing requests. The latter approach is more appropriate if heavy activity is expected.

The semantics of sleeping when the thread pool is empty helps keep load managable because the acceptor thread will be blocked while the worker threads finish up their processing. The acceptor thread is responsible for handing off new requests to worker threads. If a free thread is not available after blocking then the portability layer can return failure and the new request will be discarded.

This works well when worker threads are quickly completing their work and exiting (or returning to the free thread pool). Persistent connections can cause idle threads to wait rather than perform useful work. This effect is amplified if the timeout value for persistent connections is greatly increased. Seminole includes an optional feature called “overload protection” that attempts to release threads occupied with idle persistent connections.

Overload protection is invoked when the portability layer returns failure when spawning a thread. In this case the oldest thread is signaled to abort its wait for a request. The thread waiting for the request will be in the `HttpdSocket::Gets` method. Overload protection calls the `HttpdSocket::AbortGets` method on the socket. This method will cause `Gets` to return immediately rather than wait for the timeout. If `AbortGets` returns failure the next oldest thread is selected for reclaimation. This process is repeated a finite number of times until the thread can be spawned.

Chapter 2. Core *API* Reference

Using the *API*

The Seminole *API* is defined in several header files. The main header file, `seminole.h`, defines almost all of the “core” *API*. Advanced features are contained in their own header files which should be included after `seminole.h`.

The Seminole source tree contains some header files that are not public and should not be included by applications. The Seminole build system knows which header files are public and which are not. The public header files are copied to the `built/`*`PORT`*`/include` directory. This is the location where Seminole *applications* should include their header files from.

All of the Seminole names begin with some permutation of `Httpd`. This is to avoid clashes with application code. This may be confusing when examining the Seminole *API* because many of the supporting classes can be used without any dependencies on the HTTP protocol. In fact, the *API* was designed so that some of the tools can be used without the webserver class (`Httpd`) at all.

The *API* is heavily object-oriented. It is important that programmers understand the basic concepts of object oriented programming: encapsulation (abstraction), inheritance, and polymorphism. Each of the classes has a particular useage model. Sometimes this model is different from other classes in the *API* This inconsistency is typically due to some efficiency constraint (e.g. code size). However every attempt has been made to keep the *API* as consistent and easy to program to.

Seminole Constants, Macros, and Types

Introduction

Seminole defines a small number of custom data types for internal purposes. Most of these are used in public interfaces, and thus implementors should be aware of them. This chapter documents such types.

All definitions are portable (i.e. identical across target abstraction layer implementations) unless otherwise noted.

Constants

HTTPD_U8_BYTES

This constant is the number of bytes required to hold an 8-bit unsigned integer.

HTTPD_U16_BYTES

This constant is the number of bytes required to hold a 16-bit unsigned integer. It is almost universally `1` except for very specialized environments.

HTTPD_U32_BYTES

This constant is the number of bytes required to hold a 32-bit unsigned integer.

HTTPD_SESSION_KEY_LEN

The length (in characters) of a session identifier. This locally unique identifier is used to identify sessions with incoming requests. This value is a function of the `SESSION_NONCE_LEN` build parameter. In general it is sufficiently small that buffers of this size can be allocated as local variables.

Types

HttpdUint16

```
typedef unsigned short HttpdUint16;
```

This type is normally defined by the portability layer. It should be a 16-bit unsigned integer on the target platform. Shown above is a typical definition for most architectures.

HttpdUint32

```
typedef unsigned long HttpdUint32;
```

This type is normally defined by the portability layer. It should be a 32-bit unsigned integer on the target platform. Shown above is a typical definition for most architectures.

HttpdBitWord

```
typedef unsigned int HttpdBitWord;
```

This type represents the unit of access by the `HttpdBitSet`. It is typically defined as an unsigned int. This is the most efficient word size for the machine to access.

HttpdPair

```
struct HttpdPair
{
  const char   *mpKey;
  const char   *mpValue;
};
```

This struct is used to store name/value pairs, such as HTTP or *MIME* headers. HttpdUtilities::Lookup provides a method to search a sorted series of HttpdPairs.

HttpdIpv4Address

```
typedef HttpdUint32 HttpdIpv4Address;
```

Provides a binary representation of an Internet Protocol (IP) V4 address. This type definition may vary from one target platform to another. The POSIX abstraction layer definition is shown here. If the definition were of a more complex type, such as a structure then appropriate copy and comparison operators must be provided by the portability layer.

HttpdIpAddress

```
typedef HttpdIpv4Address HttpdIpAddress;
```

Provides a binary representation of an Internet Protocol (IP) address.

Note

This type definition may vary from one target platform to another. The POSIX abstraction layer definition is shown here when INC_IPV6_SUPPORT is disabled. If the definition were of a more complex type, such as a structure then the portability layer should define the preprocessor symbol HTTPD_HAVE_BULKY_SOCKET_ADDRESSES to a non-zero value and instead define a class named `HttpdIpAddressObject` derived from the provided HttpdIpAddressBase.

HttpdIpPort

```
typedef HttpdUint16 HttpdIpPort;
```

Provides a binary representation of a TCP/IP port number.

Note

This type definition may vary from one target platform to another. The POSIX abstraction layer definition is shown here.

HttpdSocketWaitHandle

```
typedef … HttpdSocketWaitHandle;
```

This type is defined by the portability layer. It abstracts an optional object that may be waited for alongside socket events. This capability is only used if the portability layer defines `HAVE_SOCK_WAIT` to `1`.

Code that uses this type other than simply passing it along is inherently nonportable. Different systems may use wildly different definitions to define this type. For example POSIX systems use a file descriptor here while Win32 uses a WSAEVENT handle.

HttpdTransport

```
struct HttpdTransport
{
  const char             *mpTransportName;
  HttpdSocketInterface *(*mpFactory)();
  int                   (*mpInitialize)();
  const char             *mpUriScheme;
  HttpdIpPort             mPort;
};
```

This type describes a particular *transport* associated with a `HttpdSocket` object. This type is only defined if the INC_MULTIPLE_TRANSPORTS option is enabled.

HttpdProtocolVersion

```
typedef HttpdUint16 HttpdProtocolVersion;
```

Used to encode HTTP version specifications, for purposes of comparison and matching appropriate responses to requests made via a particular version. Predefined constants exist for the HTTP versions in use at the time of this writing, as described in Table 2.1, “Predefined HttpdProtocolVersion Constants”. Version comparisons can be made through use of the standard numerical comparison operators. HttpdUtilities::ParseHttpVersion can be used to generate a HttpdProtocolVersion representation of an ASCII version string.

Table 2.1. Predefined HttpdProtocolVersion Constants

Constant Name	HTTP Version
`HTTPvUnknown`	Unknown
`HTTPv09`	HTTP/0.9
`HTTPv10`	HTTP/1.0
`HTTPv11`	HTTP/1.1

HttpdAuthSchemes

```
typedef enum
{
  Basic,
  Digest, // Only present if HTTPD_INC_DIGEST_AUTH is non-zero.

  End
} HttpdAuthSchemes;
```

This type identifies one of the supported authentication schemes. The enumeration `End` is used to terminate lists of authentication schemes.

HttpdUnicodeCharacter

```
typedef HttpdUint32 HttpdUnicodeCharacter;
```

Used to represent a Unicode character. Unicode is a 21-bit character coding scheme. As such characters are represented natively by a 32-bit unsigned value. Normally Unicode characters are encoded using a more compact scheme. For example UTF-8 is a variable length scheme that encodes Unicode characters that is optimized for compactly representing ASCII characters.

HttpdMD5Digest

```
typedef HttpdUint8 HttpdMD5Digest[16];
```

This type holds an MD5 digest. It is always 16 unsigned octets in size.

HttpdSHA1Digest

```
typedef HttpdUint8 HttpdSHA1Digest[20];
```

This type holds a SHA-1 digest. It is always 20 unsigned octets in size.

HttpdClientCounter

```
typedef unsigned char HttpdClientCounter;
```

This type is used as an event counter by the HTTP client component. Variables of this type are used to keep retry counters and limits during fetch operations.

Macros

Seminole uses the C++ preprocessor when it makes the code clearer and easier to maintain. In some cases there are some ugly preprocessor tricks used to optimize core routines but these are always kept localized to the area being optimized and are never visible in the Seminole *API* in any way.

HTTPD_NUMELEM

```
#define HTTPD_NUMELEM(a) …
```

This macro computes the number of elements in an array `a`.

Caution

The value of this macro is only correct if the compiler knows the size of the array before the macro is invoked. For example declarations such as:

```
extern int array[]; // Unknown size.
```

will not work.

HTTPD_BASED_PTR

```
#define HTTPD_BASED_PTR(p, t, o) …
```

This macro will bias a pointer *`p`* by the the offset *`o`* bytes and return a pointer to type *`t`*. This macro is a convenience macro when adding a byte offset to a pointer. Using this macro helps avoid errors where the type is not properly casted to a byte pointer. Furthermore this macro also helps document the intent of code better than a pile of casts.

httpd_often

```
#define httpd_often(x) …
```

Some compilers, notably GCC can be given hints about conditionals to produce better code. Specifically a compiler can produce more optimal code if it knows that the body of an if-statement is only executed in the event of an error, for example.

For compilers that support these hints this macro indicates that the conditional branch is frequently taken. The entire condition of the if-statement should be substituted for `x`.

Seminole uses this macro (and httpd_rarely) extensively to help improve code generation. There is nothing preventing code written against Seminole's *API* from using these macros as well.

httpd_rarely

```
#define httpd_rarely(x) …
```

This macro, like its counterpart httpd_often is used to give conditional hints to the compiler during code generation. This macro indicates that a condition is infrequently true. This is especially common for error handling code.

`HttpdUtilities` Reference

Introduction

`HttpdUtilities` is a *static class* that is used to hold various helper routines that the Seminole core depends on. All of the methods and data members of this class are static; there is no need to ever instantiate this class.

Most of these routines may be called by your handlers as well, so it is important that they be well understood.

Public Methods

`StrLimitCopy`

`bool` **`HttpdUtilities::StrLimitCopy`** `(char *`*`p_dest`*`, const char *`*`p_src`*`, size_t` *`maxlen`*`);`

This routine copies the string pointed to by *`p_src`* to the buffer pointed to by *`p_dest`*. If the source string plus the zero-termination byte exceed the length specified by *`maxlen`* then the copy is a properly null-terminated truncation of the original.

This routine returns true if the copy did not perform any truncation or false if there was truncation.

This routine is similar to the standard library routine `strncpy` with the exception that it always properly null-terminates the resulting string and returns an indication of truncation.

`StrVCat`

`char *`**`HttpdUtilities::StrVCat`** `(const char *`*`p_first`*`, …);`

This routine will concatenate a NULL terminated list of strings and return a pointer to the resultant string in storage obtained from HttpdOpSys::Malloc.

It is important to terminate the list with *`(char char *)0`*, not NULL, because some CPU architectures have different NULL pointer representations for different types and the compiler does not know the type of the pointer because it is a variable argument list.

`SaveString`

`char *`**`HttpdUtilities::SaveString`** `(const char *`*`p_original`*`);`

This method makes a copy of a string (*`p_original`*). It is identical in effect to:

```
StrVCat(p_original, (char *)0)
```

It is designed to save code space where `StrVCat` would require that two arguments be passed.

`SaveString` returns a pointer to a copy of the string in storage obtained from HttpdOpSys::Malloc. On error, `NULL` is returned.

StrChop

`char *`**`HttpdUtilities::StrChop`**` (char *&`*`p_string`*`);`

This routine will tokenize a white-space delimited string. A pointer to the next token (within *`p_string`*) is returned by the function. In addition, *`p_string`* is updated to point to the next token so that successive calls to `StrChop` will tokenize an entire input string. An empty string is returned when no more tokens are available.

MatchPattern

`bool `**`HttpdUtilities::MatchPattern`**` (const char *`*`p_pattern`*`, const char *`*`p_string`*`, unsigned short `*`depth`*` = HTTPD_PMATCH_MAX_RECURSION);`

This function determines if *`p_string`* matches a generic pattern, *`p_pattern`*. If the string matches then true is returned if the pattern does not match then false is returned.

Patterns consist of the `?` and `*` meta-characters. A `?` matches any single character and a `*` matches zero or more characters. For example the string “Seminole Webserver” is matched by the pattern “Sem*ver”. Characters that are not one of the special meta-characters or are quoted are called non-meta characters and must match themselves in the string.

If the INC_CHARCLASS_PATTERN_MATCH option is enabled then character classes are supported. For example `[abc]` would match any of those three characters. Additionally a range of characters can be provided, such as `[a-c]` which is identical to the `[abc]` character class.

To match metacharacters any character can be escaped using a backslash (\).

The *`depth`* controls the available recursion depth. In some cases this method may need to recursively call itself. To prevent stack overflows the *`depth`* parameter is decremented before each call to `MatchPattern`. If it reaches `0` then the match is considered a failure and false is returned.

The *`depth`* has a default value of `HTTPD_PMATCH_MAX_RECURSION` so it does not have to be specified in calls to this method unless some specific limit is desired for a particular call site.

StringIsEmpty

`bool `**`HttpdUtilities::StringIsEmpty`**` (const char *`*`p_string`*`);`

This routine determines if the string *`p_string`* is composed of only whitespace chatacters.

StrCmp

`int `**`HttpdUtilities::StrCmp`**` (const char *`*`p_a`*`, const char *`*`p_b`*`);`

Carry out a case-sensitive lexicographic comparison between *`p_a`* and *`p_b`*. Returns 0 if they are equal, less than 0 if *`p_a`* is lexicographically less than *`p_b`*, or greater than 0 if *`p_a`* is lexicographically greater than *`p_b`*.

Note

This method is identical to the `strcmp` method and is used to prevent using the address of `strcmp` directly. In some environments (due to calling convention) this can perturb the runtime library or compiler. In general a pointer to `strcmp` is only needed for the `HttpdUtilities::Lookup` method.

StrCmpi

`int` **`HttpdUtilities::StrCmpi`** `(const char *`*`p_a`*`, const char *`*`p_b`*`);`

Carry out a case-insensitive lexicographic comparison between *p_a* and *p_b*. Returns 0 if they are equal, less than 0 if *p_a* is lexicographically less than *p_b*, or greater than 0 if *p_a* is lexicographically greater than *p_b*.

StrnCmpi

`int` **`HttpdUtilities::StrnCmpi`** `(const char *`*`p_a`*`, const char *`*`p_b`*`, size_t` *`len`*`);`

Carry out a case-insensitive lexicographic comparison between *p_a* and *p_b*. A maximum of *len* characters are compared. Returns 0 if they are equal, less than 0 if *p_a* is lexicographically less than *p_b*, or greater than 0 if *p_a* is lexicographically greater than *p_b*.

UriStringCompare

`int` **`HttpdUtilities::UriStringCompare`** `(const char *`*`p_encoded`*`, const char *`*`p_string`*`);`

Carry out a case-sensitive lexicographic comparison between *p_encoded* and *p_string*. Characters that are URL-escaped in *p_encoded* match against their unencoded counterparts in *p_string*. This method returns 0 if they are equal, less than 0 if *p_encoded* is lexicographically less than *p_string*, or greater than 0 if *p_encoded* is lexicographically greater than *p_string*.

If the escapes are malformed in *p_encoded* then INT_MIN is returned.

SkipWhitespace

`char *`**`HttpdUtilities::SkipWhitespace`** `(char *`*`p_string`*`);`

This routine skips leading whitespace and returns a pointer to either the end of string (pointing at the null terminator byte) or the first non-whitespace character.

There is also an identical version of this method that works on constant strings.

SkipNonWhitespace

`char *`**`HttpdUtilities::SkipNonWhitespace`** `(char *`*`p_string`*`);`

This routine skips leading non-whitespace characters and returns a pointer to either the end of string (pointing at the null terminator byte) or the first whitespace character in the string.

There is also an identical version of this method that works on constant strings.

UrlPrefixMatches

`char *`**`HttpdUtilities::UrlPrefixMatches`** `(char *`*`p_string`*`, const char *`*`p_prefix`*`);`

This routine determines if *p_prefix* is present in the URL *p_string*. Percent-escaped characters in *p_string* match against their unescaped versions in *p_prefix*. This routine returns a pointer to the suffix where the match ended or NULL if the prefix was not present.

There is also an identical version of this method that works on constant strings.

`UrlPathPrefixMatches`

`char *`**`HttpdUtilities::UrlPathPrefixMatches`**` (char *`*`p_string`*`, const char *`*`p_prefix`*`);`

This routine determines if *`p_prefix`* is present in the URL *`p_string`*. The match is only considered successful if the match terminates on a path separator or the end of *`p_string`*. Percent-escaped characters in *`p_string`* match against their unescaped versions in *`p_prefix`*. This routine returns a pointer to the suffix where the match ended or NULL if the prefix was not present.

There is also an identical version of this method that works on constant strings.

`RemoveChars`

`void `**`HttpdUtilities::RemoveChars`**` (char *`*`p_string`*`, const char *`*`p_set`*`);`

This function removes any characters from *`p_string`* that are in *`p_set`*.

`FilterChars`

`void `**`HttpdUtilities::FilterChars`**` (char *`*`p_string`*`, const char *`*`p_set`*`);`

This function removes any characters from *`p_string`* that are not in *`p_set`*.

`GetLcExtension`

`char *`**`HttpdUtilities::GetLcExtension`**` (char *`*`p_file_name`*`);`

Get the extension of a file name (in lower case).

The input string is modified in place and the return value points into that string. You should make a copy of the string for this routine if you need it after the extension is obtained.

`GetComponentPath`

`char *`**`HttpdUtilities::GetComponentPath`**` (const char *`*`p_uri`*`, const char *`*`p_filename`*`);`

Given a URI or root path concatenate the *`p_filename`* to the *`p_uri`* path to produce a new path. Trailing forward slashes are adjusted so as to avoid duplicates.

Upon success a pointer to the newly formed path string is returned. It is the caller's responsibility to free it (using HttpdOpSys::Free). Upon failure NULL is returned.

`Normalize`

`char *`**`HttpdUtilities::Normalize`**` (const char *`*`p_path`*`, const char *`*`p_prefix`*`);`

Prepends the string *`p_prefix`* to the string *`p_path`* and removes any . or .. references. It returns a pointer to the resultant string in storage obtained from `Malloc()`. It is the caller's responsibility to free it (using HttpdOpSys::Free).

Because this method is typically used to convert URL paths into filesystem paths .. can not be used to access any path above `p_path`.

NormalizeUrl

`char *`**`HttpdUtilities::NormalizeUrl`**` (const char *`*`p_uri`*`, const char *`*`p_prefix`*`);`

This method is similar to the non-URL method HttpdUtilities::Normalize. One difference is that escaped characters in `p_uri` are interpreted for their actual value. For example if a path were `/example/path/%2e%2e/file` this would be normalized to `/example/file`. Another difference is that .. can be used to generate a URL that is the parent of `p_prefix`.

Hash

`size_t `**`HttpdUtilities::Hash`**` (const char *`*`p_key`*`);`

This method computes the hash index of `p_key` that can be used to speed up searches for that particular key. The returned value is a (non-unique) function of `p_key`.

HasTrailingSlash

`bool `**`HttpdUtilities::HasTrailingSlash`**` (const char *`*`p_path`*`);`

This function returns true if `p_path` ends in a forward slash.

HasPrefix

`const char *`**`HttpdUtilities::HasPrefix`**` (const char *`*`p_str`*`, const char *`*`p_prefix`*`);`

Determine if a string has a certain prefix (case insensitive). If the prefix is present then the returned value is the point past the prefix portion of `p_str`. If the prefix is not present then NULL is returned.

IsUriPathPrefix

`bool `**`HttpdUtilities::IsUriPathPrefix`**` (const char *`*`p_path`*`, const char *`*`p_prefix`*`);`

This method determines if the path pointed to by `p_path` contains the prefix specified by `p_prefix`. The path is a URL-style path and must use `/` as a path separator. The prefix must be at least one character in length.

If the path contains the prefix (either in its entirety or from the start of the path to some component boundary) then `true` is returned. Otherwise `false` is returned.

IsUriProtocol

`const char *`**`HttpdUtilities::IsUriProtocol`**` (const char *`*`p_uri`*`);`

Determine if a URL contains a protocol. The currently supported protocols are:

- *`http:`*
- *`https:`*
- *`ftp:`*

- `file:`

This is useful for deciding if a URL is relative or absolute. If the string is an absolute URL then the returned value is a pointer to the scheme-specific part of *`p_uri`*. If the string is a relative path then NULL is returned.

HostPortion

`char * **HttpdUtilities::HostPortion** (const char *p_uri);`

This helper method attempts to find a hostname portion in the standard URL schema. If no hostname is found or there is no memory available to hold the host name then NULL is returned.

It is the caller's responsibility to free the return value using HttpdOpSys::Free.

UriEncode

`char *HttpdUtilities::UriEncode (const char *p_uri, bool compact_space = false);`

URL-encode a string by quoting the metacharacters used in URL strings. Returns a pointer to an encoded string on success, NULL on failure. It is the caller's responsibility to free it (using HttpdOpSys::Free).

If *`compact_space`* is `true` then space characters (ASCII 0x20) are replaced with plus characters ("`+`").

NeedsUriEncoding

`bool HttpdUtilities::NeedsUriEncoding (const char *p_uri);`

This method determines if *`p_uri`* needs to be URL-encoded.

UriDecode

`char *HttpdUtilities::UriDecode (const char *p_encoded, bool plus_xlat);`

URL-decode a string, being safe about what we quote. A `?` character terminates the decoding (any characters following the `?` are truncated from the output).

`UriDecode` returns a pointer to the decoded string on success, NULL on failure. It is the caller's responsibility to free it (using HttpdOpSys::Free).

This method has two basic modes of operation. If *`plus_xlat`* is `false`, then `UriDecode` operates in URL mode. In this mode, the `+` character is not translated to a space. This mode is most often used to obtain the path component of a URL while removing the query string portion.

If the *`plus_xlat`* parameter is `true` then it is assumed that the string being decoded is a substring of a URL query string. In this case, the `+` character is translated into an ASCII space (character value 32).

UriDecodeSingle

`const char *HttpdUtilities::UriDecodeSingle (const char *p_encoded, char *p_output);`

This method URL-decodes the next character in *`p_encoded`*. When successful, the resultant character is placed in the variable pointed to by *`p_output`*. A pointer to the character after the one that was just processed is returned.

On failure, NULL is returned. The string pointed to by *`p_encoded`* must contain at least one character.

`HtmlQuote`

```
char *HttpdUtilities::HtmlQuote (const char *p_str);
```

Escape any HTML specific character entities in *`p_str`*. Returns a pointer to an encoded string on success, NULL on failure. It is the caller's responsibility to free it (using HttpdOpSys::Free).

`NeedsHtmlQuoting`

```
bool HttpdUtilities::NeedsHtmlQuoting (const char *p_str);
```

This method returns true if there are any characters in *`p_str`* that need escaping. Typically this method can be used to avoid the memory allocation done by `HtmlQuote` if no work is needed.

`CQuoteString`

```
char *HttpdUtilities::CQuoteString (const char *p_str, unsigned int
flags = STR_QUOTE_C);
```

This method escapes any necessary characters in *`p_str`* according to the rules of C-like languages. It returns a pointer to an escaped string on success, NULL on failure. It is the caller's responsibility to free it (using HttpdOpSys::Free).

`flags` consists of zero or more of the following flags:

Flag	**Meaning**
`STR_QUOTE_UNICODE`	Enabled the `\u` escape sequence for Unicode characters.
`STR_QUOTE_HEX`	Enabled the `\x` escape sequence for byte values and ASCII characters.
`STR_QUOTE_APOS`	If this flag is present then the single quote (`'` character) is escaped.
`STR_QUOTE_C`	This flag specifies that strings should be escaped in a manner that is compatible with the C language.
`STR_QUOTE_JSON`	This flag specifies that strings should be escaped in a manner that is compatible with the JSON encoding.

`BinToHex` (static buffer version)

```
char *HttpdUtilities::BinToHex (char *p_buffer, const void *p_data,
size_t nbytes);
```

This routine formats *`nbytes`* of data pointed to by *`p_data`* into an ASCII representation in the buffer pointed to by *`p_buffer`*. The destination buffer must be large enough to hold the formatted data and the resulting string is *not* zero terminated.

A pointer to the next slot in the buffer (i.e. one past the last written character) is returned.

`BinToHex` (dynamic string version)

```
char *HttpdUtilities::BinToHex (const void *p_data, size_t nbytes);
```

This routine formats *nbytes* of data pointed to by *p_data* into a dynamically allocated buffer that is zero terminated. If there is insufficient memory for the buffer then NULL is returned. It is the responsability of the caller to free the allocated buffer using HttpdOpSys::Free.

AssembleU16

HttpdUint16 **HttpdUtilities::AssembleU16** (const unsigned char **p_buf*);

To avoid processor architecture and endian issues, 16-bit values are encoded in a particular way by tools like SCPG. This method decodes the encoded 16-bit value produced by the host tools. The buffer pointed to by *p_buf* must be at least `HTTPD_U16_BYTES` bytes in length.

AssembleU32

HttpdUint32 **HttpdUtilities::AssembleU32** (const unsigned char **p_buf*);

To avoid processor architecture and endian issues, 32-bit values are encoded in a particular way by tools like SCPG. This method decodes the encoded 32-bit value produced by the host tools. The buffer pointed to by *p_buf* must be at least `HTTPD_U32_BYTES` bytes in length.

Lookup (Generic)

const void ***HttpdUtilities::Lookup** (const void **p_table*, size_t *table_size*, size_t *record_size*, size_t *key_offs*, const char **p_key*, int (*p_comp)(const char *p_a, const char *p_b));

Carries out a binary search for key *p_key* in the pre-sorted table *p_table*, which contains *table_size* elements of *record_size* bytes. The location of the key (as a `const char *`) is determined by *key_offs* which can be obtained with the standard `offsetof` macro.

The *p_comp* parameter points to a comparison function that should return a positive, non-zero value if *p_a* is sorted higher in the table than *p_b*; or a negative, non-zero value if *p_b* is sorted higher in the table than *p_a*; or zero if the two elements are equal.

The first parameter to the comparison function (*p_a*) is always the current entry being examined in the table.

Returns the discovered record upon success, NULL upon failure.

Lookup (Pairs)

const char ***HttpdUtilities::Lookup** (const HttpdPair **p_table*, size_t *table_size*, const char **p_key*, int (*p_comp)(const char *p_a, const char *p_b));

Carries out a binary search for key *p_key* in the pre-sorted table *p_table*, which contains *table_size* elements.

The *p_comp* parameter points to a comparison function that should return a positive, non-zero value if *p_a* is lexicographically greater than *p_b*; or a negative, non-zero value if *p_b* is lexicographically than *p_a*; or zero if the two elements are equal.

This function is based upon the more general `Lookup`. It is provided for convenience because `HttpdPair` tables are so common.

Returns the discovered value upon success, NULL upon failure.

FormatTime

`void **HttpdUtilities::FormatTime** (char *p_buf, size_t bufsz, const char *p_format, time_t t);`

Formats ANSI C time_t value *t* according to the specification supplied in *p_format*, and stores the result in *p_buf* (previously allocated by the caller to be *bufsz* in length.)

For more information, please refer to your C library documentation on the standard `strftime()` function.

Encode64

`char * **HttpdUtilities::Encode64** (const unsigned char *p_data, size_t len);`

This method encodes *len* bytes of data located at *p_data* into Base-64. The encoded data is returned as a NUL-terminated string in allocated memory. It is the caller's responsibility to free it (using HttpdOpSys::Free).

Decode64 (binary version)

`unsigned char * **HttpdUtilities::Decode64** (const char *p_encoded, size_t &output_len);`

This method decodes the Base-64 encoded data in *p_encoded*. Upon success, the decoded data is returned in an allocated buffer and *output_len* is set to be the decoded length in bytes. Upon failure, NULL is returned. It is the caller's responsibility to free it (using HttpdOpSys::Free).

Decode64 (String version)

`char * **HttpdUtilities::Decode64** (const char *p_encoded);`

This method decodes the Base-64 encoded data in *p_encoded*. Upon success, the decoded data is NUL-terminated and returned in an allocated buffer. Upon failure, NULL is returned. It is the caller's responsibility to free it (using HttpdOpSys::Free).

NextCharInUtf8

`bool **HttpdUtilities::NextCharInUtf8** (HttpdUnicodeChar &uc, const char *&p_buffer, size_t window);`

This method gets the next Unicode character in the provided buffer of UTF-8 encoded Unicode characters. The method returns true if the next character is obtained without error or false if more than *window* bytes are needed to decode the character, the value of *window* is `0`, or the buffer is not a valid UTF-8 string.

If successful then the buffer pointer, *p_buffer* is updated to point to the byte after the decoded character that is placed into *uc*.

The typical use of this function is in a loop to decode each Unicode character as a string is walked. For example:

```
…
char    *p_utf8 = some_utf8_source();
size_t  *p_end  = p_utf8 + strlen(p_utf8) + 1;
```

```
HttpdUnicodeCharacter uc;
while (HttpdUtilities::NextCharInUtf8(uc, p_utf8, (size_t )(p_end - p_utf8)))
{
  if (uc == 0)
    break; // End of string.

  // Process character uc here.
}
…
```

AppendUtf8

`bool` **`HttpdUtilities::AppendUtf8`** `(HttpdUnicodeChar` *`uc`*`, const char *&`*`p_buffer`*`, size_t &`*`buflen`*`);`

This method appends a Unicode character to the buffer using the UTF-8 encoding. True is returned if there is enough room to hold the character and it can be properly encoded or false if there is an error. If successful then *`p_buffer`* is updated to point to the byteafter the encoded data and *`buflen`* is decremented by the number of bytes needed to encode the character.

DequoteToken

`char *` **`HttpdUtilities::DequoteToken`** `(const char *&`*`p_front`*`, const char *`*`p_term`*` = …);`

There are several places in the HTTP protocol where strings within *MIME* headers are quoted. This is very typical of token/value pairs following a *MIME* value. This method dequotes those strings and returns the real values.

`p_front` should point to the start of the token. On success, a pointer to the dequoted string is returned and *`p_front`* is updated to point to just after the string value. It is the caller's responsibility to free the return value (using HttpdOpSys::Free).

On failure, NULL is returned and the value of *`p_front`* is undefined.

By default a non-quoted string is terminated according to the definition of `token` in RFC 2616. If *`p_term`* is specified then it is the set of characters that terminate a token. Keep in mind that there are sometimes subtle differences between the separator in different *MIME* headers.

QuoteToken

`void` **`HttpdUtilities::QuoteToken`** `(char *&`*`p_dest`*`, const char *`*`p_plain`*`);`

This method quotes the string in *`p_plain`* if necessary to make it safe for use as a *MIME* token. The quoted result is placed into the buffer pointed to by *`p_dest`*. On return, the pointer *`p_dest`* is updated to point to the unused byte after the quoted string.

Important

It is important to remember when using this function that no null terminator is stored in *`p_dest`*. It is the responsibility of the caller to add one if necessary.

Another important attribute about this function is that the buffer pointed to by *`p_dest`* must be sufficiently large to hold the worst-case scenario of every character requiring quoting. This is 2 characters larger than double the length of *`p_plain`*.

TokenPresent

bool **HttpdUtilities::TokenPresent** (const char **p_mime*, const char **p_token*);

This method searches for *p_token* in the *MIME* line that is a set of tokens, *p_mime*. If the token is present (regardless of value, if any) then true is returned. Otherwise false is returned.

RandomString

void **HttpdUtilities::RandomString** (char **p_dest*, size_t *len*);

This method fills *p_dest* with a string of random alphanumeric characters, *len* characters long. The buffer passed in must be at least large enough to hold *len* characters and the terminating zero byte.

ParseHttpVersion

HttpdProtocolVersion **HttpdUtilities::ParseHttpVersion** (const char **p_version*);

Given an HTTP version string *p_version*, return a representative HttpdProtocolVersion value. If no version can be distinguished, the constant referring to HTTP version 0.9 is returned (that protocol version is recognized by its lack of version identification on the wire).

The expected formatting of *p_version* is “`HTTP/`*`X.Y`*”, where *X* represents the major version, and *Y* the minor version.

A list of current protocol version constants can be found in Table 2.1, “Predefined HttpdProtocolVersion Constants”, to simplify comparisons.

TokenizePortions

bool **HttpdUtilities::TokenizePortions** (char *sep*, char **p_buf*, char ***pp_target*, …);

This method chops up the string pointed to by *p_buf* at boundaries specified by the character *sep*. The address of the first character is placed into each successive parameter starting with *pp_target*. The list of pointers must be terminated with a NULL value.

This method does not copy the string in any way, the pointers assigned to *pp_target* are only valid as long as the buffer pointed to by *p_buf* is valid.

It is important to terminate the list with *(char **)0*, not NULL, because some CPU architectures have different NULL pointer representations for different types and the compiler does not know the type of the pointer because it is part of a variable argument list.

MemPBrk

void ***HttpdUtilities::MemPBrk** (void **p_buffer*, size_t *n*, const void **p_term*, size_t *termsz*);

This method is similar to the `strpbrk()` routine. It searches in upto *n* bytes pointed to by *p_buffer* for the termination bytes. The termination bytes are specified by *p_term* and *termsz*.

If the search bytes do not exist anywhere in the extent of the buffer then NULL is returned. Otherwise the address of the byte preceeding the first termination byte found is returned.

MemCountByte

`size_t` **`HttpdUtilities::MemCountByte`** `(const void *`*`p_buffer`*`, size_t` *`buflen`*`, unsigned char` *`val`*`);`

This method scans the buffer and counts the number of times *`val`* appears in the buffer.

FindBoundary

`int` **`HttpdUtilities::FindBoundary`** `(HttpdReceiver *`*`p_receiver`*`, const char *`*`p_boundary`*`);`

Before `HttpdBoundaryReader::Read` can be called, this method must be called to find the initial starting point of the data (also delimited by *`p_boundary`*).

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

When `0` is returned, `HttpdUtilities::IsLastBoundary` should be called to complete the parsing of the boundary string. It is entirely possible to have a multipart *MIME* message that contains no subparts. In that case, the `IsLastBoundary` routine will indicate that no more boundaries are expected. In this case, no instance of `HttpdBoundaryReader` should be created.

IsLastBoundary

`int` **`HttpdUtilities::IsLastBoundary`** `(HttpdReceiver *`*`p_receiver`*`, bool &`*`finished`*`);`

After `HttpdBoundaryReader::Read` or `HttpdUtilities::FindBoundary` are called and return success, this method should be called. Aside from completing some parsing activities, the parameter *`finished`* will be set to `false` if more subparts are expected. Otherwise, *`finished`* will be set to `true`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Public Data

mRoot

`const char mRoot[];`

This variable represents the path separator used in concatenating path segments, and other miscellaneous uses. Its default value of “/” should not be changed without considerable thought, since URI formatting standards specify it as a hierarchical separator.

mNetTimeFormat

`const charmNetTimeFormat[];`

Supplied as a parameter to HttpdUtilities::FormatTime, this is a string representing the desired human-readable format to which system time should be converted. In an ANSI C environment, the underlying library function ultimately responsible for this transformation is `strftime()`. For further information on the contents of this variable, local C library documentation on that function should be consulted.

`mPastTime`

```
const charmPastTime[];
```

This is a hard-coded time constant (formatted for HTTP) of the UNIX epoch, which is always considered to be in the past.

`mContentLength`

```
const charmContentLength[];
```

This hard-coded array is the HTTP `Content-length` string used when generating headers.

`mContentType`

```
const charmContentType[];
```

This hard-coded array is the HTTP `Content-type` string used when generating headers.

`mLineTerm`

```
const charmLineTerm[];
```

This hard-coded array is the HTTP newline sequence. There is no NULL terminating byte on this array.

`HttpdMD5` Reference

Introduction

MD5 is a one-way *hashing function* defined in RFC-1321. It is useful both as a checksum and (in some cases) for cryptographic purposes. The `HttpdMD5` class provides a implementation of the MD5 hashing function.

If the compile-time option INC_FAST_MD5 is enabled the `HttpdMD5` class uses a high-speed but large algorithm. Otherwise a slower but compact algorithm is used. In general, it is recommended that the more compact algorithm be used unless MD5 hashes are used extensively in the application.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`Update` (Buffer version)

```
void HttpdMD5::Update (const void *p_data, size_t count);
```

Hash the data pointed to by *`p_data`* that is *`count`* bytes long. After the `HttpdMD5` object is constructed (or reset) this method (and the string version of `Update`) may be called as many times as necessary in succession to hash all of the data.

`Update` (String version)

```
void HttpdMD5::Update (const char *p_string);
```

Hash the contents of the string pointed to by *`p_string`*, not including the terminating zero byte. After the `HttpdMD5` object is constructed (or reset) this method (and the buffer version of `Update`) may be called as many times as necessary in succession to hash all of the data.

`Final`

```
void HttpdMD5::Final (HttpdMD5Digest digest);
```

After all of the data has been hashed (via `Update`) this method retrieves the digest from the `HttpdMD5` object and copies it to *`digest`*.

Important

It is important to note that once this method is called the `HttpdMD5` object is no longer valid and no further hashing on that particular instance should be performed until `Reset` is called.

`Reset`

```
void HttpdMD5::Reset (void);
```

This method resets the state of the hashing engine. This method allows a single object instance to compute any number of hashes.

`HttpdMimeParser` Reference

Introduction

`HttpdMimeParser` provides a general purpose parsing engine for *MIME* headers. These headers are found in many Internet protocols, and HTTP is no exception. The `HttpdRequest` class uses `HttpdMimeParser` to process headers from incoming requests.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdMimeParser`

```
HttpdMimeParser::HttpdMimeParser (void);
```

The constructor just does a simple initialization. The heavy-weight initialization is done by the `Initialize` method. This allows the memory allocation to be avoided if no *MIME* headers need to be parsed (but the object is owned by another object).

Initialize

`bool` **`HttpdMimeParser::Initialize`** `(void);`

This method must be called before any other methods are called. If this method returns false, then the `HttpdMimeParser` object could not be initialized and should not be used. Success is indicated by a return value of true.

ReadLine (socket version)

`char *` **`HttpdMimeParser::ReadLine`** `(char *`*`p_line_buf`*`, HttpdSocket &`*`socket`*`, unsigned int `*`timeout`*`);`

This static method reads a line, structured the way *MIME* lines are, from *`socket`*. The *`p_line_buf`* pointer is used as a temporary working space and must point to at least HTTPD_MAX_INPUT_LINE bytes of storage. The value of the HTTPD_MAX_INPUT_LINE constant is determined by the value of the `MAX_INPUT_LINE` build-time parameter.

Any kind of error reading the line from *`socket`* is indicated by a return value of NULL. If a valid line is not received by *`timeout`* seconds, it is considered an error and NULL is returned.

If a line (that is not empty) is read successfully, it is copied into a fresh buffer and a pointer to that buffer is returned. However, *MIME* headers are always terminated by a blank line. As an optimization to avoid allocating a buffer for an empty string, the value of *`p_line_buf`* is returned if a line is read, but empty.

Callers should check for this special return value and consider that a marker for the end of the headers. This approach allows three different outcomes (error, empty, or valid data) to be returned through a single pointer.

Note

This function is static and does not depend on an initialized `HttpdMimeParser` object.

When `ReadLine` returns a pointer to a processed string (and not NULL or *`p_line_buf`*), it is the caller's responsibility to free it (using HttpdOpSys::Free).

ReadLine (`HttpdReceiver` version)

`char *` **`HttpdMimeParser::ReadLine`** `(char *`*`p_line_buf`*`, HttpdReceiver *`*`p_receiver`*`, unsigned int `*`timeout`*`);`

This static method reads a line from *`p_receiver`*. The semantics are identical to the socket version (above).

Note

This function is static and does not depend on an initialized `HttpdMimeParser` object.

ParseLine

`bool` **`HttpdMimeParser::ParseLine`** `(char *`*`p_line`*`);`

This method processes a header line (typically read with `ReadLine`). If the header line contained in *`p_line`* is valid, true is returned. On failure, false is returned and no further method calls should be made to the `HttpdMimeParser` object.

Note

This method should only be called after the `Initialize` method is called.

The string pointed to by *`p_line`* is modified during the parse. It does not have to remain valid after this method completes, but it should be in modifiable storage. This is normally transparent as `ReadLine` returns a saved copy of the string.

Finish

`void` **`HttpdMimeParser::Finish`** `(void);`

This method should be called after all header lines are parsed (using `ParseLine`).

Header

`const char *` **`HttpdMimeParser::Header`** `(const char *`*`p_key`*`);`

This method is used to look up the value of a *MIME* header (only if it is unique by name). A pointer to the value of the header named *`p_key`* is returned on success, NULL is returned if the specified header does not exist.

Note

This method should only be called after the `Finish` method is called.

Count

`size_t` **`HttpdMimeParser::Count`** `(void);`

This method returns the number of available header entries.

Note

This method should only be called after the `Finish` method is called.

Pair

`HttpdPair *` **`HttpdMimeParser::Pair`** `(size_t `*`index`*`);`

This method returns a pointer to the `HttpdPair` object for the header specified by *`index`*. This method is useful for certain types of headers which may appear more than once. Using the `Count()` method to compute the upper bound of the index, all of the headers can be enumerated.

Note

This method should only be called after the `Finish` method is called.

ParseHeaders

`bool` **`HttpdMimeParser::ParseHeaders`** `(HttpdReceiver *`*`p_recvr`*`);`

This method reads a series of *MIME* headers into the parser object. Upon success, true is returned. In the case of an error, false is returned.

Note

This method automatically calls `Initialize` and `Finish`. The state of the parser is completely initialized by calling this routine. If it returns true then headers may be retrieved from the object using the `Header` method.

`HttpdTimeStamp` Reference

Introduction

The profusion of alternative time and date format conventions makes it advantageous to provide a single point of conversion and storage. `HttpdTimeStamp` provides such an interface. Methods are provided to populate its member variables from ANSI C time_t values and various character string formats, as well as to compare a previously constructed `HttpdTimeStamp` with a current one.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`Parse`

```
bool HttpdTimeStamp::Parse (const char *p_str_rep);
```

Given formatted string *`p_str_rep`*, populate the parent object's member variables with time and date information. The input character string may be formatted in a few alternative ways, as described by the following table:

Table 2.2. Supported Time Format Specifications

Format Type	Example
RFC 822	“Sun, 06 Nov 1994 08:49:37 GMT”
RFC 850	“Sunday, 06-Nov-94 08:49:37 GMT”
ANSI C `asctime()`	“Sun Nov 6 08:49:37 1994”

Returns true upon success, false upon failure.

`Convert`

```
bool HttpdTimeStamp::Convert (const struct tm *p_tm);
```

Given a pointer to an ANSI C time structure *`p_tm`*, populate the member variables of the current `HttpdTimeStamp` object.

Returns true upon success, false upon failure.

Validate

`bool` **`HttpdTimeStamp::Validate`** `(void);`

Validate that the values of the `HttpdTimeStamp` structure are within their valid ranges.

Returns true if all the fields are valid, false otherwise.

FindDayOfWeek

`void` **`HttpdTimeStamp::FindDayOfWeek`** `(void);`

This method adjusts the `mWeekDay` parameter to reflect the day for specified by `mDay`, `mMonth`, `mYear`.

Note

This routine can only compute the correct day from the year 1583 or later.

Compare

`int` **`HttpdTimeStamp::Compare`** `(const HttpdTimeStamp *`*`p_to`*`);`

Compares the parent `HttpdTimeStamp` object with the one indicated by *`p_to`*, and indicates their relationship along a timeline.

Returns less than zero if *`p_to`* is in the future relative to the called `HttpdTimeStamp`, 0 if they are equal, or greater than zero if *`p_to`* is in the past.

Set

`void` **`HttpdTimeStamp::Set`** `(const HttpdTimeStamp *`*`p_to`*`);`

This method sets the time of the object to the same time of the object pointed to by *`p_to`*.

Format

`bool` **`HttpdTimeStamp::Format`** `(char *`*`p_buffer`*`, size_t` *`bufsiz`*`, const char *`*`p_format`*`);`

This function formats the time as a string. The string is written to *`p_buffer`*. If the written representation would be larger than *`bufsiz`* bytes (including the null termination byte) then false is returned. The *`p_format`* string is a formatting template similar to the one used in `strftime`. The supported specifiers are: `a` (weekday), `d` (day of month), `b` (month), `Y` (four digit year), `y` (two digit year), `H` (hours), `M` (minutes), and `S` (seconds).

FormatAsISO8601

`void` **`HttpdTimeStamp::FormatAsISO8601`** `(char *`*`p_buffer`*`);`

This function formats the time as a string in the ISO 8601 format. The buffer pointed to by *`p_buffer`* must be at least `HttpdTimeStamp::ISO8601_FMT_BUFSIZE` characters in size.

TimeInGMT

`bool` **`HttpdTimeStamp::TimeInGMT`** `(time_t &`*`time`*`);`

This method sets the time of the object to the specified time in GMT.

Public Data

mDay

```
unsigned int mDay
```

The numerical day-of-month (e.g. `16`).

mWeekDay

```
unsigned int mWeekDay
```

The numerical day-of-the-week (e.g. `1` is Monday).

mMonth

```
unsigned int mMonth
```

An internal table index resulting in an English representation of the month.

mYear

```
unsigned int mYear
```

The numerical year (e.g. `1996`).

mHour

```
unsigned int mHour
```

The numerical hour (e.g. `07`).

mMinute

```
unsigned int mMinute
```

The numerical minute (e.g. `54`).

mSecond

```
unsigned int mSecond
```

The numerical second (e.g. `33`).

`HttpdWritable` Reference

Introduction

The `HttpdWritable` is a base class that represents a sink of data. Typically this is the base class for a socket but can also be derived into other base classes. Many other Seminole classes implement the `HttpdWritable` interface such as HttpdSocket, HttpdContentSink, and HttpdCountingSink.

Public Methods

`Write`

`int` **`HttpdWritable::Write`** `(size_t` *`sz`*`, const void *`*`p_data`*`);`

This pure virtual function is the interface for writing data to the sink. On success, *`sz`* bytes are written to the sink from the buffer pointed to by *`p_data`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`WriteString`

`int` **`HttpdWritable::WriteString`** `(const char *`*`p_string`*`);`

This method writes a string pointed to by *`p_string`* to the sink.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`WriteStringAndFree`

`int` **`HttpdWritable::WriteStringAndFree`** `(char *`*`p_string`*`);`

If *`p_string`* is NULL then `HttpdOpSys::ERR_OUTOFMEM` is returned. Otherwise, *`p_string`* is written to the sink and then released using `HttpdOpSys::Free`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`NewLine`

`int` **`HttpdWritable::NewLine`** `(void);`

This method writes an HTTP line terminator (`\r\n`).

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Printf`

`int` **`HttpdWritable::Printf`** `(const char *`*`p_format`*`, …);`

This method implements a subset of the features provided by ANSI C's `printf()` library function. Formatted output is written to the writable using the abstract `HttpdWritable::Write` method. Permissible format specifiers are as follows:

Table 2.3. Supported Print Format Specifications

Specification	Arguments	Formatting
`%s`	String (const char pointer)	The string is written out “as is”.

Specification	**Arguments**	**Formatting**
`%d`	int	The number is formatted as a signed decimal number.
`%x`	unsigned long	The number is formatted as an unsigned hexadecimal number.
`%t`	unsigned long	The number is formatted as an unsigned decimal number.
`%%`	No arguments	Produces a literal percent sign (%)
`%f`	Number of bytes to write (unsigned int) and the fill character (char)	The fill character is repeated as many times as indicated by the first argument (count).
`%q`	String (const char pointer)	The string is written out as defined by RFC 2616 for quoted strings appearing as attributes in *MIME* headers.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Indent

```
int HttpdWritable::Indent (unsigned int depth);
```

This method writes *`depth`* space characters. For efficiency this routine attempts to avoid single byte writes.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Httpd Reference

Introduction

The `Httpd` is an instance of a webserver that is associated with a particular port and (optionally) *transport*. This class is also the hub of request processing; it creates instances of `HttpdRequest` and applies them to subclasses of `HttpdHandler`. It also contains support methods that `HttpdRequest` objects can use to handle requests.

In addition to request processing the `Httpd` class also provides various administrative functions such as:

- Startup and shutdown of Seminole
- Security checks prior to request handler invocation
- Methods for adding and removing request handlers

Public Methods

Httpd

```
Httpd::Httpd (const char *p_host_name, HttpdIpPort port = 80);
```

Constructs a web-server object. The `p_host_name` parameter should be a valid host name or IP address for referncing this web server. The `port` parameter specifies the port address for this web server. By default it is `80` but may need to be adjusted for other *transports* (such as SSL).

Init

```
static int Httpd::Init (void);
```

This static method must be the first method called in the Seminole *API*. It initializes the portability layer, the socket layer and other global internal state not specific to any particular instance of `Httpd`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Note

This method is not idempotent and should only be called once before any other Seminole objects are constructed or methods invoked.

ServerName

```
const char *Httpd::ServerName (void);
```

Returns the "brand name" of the web server. The lifetime of this string must be equal to or greater than the lifetime of this object.

If INC_DYNAMIC_SERVER_NAME is enabled then this is an overrideable method; it can be used for internationalization and product customization purposes. If the feature is disabled then this method is static.

It is recommended that the returned string is always in the form of:

```
Seminole/X.XX (…)
```

Where `X.XX` is the current version number of Seminole. Application-specific branding should be placed within the parenthesis.

Start

```
bool Httpd::Start (const char *const *pp_options);
```

Perform the appropriate network layer initialization and begin accepting HTTP requests. Returns true when startup is successful, false otherwise.

The NULL terminated list of strings pointed to by `pp_options` are passed to the `HttpdSocket::Listen` method. If the INC_MULTIPLE_TRANSPORTS option is enabled then an required attribute of `sock:` determines the *transport* used for this instance of `Httpd`.

The `pp_options` must not be NULL its self. If no socket options are desired then the default value of the parameter, `HttpdSocket::mEmptySocketOptions`, may be passed as this parameter.

Stop

```
void Httpd::Stop (HttpdShutdownType type = GRACEFUL);
```

Perform a shutdown of the webserver. If the `type` parameter is `GRACEFUL` (the default) then a graceful shutdown is performed. A graceful shutdown stops processing any additional clients and ceases accepting new clients. The caller of `Stop` will be blocked until any in-process HTTP requests are completed. This ensures that after the call to `Stop` returns any objects involved in request processing may be safely destroyed.

A `HARD` stop will abruptly terminate any in-process requests. Although, as with a `GRACEFUL` shutdown the `Stop` method does not return until it is safe to destroy all resources, this is usually much quicker with a `HARD` stop than with a `GRACEFUL` stop.

Note

This method is not defined unless the INC_SHUTDOWN option is enabled.

The ability to perform a hard stop is dependant on the socket layer implementation and may not be implemented on a particular target platform.

Install

```
void Httpd::Install (HttpdHandler *p_handler);
```

Install the specified handler. Requests will then be dispatched through the handler if the prefix matches. The server must be in a stopped state before calling this method.

ServerHost

```
const char * Httpd::ServerHost (void);
```

Returns the machine hostname associated with this instance of `Httpd`. This value is simply the second argument of the class constructor.

Remove

```
HttpdHandler *Httpd::Remove (HttpdHandler *p_handler);
```

```
HttpdHandler *Httpd::Remove (const char *p_prefix);
```

Remove the specified handler/prefix mapping; a prefix can be supplied, in which case the corresponding handler will be removed, or a direct pointer to the unwanted handler can be supplied, in which case that handler is removed.

Returns a pointer to the handler which was removed, or NULL if no matching handler was found.

Note

These methods are not defined unless the INC_SHUTDOWN option is enabled.

Port

```
HttpdIpPort Httpd::Port (void);
```

Returns the network port to which this instance of `Httpd` is bound (specified as the first argument of the class constructor).

ListenSock

`HttpdSocket &`**`Httpd::ListenSock`**` (void);`

Returns a reference to the listening socket object.

ServerWideRequest

`void `**`Httpd::ServerWideRequest`**` (HttpdRequest *`*`p_request`*`);`

HTTP includes the notion of a "server wide request." These are request methods that are not directed to any particular URL on the server (identified by a a `*` in the request).

These requests are sent to this method. Subclasses may override this method to provide additional functionality. Subclasses should call the `Httpd` implementation of this method if they do not respond to the request.

Protected Methods

These methods are virtual and may be extended by subclasses of `Httpd`. Because instances of `HttpdRequest` are created internally there is no way for client code to extend the behavior of methods in that class. The typical pattern is that methods that need to be overridden are in the `Httpd` class and a wrapper method from the `HttpdRequest` object calls these methods below.

Allowed

`virtual bool `**`Httpd::Allowed`**` (HttpdIpAddress `*`addr`*`);`

This method determines if the client with the specified address is allowed to connect. If the connection is to be allowed then this method should return true, otherwise false should be returned.

ResponseHeader

`virtual void `**`Httpd::ResponseHeader`**` (HttpdRequest *`*`p_request`*`, const HttpdResponse &`*`resp`*`);`

Send a basic set of HTTP response headers to the client, possibly in preparation for further data from a handler.

The currently supported response codes are detailed in Supported HTTP Response Codes. For a more detailed discussion of the circumstances under which a given code might be used, the reader is referred to the appropriate standards document, RFC 2616 (Hypertext Transfer Protocol 1.1) [ftp://ftp.isi.edu/in-notes/rfc2616.txt] or previous versions as appropriate to the desired application.

Supported HTTP Response Codes

- `HTTPD_RESP_CONTINUE` (`100`) - Continue with request
- `HTTPD_RESP_PROTOCOL` (`101`) - Protocol switch OK
- `HTTPD_RESP_OK` (`200`) - Request succeeded - content follows
- `HTTPD_RESP_CREATED` (`201`) - Request for resource creation succeeded (typically in response to `PUT` requests)
- `HTTPD_RESP_ACCEPTED` (`202`) - Request accepted

- `HTTPD_RESP_NONAUTH_INFO` (`203`) - Metainformation not authoritative
- `HTTPD_RESP_NO_CONTENT` (`204`) - Request succeeded, no entity-body
- `HTTPD_RESP_SOME_CONTENT` (`205`) - Request succeeded, client to reset view
- `HTTPD_RESP_PARTIAL_CONTENT` (`206`) - Request succeeded, partial entity body follows (used for with the `Range:` header)
- `HTTPD_RESP_MULTI_STATUS` (`207`) - Multiple objects were affected. The entity body describes the outcomes of the operation. This is typically used with *WebDAV* methods
- `HTTPD_RESP_MULTI_CHOICE` (`300`) - Requested resource has multiple representations
- `HTTPD_RESP_MOVED_PERM` (`301`) - Requested resource has new, permanent URI
- `HTTPD_RESP_MOVED_TEMP` (`302`) - Requested resource has new, temporary URI
- `HTTPD_RESP_SEE_OTHER` (`303`) - Requested resource has moved, use `GET` to new URI
- `HTTPD_RESP_NOT_MODIFIED` (`304`) - Conditional `GET`, but document not modified
- `HTTPD_RESP_USE_PROXY` (`305`) - Requested resource must be accessed via a proxy
- `HTTPD_RESP_NEO_TEMP_MOVED` (`307`) - Requested resource has new, temporary URI
- `HTTPD_RESP_BAD_REQ` (`400`) - Malformed request syntax
- `HTTPD_RESP_UNAUTHORIZED` (`401`) - Request requires authentication, but none was provided or incorrect credentials were supplied
- `HTTPD_RESP_PAYMENT_REQ` (`402`) - Reserved for future use
- `HTTPD_RESP_FORBIDDEN` (`403`) - Request administratively forbidden
- `HTTPD_RESP_NOT_FOUND` (`404`) - Requested URI was not found
- `HTTPD_RESP_METHOD_NOT_ALLOWED` (`405`) - Request method specified not legal for this URI
- `HTTPD_RESP_NOT_ACCEPTABLE` (`406`) - Resource requested cannot generate response entities acceptable to the client
- `HTTPD_RESP_PROXY_AUTH` (`407`) - Authentication required to use a proxy, but none was provided or incorrect credentials were supplied
- `HTTPD_RESP_REQUEST_TIMEOUT` (`408`) - Client request timeout
- `HTTPD_RESP_CONFLICT` (`409`) - Current resource state in conflict with request
- `HTTPD_RESP_GONE` (`410`) - Requested resource not available, and no new URI is known
- `HTTPD_RESP_LENGTH_REQ` (`411`) - Missing `Content-Length:` header entry
- `HTTPD_RESP_PRECOND_FAILED` (`412`) - Request pre-condition failed on server side
- `HTTPD_RESP_TOO_LARGE` (`413`) - Request entity too large
- `HTTPD_RESP_URI_TOO_LARGE` (`414`) - Request URI too long
- `HTTPD_RESP_UNSUPPORTED_MEDIA` (`415`) - Unsupported media type for requested resource and/or method

- `HTTPD_RESP_RANGE` (`416`) - Resource extent does not match requested range
- `HTTPD_RESP_EXPECTATION_FAILED` (`417`) - Client expectation cannot be met by server
- `HTTPD_RESP_LOCKED` (`423`) - The resource is locked
- `HTTPD_RESP_SRV_ERROR` (`500`) - Unexpected server error
- `HTTPD_RESP_METHOD_NOT_IMPL` (`501`) - Unrecognized or unimplemented request method
- `HTTPD_RESP_BAD_GATEWAY` (`502`) - Invalid response from upstream provider or application
- `HTTPD_RESP_UNAVAILABLE` (`503`) - Temporary inability to service request
- `HTTPD_RESP_GATEWAY_TIMEOUT` (`504`) - Timeout on upstream provider or application response
- `HTTPD_RESP_HTTP_VERSION` (`505`) - Requested HTTP version is not supported
- `HTTPD_RESP_INSUFFICIENT_SPACE` (`507`) - There is insufficient storage space to perform the requested operation

ResponseBody

`virtual void` **`Httpd::ResponseBody`** `(HttpdRequest *`*`p_request`*`, const HttpdResponseMsg &`*`response`*`, const char *`*`p_url`*`);`

This method is used to send a simple HTML-formatted document in accompaniment to HTTP error responses. *`p_title`* points to a string describing an alternative URL. For example, in the case of a redirect this should point to a string containing the new URL. This parameter may also be NULL.

Respond

`void` **`Httpd::Respond`** `(HttpdRequest *`*`p_request`*`, int `*`status`*`);`

This method generally encapsulates the `ResponseHeader` and the `ResponseBody` methods, and is used as a general error handling mechanism when HTTP errors and associated human-readable messages are to be sent to a client. An HTTP status code is provided in *`status`*. See Supported HTTP Response Codes for possible values.

`HttpdRequest` Reference

Introduction

`HttpdRequest` represents a single HTTP request in time being handled by Seminole. An `HttpdRequest` object is instantiated by the `Server()` method within `Httpd` for each incoming request, and then discarded once a handler either processes the request or no handler is found to do so.

The constructor of this class performs some basic request processing before a handler is located:

- The request is parsed and checked for syntactic validity
- Public variables are populated with various things of interest in the headers
- *MIME* headers are tokenized, and the entire header list sorted

Understanding the contents and interfaces of `HttpdRequest` is quite important when writing a new handler class, since it is the primary unit of data passed from Seminole's core to the handlers.

The request object also contains an unused data member called `mpData` that can be used by subclasses for tracking additional state. This void pointer can be used to reduce the amount of context that needs to be passed between the methods of a complex handler implementation. This pointer is initialized to NULL when the request object is created.

Once a request is received by a handler and the necessary processing performed, the next step is naturally to send a response. `HttpdRequest` provides methods to do this also; for most handlers the starting point will often be `ResponseHeader()`.

Public Methods

Server

```
Httpd *HttpdRequest::Server (void);
```

Returns a pointer to the `Httpd` object which instantiated this `HttpdRequest`.

Method

```
const char *HttpdRequest::Method (void);
```

Returns a character pointer to a string describing the HTTP method associated with this `HttpdRequest` (`GET`, `PUT`, and so on).

IsHeadRequest

```
bool HttpdRequest::IsHeadRequest (void);
```

Determine if the request method is `HEAD`.

IsGetRequest

```
bool HttpdRequest::IsGetRequest (void);
```

Determine if the request method is `GET`.

IsPostRequest

```
bool HttpdRequest::IsPostRequest (void);
```

Determine if the request method is `POST`.

IsOptionsRequest

```
bool HttpdRequest::IsOptionsRequest (void);
```

Determine if the request method is `OPTIONS`.

PostIsMultipartMime

```
bool HttpdRequest::PostIsMultipartMime (void);
```

If the request method is `POST` and this method returns true then the request body is encoded using mutli-part *MIME* and should be processed using the `HttpdMultipartCgiParser` class.

If this method returns false then the body of the `POST` should be processed with `HttpdCgiParameter::ParsePostData`.

ContentAvailable

`bool` **`HttpdRequest::ContentAvailable`** `(const char *`*`p_type`*`);`

This determines if an entity body is available in the request with a *MIME* type of *`p_type`*.

Protocol

`HttpdProtocolVersion` **`HttpdRequest::Protocol`** `(void);`

Returns an HttpdProtocolVersion value specifying the HTTP version associated with this `HttpdRequest`.

Path

`char *`**`HttpdRequest::Path`** `(void);`

Returns a character pointer to the path component of the URI contained within this `HttpdRequest`, exactly as it was sent by the client.

Query

`const char *`**`HttpdRequest::Query`** `(void);`

Returns a character pointer to the query component of the URI contained within this `HttpdRequest`. If no query was provided in the request, NULL is returned.

ClientAddr

`HttpdIpAddress` **`HttpdRequest::ClientAddr`** `(void);`

Returns the client IP address for this request.

Socket

`HttpdSocket &`**`HttpdRequest::Socket`** `(void);`

Returns the `HttpdSocket` object which provides communication with this request's client.

Note

Sending data to the HTTP client should not be done through the socket. Instead, the `Output` method returns the correct object.

Output

`HttpdWritable &`**`HttpdRequest::Output`** `(void);`

Returns the object that should be the destination for data sent to the HTTP client.

Header

`const char *`**`HttpdRequest::Header`** `(const char *`*`p_mime`*`);`

Returns the value associated with HTTP request header `p_mime`, or NULL if the header is not found.

`Headers`

```
HttpdMimeParser & HttpdRequest::Headers (void);
```

Returns a reference to the *MIME* parser object used to parse the header section of this request.

`CompleteUri`

```
const char * HttpdRequest::CompleteUri (void);
```

If a full request URI was presented as the argument to the request then this method will return that URI. If the request did not include a full URI, then this method returns NULL.

`LastReq`

```
bool HttpdRequest::LastReq (void);
```

This method determines if this is the last request that will be processed by the current connection. This can be the case if any of the previous request processing resulted in a case that requests the end of the connection.

This is most typically done because a content handler was not able to determine the content-length of the data it was going to send. There are also administrative reasons for why this request may be the last, such as a timeout waiting for a request or limit or quota reached.

Note

This method is not available if INC_PERSISTENT_CONN is not enabled.

`ResponseHeadersSent`

```
bool HttpdRequest::ResponseHeadersSent (void);
```

This method determines if the response headers have been sent already, typically via the `Respond` or `ResponseHeader` methods. The response headers generated by the `ResponseHeader` method include the `Connection` line.

It is best if the `SetLastReq` method is called before the response headers. Otherwise if no buffering is being performed and the content being generated has an unknown size then the server must abruptly close the connection after the content is sent. This can result in additional round trips by clients.

Note

The `HttpdDynamicOutput` class handles these complexities automatically.

This method is not available if INC_PERSISTENT_CONN is not enabled.

`SetLastReq`

```
void HttpdRequest::SetLastReq (void);
```

This forces this request to be the last on the connection. After calling this method, `LastReq` will return true. There is no way to undo the effects of this method once called.

This method should be called before any of `Respond`, `ResponseHeader`, `Redirect`, or `RedirectWithQuery` are called. This restriction is because part of the HTTP protocol specification requires that a `Connection: close` header be sent out on the last result.

If this method is called after the response is issued then no further requests will be allowed from the connection after processing. However this mode of operation should only be done in error scenarios.

Note

This method is not available if INC_PERSISTENT_CONN is not enabled.

`RequestedHostName`

`const char *` **`HttpdRequest::RequestedHostName`** `(void);`

This method returns the name of the host that was requested. There are several different ways this is obtained depending on the structure of the request. On failure, NULL is returned.

`ResponseHeader`

`void` **`HttpdRequest::ResponseHeader`** `(const HttpdResponse &`*`resp`*`);`

This method calls the `Httpd::ResponseHeader` to deliver the response headers to the client. Request processing code should call this method rather than calling the method in `Httpd` directly.

`NeedHeaders`

`bool` **`HttpdRequest::NeedHeaders`** `(void);`

Indicates whether the current request requires HTTP headers to be printed within the response.

Returns true if headers are needed, false if not.

`ResponseBody`

`void` **`HttpdRequest::ResponseBody`** `(const HttpdResponseMsg &`*`response`*`, const char *`*`p_url`*`);`

This method calls the `Httpd::ResponseBody` to deliver the response headers to the client. Request processing code should call this method rather than calling the method in `Httpd` directly.

`Respond`

`void` **`HttpdRequest::Respond`** `(int` *`status`*`);`

This method generally encapsulates the HttpdRequest::ResponseHeader and HttpdRequest::ResponseBody methods, and is used as a general error handling mechanism when HTTP errors and associated human-readable messages are to be sent to a client. An HTTP status code is provided in *`status`*. See Supported HTTP Response Codes for possible values.

`CustomResponse`

`void` **`HttpdRequest::CustomResponse`** `(int` *`status`*`);`

This method is similar to HttpdRequest::Respond except it allows custom response bodies for negative response codes.

`Redirect`

`void` **`HttpdRequest::Redirect`** `(int` *`status`*`, const char *`*`p_location`*`);`

Causes an HTTP redirect to be sent to the client. *`p_location`* should point to a string containing the path or URI the client should be redirected to. The redirect is sent with the HTTP status code *`status`*; see Supported HTTP Response Codes for possible values.

Important

`p_location` should always be normalized (using HttpdUtilities::Normalize).

The `HttpdRedirector` class provides a general handler for redirecting portions of Seminole's URL space, and would normally encapsulate `Redirect()`. It is suggested that `HttpdRedirector` be used when possible or sensible.

`RedirectWithQuery`

`void` **`HttpdRequest::RedirectWithQuery`** `(int` *`status`*`, const char *`*`p_location`*`);`

In some cases, a redirect should include the incoming query string. In particular, some cases in `HttpdFileHandler` require a redirect to add `/` separator characters. In those cases, the query string (if any) needs to be preserved.

This method behaves just like `HttpdRequest::Redirect` with the exception of appending the query string if one exists to the outgoing URL

`NoCacheHeaders`

`int` **`HttpdRequest::NoCacheHeaders`** `(HttpdWritable *`*`p_out`*`);`

This method writes the appropriate HTTP headers to prevent caching to the stream *`p_out`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

`QueueHeader`

`void` **`HttpdRequest::QueueHeader`** `(const char *`*`p_header`*`, const char *`*`p_value`*`);`

This method queues the specified header and associated value for transmission during the response phase.

Note

This method is only available if the `INC_QUEUED_HEADERS` option is enabled.

Public Data

`HttpdRequest` contains no publically accessible data members. Methods are provided to access relevant information when necessary.

`HttpdHandler` Reference

Introduction

Subclasses of `HttpdHandler` are installed in a webserver instance to handle requests directed at a particular URL path prefix. `HttpdHandler` is an abstract class and must be subclassed.

Typically subclasses of `HttpdHandler` will contain most of the application specific behaviors. `HttpdHandler` is very low level. The `HttpdFileHandler` can be used for more traditional (file oriented) behavior.

Protected Data

`mpPrefix`

```
const char *mpPrefix
```

This member points to the URL prefix that this handler will attempt to handle.

This member should be initialized by subclasses before this object is installed in an `Httpd` instance. Preferrably it should be set in the constructor of the superclass.

Protected Methods

`IsMe`

```
char *HttpdHandler::IsMe (HttpdRequest *p_req);
```

This method determines if *`p_req`* contains the prefix specified by `mpPrefix`. If so the portion of the URL after the prefix is returned. If not NULL is returned.

Handlers should call this method (or the `IsMyPath` method) in their implementation of `Handle` to determine if the handler may be responsible for the request.

Note

The return value is not unescaped or processed in any form. Typically the returned string should be passed to `HttpdUtilities::UriDecode` to obtain a usable path name.

`IsMyPath`

```
char *HttpdHandler::IsMyPath (HttpdRequest *p_req);
```

This method determines if *`p_req`* contains the path specified by `mpPrefix`. If so the portion of the URL after the path prefix is returned. If not NULL is returned.

Handlers should call this method (or the `IsMe` method) in their implementation of `Handle` to determine if the handler may be responsible for the request.

Note

The return value is not unescaped or processed in any form. Typically the returned string should be passed to `HttpdUtilities::UriDecode` to obtain a usable path name.

Public Methods

`Prefix`

```
const char *HttpdHandler::Prefix (void);
```

This method returns the value of the protected data member `mpPrefix`.

`Handle`

```
bool HttpdHandler::Handle (HttpdRequest *p_req);
```

This pure virtual method is called by `Httpd` objects to determine if the request should be handled by this handler. If the request is handled then `true` should be returned. If the request is not handled then `false` should be returned.

Implementations of this method should call either `IsMe` or `IsMyPath` to determine if the handler is even appropriate. Although implementations are free to base the decision to handle the request upon other factors as well.

Note

It is important to remember that with multi-threading this method may be called from multiple threads simultaneously. Therefore any data memebers of the handler should be considered to be global data and must be properly prepared for concurrent access.

`HttpdResponseMsg` Reference

Introduction

Since the HTTP standard has evolved over a long period of time, the possible responses and actions in a given transaction can depend on the protocol version and client capabilities. However, none of this needs to matter for the purpose of writing new handlers or version-independent extensions. `HttpdResponseMsg`, along with `HttpdRequest`, serves to encapsulate and hide the details of translating general protocol actions (such as generating a redirect) into specific network transmissions.

The typical usage scenario for this class is to use the `HttpdResponseMsg::Find` static method with a response code and protocol version to get the correct response message to return. Normally this is not necessary because the `HttpdRequest::Respond` does these actions. However, if a custom error response with custom headers is desired then this class should be used to get the appropriate response message. Although a better approach is to use the `HttpdRequest::QueueHeader` method if the INC_QUEUED_HEADERS feature is enabled.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`Find` (by response code)

```
const HttpdResponseMsg &HttpdResponseMsg::Find (int resp);
```

Given HTTP status code *`resp`*, return a reference to an appropriate `HttpdResponseMsg` object. If the given status code is unknown or invalid (i.e. it is not listed in Supported HTTP Response Codes), then a generic “server error” response object will be returned instead.

`Find` (by response code and protocol version)

`const HttpdResponseMsg &`**`HttpdResponseMsg::Find`**`(int `*`resp`*`, HttpdProtocolVersion `*`vers`*`);`

Given HTTP status code *`resp`* and client protocol version *`vers`*, return a reference to an appropriate `HttpdResponseMsg` object. If the precise status code given is not supported at protocol version *`vers`*, a more general response in the same status category may be selected instead. If the given status code is unknown or invalid (i.e. it is not listed in Supported HTTP Response Codes), then a generic “server error” response object will be returned instead.

Public Data

`mStatus`

`const int mStatus`

The HTTP status code associated with this response (see Supported HTTP Response Codes for possible values).

`mpName`

`const char *mpName`

A human-readable translation of the associated response's `mStatus` value; for example, `200` (`HTTPD_RESP_OK`) means “OK”.

`mpDescription`

`const char *mpDescription`

A more descriptive human-readable explanation of this response; for example, a redirection might result in “The document has moved.\n”.

`mVersion`

`HttpdProtocolVersion mVersion`

The minimum HTTP protocol version required to understand this response.

`HttpdRedirector` Reference

Introduction

`HttpdRedirector` provides a simple handler interface for the purpose of sending HTTP redirect responses to clients. It translates any given request within its designated URL space to a new URI, possibly on a different host.

A common use for this class is to provide several easy to remember top-level URL paths for users but redirect those into the appropriate areas of a web interface.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`HttpdRedirector`

`HttpdRedirector` `(const char *`*`p_prefix`*`, const char *`*`p_newuri`*`, int `*`status`*`);`

`HttpdRedirector`'s constructor is documented here, because of its role as the sole interface to the class' functionality. After object creation, redirectors are installed into the Seminole handler chain at runtime just as any other handler.

The URL prefix for which this redirector will handle requests is provided in *`p_prefix`*. The new URI to which clients are redirected is provided in *`p_newuri`*. If a local pathname is provided (i.e. the redirect is local to this port and host), then it must be normalized, meaning that it should not contain relative path references (such as “.” or “..”), and it must end with a forward slash (“/”). An appropriate HTTP status code is given by *`status`* (see Supported HTTP Response Codes for possible values).

Public Data

`mpNewUri`

`char *mpNewUri`;

A pointer to a string representing the URI to which requests for this `HttpdRedirector` object are to be redirected. If the new destination is a pathname rather than a complete URI, the implied server is the current one.

`mStatusCode`

`int mStatusCode`;

Contains the desired HTTP status code to be used when issuing a redirect to a client (see Supported HTTP Response Codes for possible values).

`HttpdFileHandler` Reference

Introduction

`HttpdFileHandler` provides all of the machinery necessary to serve a portion of a `HttpdFileSystem` to HTTP clients.

In addition to being a self-contained handler, `HttpdFileHandler` can also serve as the base for a variety of extensions by subclassing. Only the methods that should be overridden in subclasses are documented here. The rest of the methods are considered internal and should not be overridden.

The architecture of `HttpdFileHandler` is similar to the way Seminole handles all requests: a state object is allocated for the request and released at the end of processing. In the case of `HttpdFileHandler` the state information is stored in a structure defined within the class called RequestState. A pointer to the RequestState object is also placed in the `mpData` field of the request object.

The RequestState structure also contains an unused member called `mpData` that can be used by subclasses for tracking additional state. The `Cleanup` phase can be overridden to release resources associated with this member.

In addition to the `mpData` member, the RequestState structure contains a pointer to the handler object (`mpHandler`). This can be useful if `HttpdFileHandler` is subclassed. This pointer can be casted and then used to accesses additional data.

If the INC_BYTERANGE_SUPPORT option is defined then the RequestState structure also contains two members dictating the byte range of the response: `mByteOffsetStart` and `mByteOffsetEnd`. These are byte positions relative to the start of the file.

Each request has a certain life cycle that is divided into phases. Depending on the outcome of the previous phase request processing may terminate early. The default handling of each phase can be overridden in subclasses to either perform extra work or abort the request with no further processing.

Most of the methods within this class are passed a reference to the current RequestState structure. It is important to keep in mind that not all members are initialized when some methods may be called. Rather, some members are garbage until a certain phase of request processing completes.

Table 2.4. HttpdFileHandler Request Processing Phases

Method	Meaning
`CheckMethod`	This phase initializes the `mMethod` member of RequestState.
`ValidMethod`	This phase determines if the HTTP method is supported by the handler.
`TranslateUri`	This phase processes the URI provided in the `mpReqPath` RequestState member and initializes the `mpDecodedUri` and `mpFilePath` members.
`ProcessUri`	Given the results of the `TranslateUri` phase this method is expected to decide on an appropriate target file and load `mFileInfo` member of RequestState.
`DoFileInfo`	The `mFileInfo` member is analyzed and appropriate action should be taken. This method (by default) detects directories and applies special handling to them.
`DoFile`	At this point `mFileInfo` points to a target that is a file that needs to be delivered. This phase is where the *MIME* type is analyzed and an appropriate response must be generated. Template processing and other server-side content translations should be done here.
`Cleanup`	This phase releases storage allocated to `mpReqPath` and `mpFilePath`. It can be overridden to optionally release resources associated with `mpData`.

Directory Processing

Directory processing accomplishes two major goals. First, URI's without trailing slashes are fixed up (handled by the `DoDirectory` method). Second, a response for the directory request is sent back.

When a request is made for a directory object, the `DoFileInfo` method calls `SendIndexFile` which checks the directory for a file called `index.html`. If this file exists it is sent out as a result of the directory request.

If no `index.html` exists and the INC_DIRECTORY_LISTS option is enabled an HTML listing of the directory is generated at runtime.

Otherwise, if there is no `index.html` and INC_DIRECTORY_LISTS is not enabled a `404` (not found) response is generated.

The bulk of the directory processing code is actually the optional listing generator, which is handled with the following methods:

- `SizeToString`
- `ListDirectory`
- `ListEntry`
- `ListParentDir`
- `DirectoryBody`

Character sets & Encodings

The HTTP protocol does not designate a standard character set for textual content. In fact many different possible character sets may be specified via the `charset` extension to the `Content-Type` *MIME* header.

If the `charset` attribute is set for the file info this value is sent along with the `Content-Type` header by the `HttpdFileHandler::SendContentType` method.

Public Methods

`HttpdFileHandler`

`HttpdFileHandler::HttpdFileHandler` `(HttpdFileSystem *`*`p_filesys`*`, const char *`*`p_root`*` = HttpdUtilities::mRoot, const char *`*`p_prefix`*` = HttpdUtilities::mRoot, HttpdUint8 `*`flags`*` = 0);`

This constructor initializes the object. The parameter *`p_prefix`* is used to determine which URI strings are associated with this handler. The other two parameters are used to specify the source of files. If *`p_root`* is anything other than “/” then it must be a valid path to which all requests are relative to.

If *`flags`* includes `HttpdFileHandler::SUPPORTS_POST` then the `ValidMethod` will allow `POST` requests through.

`FileSystem` (getter)

`HttpdFileSystem *`**`HttpdFileHandler::FileSystem`**` (void);`

Returns the file system provider assigned to the handler during construction.

Protected Methods

Note

These methods typically constitute a major phase of request processing and can be overridden in subclasses for additional processing.

`CheckMethod`

`void` **`HttpdFileHandler::CheckMethod`** `(RequestState &`*`state`*`);`

This method performs the `CheckMethod` phase. The following members of *`state`* are initialized during this phase:

- `mHandled` (to the value `true`)
- `mpRequest` (to the current `HttpdRequest` object)
- `mpReqPath` (to the requested URI)
- `mpData` (to `NULL`)

This method is required to initialize the `mMethod` member of *`state`*. It should be set to one of UNKNOWN_METHOD, HEAD_METHOD, GET_METHOD, or POST_METHOD.

`ValidMethod`

`int` **`HttpdFileHandler::ValidMethod`** `(RequestState &`*`state`*`);`

This method returns 0 if `mMethod` is a supported method. If the method is not supported then an appropriate HTTP error status should be returned, such as HTTPD_RESP_METHOD_NOT_IMPL. The following members of *`state`* are initialized during this phase:

- `mHandled` (to the value `true`)
- `mpRequest` (to the current `HttpdRequest` object)
- `mpReqPath` (to the requested URI)
- `mMethod` (to the method of the request)

`TranslateUri`

`bool` **`HttpdFileHandler::TranslateUri`** `(RequestState &`*`state`*`);`

This method performs the `TranslateUri` phase. This phase occurs after the `CheckMethod` phase. All of the members of *`state`* initialized before and during `CheckMethod` are valid for this phase.

This method is required to initialize the `mpDecodedUri` and `mpFilePath` members of *`state`*. These values should be initialized to point to storage allocated from HttpdOpSys::Malloc because `Cleanup` will release them using HttpdOpSys::Free.

If this method returns false processing is aborted. The `Cleanup` phase is still executed, however. It is therefore always necessary to initialize `mpDecodedUri` and `mpFilePath`. If they do not point to valid storage then they should be set to `NULL`.

If processing should be passed on to other handlers (and no error response was sent out during this phase) then `mHandled` can be set to `false`.

If this method returns true then processing continues with the `ProcessUri` phase.

`ProcessUri`

void **HttpdFileHandler::ProcessUri** (RequestState &*state*);

This method performs the `ProcessUri` phase. This phase occurs after the `TranslateUri` phase. All of the members of *`state`* initialized before and during `TranslateUri` are valid for this phase.

This method is required to initialize the `mFileInfo` member of *`state`*. This is typically done using the path name in `mpFilePath` in *`state`* that was generated during the `TranslateUri` phase. If `mFileInfo` is not able to be initialized then an appropriate error response should be generated.

The default implementation does not do any additional translations on `mpFilePath` as these should normally be done in the `TranslateUri` phase. This method may be used in subclasses to locate files on a different filesystem for some criteria.

`DoOptions`

int **HttpdFileHandler::DoOptions** (RequestState &*state*, HttpdStringSink &*allowed_methods*, HttpdStringSink &*headers*);

This method is called when the request is an `OPTIONS` method. Additional headers in the response should be written to *`headers`* while additional methods (separated by a comma) are written to *`allowed_methods`*.

Subclasses of `HttpdFileHandler` may override this method to add additional information to the response. Subclasses should call the implementation in `HttpdFileHandler` before writing additional data.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if the INC_OPTIONS_METHOD feature is enabled.

`DoFileInfo`

void **HttpdFileHandler::DoFileInfo** (RequestState &*state*);

This method performs the `DoFileInfo` phase. This phase does the first analysis of the `mFileInfo` member of *`state`* to determine the appropriate way to continue processing the request.

The critical test performed here is that of directory objects. If `mFileInfo` is determined to be a directory object and not a file object the request is passed off to directory processing.

`DoFile`

void **HttpdFileHandler::DoFile** (RequestState &*state*);

This method performs the `DoFile` phase. At this point we know we have a valid target file that was selected by the previous phases. This phase allows the remainder of request processing to proceed with that assumption.

This phase analyzes the remaining attributes of the `mFileInfo` member of *`state`* to determine an appropriate response.

The most common tests to be performed are on the *MIME* type of the file. The default implementation checks for a *MIME* type of `x-server-internal/private` and generates an error for files of this type. Otherwise the request is passed on to the `SendFile` method.

Cleanup

```
void HttpdFileHandler::Cleanup (RequestState &state);
```

This method performs the `Cleanup` phase. Request processing is about to terminate and any resources allocated during the processing of this request should be freed.

SendFile

```
void HttpdFileHandler::SendFile (RequestState &state);
```

This method takes a RequestState object that has completed the `DoFile` phase and sends out the appropriate response to the client. Depending on the request, headers are optionally generated. The file is opened and pushed to the socket of the `HttpdRequest` object.

NeedToSendOut

```
bool HttpdFileHandler::NeedToSendOut (RequestState &state);
```

This method analyzes the `If-Modified-Since` header to determine if content needs to be sent out at all. If the content must be sent true is returned. Otherwise, false is returned and a HTTP 304 response can be sent instead.

Note

This method only exists if the INC_MODIFIED_SINCE option is enabled.

ResultHeader

```
bool HttpdFileHandler::ResultHeader (RequestState &state, int resp);
```

This method generates an appropriate header and response message. The *`resp`* is the status code for which the response is being generated.

The value false is returned on error. Otherwise true is returned.

SendIndexFile

```
bool HttpdFileHandler::SendIndexFile (RequestState &state);
```

This method is called from the default implementation of `DoFileInfo` for a directory. It attempts to locate a file called `index.html` inside the specified directory. If this file exists its contents are sent as response.

If no response was sent this method returns false and request processing should continue. Otherwise true is returned if the request was handled.

DoDirectory

void **HttpdFileHandler::DoDirectory** (RequestState &*state*);

This method is used to fix up URI's that point to a directory name without a trailing slash character (“/”). In addition, this is where the response to a directory request is generated for directories that do not contain an `index.html`.

SendContentType

int **HttpdFileHandler::SendContentType** (RequestState &*state*);

This method sends a `Content-Type` header to the request object in the *`state`* object. The value of the header is derived from the `mInfo` member of *`state`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

FullRange

void **HttpdFileHandler::FullRange** (RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. This method adjusts the current byte range to be inclusive of the entire file. It can be called any time after the `CheckByteRanges` method is called to respond with complete content and a status code of `HTTPD_RESP_OK`.

CheckByteRanges

void **HttpdFileHandler::CheckByteRanges** (RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. This method analyzes the HTTP request headers to see if a partial range request is desired. If so the range in the *`state`* object is updated with the new range. The range is not validated at this point therefore upon return of this method the byte range within the RequestState may be invalid.

IsRangeASubset

bool **HttpdFileHandler::IsRangeASubset** (const RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. If the range in the RequestState object does not cover the entire entity body then true is returned. Otherwise the request is for a full entity body and false is returned.

ValidRange

bool **HttpdFileHandler::ValidRange** (const RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. A byte range is considered valid if it is in a forward direction (the ending position is greater or equal to the starting position) and the range is within the entity body length. If the range is valid then true is returned. For invalid ranges false is returned.

InvalidValidRangeResponse

void **HttpdFileHandler::InvalidValidRangeResponse** (const RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. This method returns a `HTTPD_RESP_RANGE` with the appropriate `Content-Range:` header value. No further response should be issued for the request, either by calling the `Respond` method of the request or by calling the `ResultHeader` method of the file handler.

CheckForRangeCondition

void **HttpdFileHandler::CheckForRangeCondition** (const RequestState &*state*);

This method is only present if INC_BYTERANGE_SUPPORT is enabled. This method is called by the `CheckByteRanges`. It processes conditional range requests (indicated by the presence of a `If-Range:` header line in the request) and invalidates the subrange if necessary.

HttpdRequestForwarder Reference

Introduction

Sometimes it is necessary to declare two different `Httpd` objects that both should share the same set of handlers. For example, if the same set of handlers should be accessible via either SSL or TCP. A handler can only be installed in one `Httpd` object at a time.

The `HttpdRequestForwarder` class is a handler that forwards requests to the handlers installed in another `Httpd` object. It cleanly solves the problem of having more than one `Httpd` object where the same handling must be performed across objects.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Public Methods

HttpdRequestForwarder

HttpdRequestForwarder::HttpdRequestForwarder (Httpd **p_server*);

Initialize the handler to send all requests to *`p_server`*.

HttpdUrl Reference

Introduction

In many cases Seminole parses a URL using open-coded logic — for efficiency. Although when necessary `HttpdUrl` can be used to decompose a URL easily.

The `HttpdUrl` can be used repeatedly without recreating the object. However it is important to remember that each component string will be invalidated when a new URL is parsed with the object.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`Parse`

```
int HttpdUrl::Parse (const char *p_url);
```

This method parses the URL. If successful `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Cleanup`

```
void HttpdUrl::Cleanup (void);
```

This function releases any stored memory from a previously parsed URL. It is safe to call this function at any time; even if there is no previously parsed URL. You should call this method if you are sure that the parsed URL components are no longer needed. This releases the allocated memory for use elsewhere. It is not necessary to call this method; it is strictly a space optimization.

`Path`

```
const char *HttpdUrl::Path (void);
```

Returns a pointer to the path component of the URL. This method will never return NULL if a URL has been parsed.

`Host`

```
const char *HttpdUrl::Host (void);
```

Returns a pointer to the host name component of the URL. This method will never return NULL if a URL has been parsed.

`Scheme`

```
const char *HttpdUrl::Scheme (void);
```

Returns a pointer to the scheme component of the URL. This method will never return NULL if a URL has been parsed.

`Transport`

```
const char *HttpdUrl::Transport (void);
```

Returns a pointer to the transport name used for this URL. If the INC_MULTIPLE_TRANSPORTS feature is not enabled then this method will always return NULL.

Query

`const char *`**`HttpdUrl::Query`**` (void);`

Returns a pointer to the query string of this URL If no query string is present then NULL is returned.

Port

`HttpdIpPort `**`HttpdUrl::Port`**` (void);`

This method returns the port the URL references. If no port is specified then the appropriate port for the scheme is returned.

StandardPort

`bool `**`HttpdUrl::StandardPort`**` (void);`

This method returns true if the port is the default for the scheme of the URL and need not be specified. If false is returned then the port is special and needs to be specified for this scheme.

Url

`const char *`**`HttpdUrl::Url`**` (void);`

Returns a pointer to the URL that was parsed. This method will never return NULL if a URL has been parsed.

Authority

`const char *`**`HttpdUrl::Authority`**` (void);`

Returns a pointer to the authority information of the URL. If the URL does not contain any authority information then NULL is returned.

IsRelative

`bool `**`HttpdUrl::IsRelative`**` (const char *`*`p_relative`*`);`

This method determines if *`p_relative`* is a component relative to this URL. If true is returned then *`p_relative`* can be converted to an absolute URL using the `Relative` method, described below.

Relative

`char *`**`HttpdUrl::Relative`**` (const char *`*`p_relative`*`);`

This method returns an absolute URL that is the current URL adjusted by *`p_relative`*. NULL is returned upon failure. If successful it is the caller's responsibility to free the returned string (using HttpdOpSys::Free).

IsSecure

`bool `**`HttpdUrl::IsSecure`**` (void);`

This method determines if the URL is “secure”. For example a URL is considered secure if the transport for the scheme is SSL.

HostNameMatchesHeader

`bool` **`HttpdUrl::HostNameMatchesHeader`** `(const char *`*`p_host_header`*`);`

This method determines if the hostname portion of the `Host:` header matches the host used in the URL

SeparatePath

`char **`**`HttpdUrl::SeparatePath`** `(const char *`*`p_path`*`);`

This static method separates *`p_path`* into an array of components. If successful it returns a pointer to an array of strings. Each string is the decoded path component (decoded). The returned array is terminated by a NULL entry. If unsuccessful then NULL is returned.

It is the responsability of the caller to release the memory allocated by the array by calling the `FreePathList` method.

FreePathList

`void` **`HttpdUrl::FreePathList`** `(char **`*`pp_path`*`);`

This static method frees the memory allocated by the `SeparatePath` method.

TrimLastEntry

`void` **`HttpdUrl::TrimLastEntry`** `(char **`*`pp_path`*`);`

This static method removes the very last entry (if one exists) of the provided path array. Typically this final component is determined to be a file name while all of the other components are directories.

PathIsSubset

`bool` **`HttpdUrl::PathIsSubset`** `(const char *const *`*`pp_base`*`, const char *const *`*`pp_path`*`);`

This static method determines if *`pp_base`* contains *`pp_path`*.

`HttpdCgiParameter` Reference

Introduction

Serving anything more complex than static documents via HTTP typically requires use of the Common Gateway Interface, or CGI. CGI parameters are passed by encoding them in URI strings, or through use of the `POST` method. Seminole provides common mechanisms to decode and parse CGI parameters within the `HttpdCgiParameter` class.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`ParseUriString`

`HttpdCgiParameter *`**`HttpdCgiParameter::ParseUriString`**` (const char *`*`p_query`*`);`

Given a set of URL-encoded CGI parameters in *`p_query`*, returns a pointer to a `HttpdCgiParameter` object containing the decoded parameters. Each subsequent parameter can be obtained by following each `HttpdCgiParameter`'s `mpNext` pointer. NULL is returned if there are no encoded parameters in *`p_query`*.

The caller is expected to free the returned pointer using HttpdCgiParameter::FreeList, if the call is successful.

`ParsePostData`

`HttpdCgiParameter *`**`HttpdCgiParameter::ParsePostData`**` (HttpdRequest *`*`p_request`*`);`

Parse data sent via the HTTP `POST` method, in the request *`p_request`*. A pointer to a `HttpdCgiParameter` object containing the decoded parameters is returned. Each subsequent parameter can be obtained by following each `HttpdCgiParameter`'s `mpNext` pointer. NULL if none are found or there is an error in processing.

The caller is expected to free the returned pointer using HttpdCgiParameter::FreeList, if the call is successful.

`ParseFormData`

`HttpdCgiParameter *`**`HttpdCgiParameter::ParseFormData`**` (HttpdRequest *`*`p_request`*`);`

This method parses all form data from the request. Pairs from the query string appear first in the resultant list followed by any posted form data (if the method is `POST`).

The caller is expected to free the returned pointer using HttpdCgiParameter::FreeList, if the call is successful.

`ParseString`

`HttpdCgiParameter *`**`HttpdCgiParameter::ParseString`**` (char *`*`p_attr`*`);`

Given a set of URL-encoded CGI parameters in *`p_attr`*, returns a pointer to a `HttpdCgiParameter` object containing the decoded parameters. Each subsequent parameter can be obtained by following each `HttpdCgiParameter`'s `mpNext` pointer. NULL is returned if there are no encoded parameters.

The caller is expected to free the returned pointer using HttpdCgiParameter::FreeList, if the call is successful.

Important

The attribute string may only be parsed once. For efficiency reasons the parsing modifies the string in place. It is important that the string not be parsed again. If the parsing must be performed more than once then a copy of the string should be made (using `HttpdUtilities::SaveString`).

FreeList

`void HttpdCgiParameter::FreeList (HttpdCgiParameter *p_list);`

Destroys a `HttpdCgiParameter` object, and frees its resources. CGI handlers should call this method when finished with their processing.

Find

`const char *HttpdCgiParameter::Find (const char *p_name);`

Find the named parameter from the current node forward. Typically this method is called from the first node in the list but it can be used to walk a list with multiple parameters of the same name.

On success this method returns the value of the found node. NULL is returned on error.

FindNode

`HttpdCgiParameter *HttpdCgiParameter::FindNode (const char *p_name);`

Find the named parameter from the current node forward. Typically this method is called from the first node in the list but it can be used to walk a list with multiple parameters of the same name.

On success this method returns the address of the found node. NULL is returned on error.

Lookup

`HttpdCgiParameter *HttpdCgiParameter::Lookup (HttpdCgiParameter *p_list, const char *p_name);`

Find the named parameter in the list.

On success this method returns the address of the found node. NULL is returned on error. If *`p_list`* is NULL then the list is considered empty and NULL is always returned.

CompareLists

`bool HttpdCgiParameter::CompareLists (const HttpdCgiParameter *p_list_a, const HttpdCgiParameter *p_list_b);`

This static method determines if the contents pointed to by *`p_list_a`* are identical (in both value and order) to the nodes pointed to by *`p_list_b`*.

If the two lists are identical true is returned. If they are not identical then false is returned.

CopyList

`bool HttpdCgiParameter::CopyList (HttpdCgiParameter *&p_dest, const HttpdCgiParameter *p_src);`

This static method copies all of the nodes pointed to by *`p_src`* into a new list with the first node pointed to by *`p_dest`*. If successful true is returned. If there is insufficient memory then false is returned.

Public Data

mPair

`HttpdPair mPair`;

The parameter name and value of this `HttpdCgiParameter`.

mpNext

`HttpdCgiParameter *mpNext`;

A pointer to the next `HttpdCgiParameter` object on the list, or NULL if this object is the last member of the list.

HttpdCgiHash Reference

Introduction

The HttpdCgiParameter class is designed to be generic. The parameters are stored in a linked list to preserve order and allow for duplicate parameters. The intention was to allow anything from a remote procedure call interface to a real-time data stream to be transported using CGI parameter encoding. The cost for that flexibility is speed. Searching for a particular parameter by name linearly scans the entire parameter list.

`HttpdCgiHash` re-orders the nodes of a `HttpdCgiParameter` list to make searching for parameters by name significantly faster. This class takes advantage of the fact that after parsing CGI parameters are stored in a singly linked list. It is a very easy transform to convert the linear list to an open-chained *hash table*. The nodes are not copied, they are re-linked in place into the appropriate bucket.

The `HttpdCgiHash` also behaves as an array of pointer to `HttpdCgiParameter` objects. The array is always HTTPD_CGI_HASH_SIZE (defined by the `CGI_HASH_SIZE` build parameter) elements in size. This behavior allows easy iteration of a collection of CGI parameters as long as the order is not important. For example:

```
for(size_t i = 0; i < HTTPD_CGI_HASH_SIZE; i++)
 for(HttpdCgiParameter *p_param = hash[i];
     p_param != NULL;
     p_param = p_param->mpNext)
```

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdCgiHash

`HttpdCgiHash::HttpdCgiHash` `(void);`

The constructor of `HttpdCgiHash` initializes the object. After initialization the hash is empty. Nodes can be added with the `Append` method.

~HttpdCgiHash

`HttpdCgiHash::~HttpdCgiHash` `(void);`

It is important to understand that any parameter nodes contained in the hash are owned by the hash. When the `HttpdCgiHash` instance is destroyed, so are all the nodes contained in the hash.

Append

`void` **`HttpdCgiHash::Append`** `(HttpdCgiParameter *`*`p_list`*`);`

The CGI parameters contained in *`p_list`* are appended to the hash. After being appended, the nodes in *`p_list`* are owned by the hash table. They should not be released (via `HttpdCgiParameter::FreeList`) or manipulated in any way.

It is possible to append several sets of parameters to the hash table. In this case the hash table contains the union of all of the appended sets of parameters. In the cases of duplicate parameter names all of the nodes are stored but in an unpredictable order.

Find

`HttpdCgiParameter` **`HttpdCgiHash::Find`** `(const char *`*`p_name`*`);`

Find the named parameter in the hash. If a parameter by the name of *`p_name`* exists then the address of the first parameter by that name in the chain is returned. If no parameter by that name exists, NULL is returned.

If more than one parameter with the name *`p_name`* exists in the hash, the `Find` method can be applied to the returned pointer to find additional nodes with the same name.

Remove

`void` **`HttpdCgiHash::Remove`** `(HttpdCgiParameter *`*`p_obj`*`);`

Remove *`p_obj`* from the table. The object must be a member of the hash or the behavior is undefined. After removal, the object must be freed using `HttpdCgiParameter::FreeList` when it is no longer needed.

`HttpdMultipartCgiParser` Reference

Introduction

To support HTML forms with file upload a special *MIME* encoding of `multipart/form-data` is used with the `POST` method. This class parses `POST` request data in this format. This class may also be subclassed to handle incoming data without storing it in memory or to handle binary data that does not store well as a string.

For low-memory devices this feature is very important. Incoming data can be processed as it is received by substituting a customized subclass of HttpdWritable for a particular parameter.

Alternatively data can be processed by “pulling” data directly from an instance of `HttpdBoundaryReader`.

Subclassing Using a Push Model

When using a push model the `OpenDestination` and `CloseDestination` methods are typically modified to special case certain parameters. Let us assume we are loading a binary file into a `HttpdStringSink` for later firmware updates.

```
class Update_MultipartParser : public HttpdMultipartCgiParser
{
  HttpdStringSink  mFirmware;
  bool             mFirmwareOpen;

  protected:
    virtual int OpenDestination(State &state, HttpdWritable *&p_dest);
    virtual int CloseDestination(State &state,
                                 HttpdWritable *p_dest,
                                 int rc);
};
```

To open the destination it is necessary to examine `state` and determine if this is part requires special processing:

```
int Update_MultipartParser::OpenDestination
 (
   HttpdMultipartCgiParser::State   &state,
   HttpdWritable                   *&p_dest
 )
{
  if (strcmp(state.mAttributes.mpName, "new_fw_file") == 0)
  {
    // If the firmware destination is open then the client is confused...
    // The constructor of this class sets this to false. So just ignore
    // this data from the confused client -- our firmware files are
    // checksummed anyhow.
    if (mFirmwareOpen)
    {
      p_dest = HttpdNullSink::Null();
      return (0);
    }

    // Clear out any previous content that may be in there.
    HttpdOpSys::Free(mFirmware.TakeBuffer());

    // Here we are!
    p_dest        = &mFirmware;
    mFirmwareOpen = true;
    return (0);
  }

  // For other fields -- handle normally.
  return (HttpdMultipartCgiParser::OpenDestination(state, p_dest));
}
```

The close case is similar but there are a few additional things to check for:

```
int Update_MultipartParser::CloseDestination
 (
   HttpdMultipartCgiParser::State   &state,
   HttpdWritable                   *&p_dest,
   int                               rc
 )
{
  // If this is our special-cased part:
  if (p_dest == &mFirmware)
  {
    // If successful.
    if (rc == 0)
    {
      // Do whatever application specific processing is needed.
      …
    }
    else // Clear out any partial data on error.
      HttpdOpSys::Free(mFirmware.TakeBuffer());

    // Pass the status up to the higher layers.
    return (rc);
  }

  // Call the superclass method for default processing.
  return (HttpdMultipartCgiParser::CloseDestination(state, p_dest, rc));
}
```

Of course it is possible to use any object that implements the HttpdWritable interface; even files and sockets are legitimate targets.

Subclassing Using a Pull Model

It isn't always easy or convenient to process data by getting calls to a method in a class. For these cases a different approach can be taken. Consider the idea of loading an FPGA from a file upload:

```
class FPGA_MultipartParser : public HttpdMultipartCgiParser
{
  protected:
    virtual int HandlePart(State &state, HttpdBoundaryReader &reader);
};
```

For a “pull” model only one method must be subclassed. Neither mode is mutually exclusive. The default implementation of `HandlePart` is what calls `OpenDestination` and `CloseDestination`. Therefore it is possible to override all three methods and handle each named part differently.

However, with the pull-only approach above `HandlePart` is implemented as follows:

```
int FPGA_MultipartParser::HandlePart
 (
   HttpdMultipartCgiParser::State  &state,
   HttpdBoundaryReader             &reader
 )
 {
   // Is this our special part?
   if (strcmp(state.mAttributes.mpName, "fpga_image") == 0)
   {
     int rc;

     for(;;)
     {
       const void   *p_buffer;
       size_t        len;

       // Read a block of data. Use the normal timeout used for other
       // CGI processing. We can of course us a different timeout if
       // necessary here.
       rc = reader.Read(p_buffer, len, HTTPD_CGI_TIMEOUT);
       if (rc == HttpdBoundaryReader::HTTPD_MIME_BOUNDARY)
         return (0); // Nothing left to read.
       else if (rc != 0)
         break; // Error.

       // Process the buffer of len bytes here.
       …
     }

     return (rc);
   }

   // Otherwise perform the normal processing.
   return (HttpdMultipartCgiParser::HandlePart(state, reader));
 }
```

Here the data can be processed in place without buffering it up or wrapping up processing into a self-contained object.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdMultipartCgiParser

HttpdMultipartCgiParser::HttpdMultipartCgiParser (HttpdRequest **p_request*);

This constructor initializes the parser and associates it with the request `p_request`. To parse the incoming data use the `HttpdMultipartCgiParser::Parse` method.

List

```
HttpdCgiParameter *HttpdMultipartCgiParser::List (void);
```

This method gets the current parameter list. A pointer to the first node in the parameter list is returned. Unless the list is removed with `TakeList`, the list will be automatically destroyed when the `HttpdMultipartCgiParser` object is destroyed.

TakeList

```
HttpdCgiParameter *HttpdMultipartCgiParser::TakeList (void);
```

This method is similar to `HttpdMultipartCgiParser::List` except that it removes the current list of parameters from the parser object.

If the parser is re-invoked (via `Parse`) after the list is taken, it is as if the parser was newly constructed and had no parameters.

Note

It is the responsibility of the caller to free the list using HttpdCgiParameter::FreeList.

OpenDestination

```
int HttpdMultipartCgiParser::OpenDestination
(HttpdMultipartCgiParser::State &state, HttpdWritable *&p_dest);
```

This method gets a target HttpdWritable object for a parameter encoded as one part of a *MIME* multipart message. The default behavior implemented by this method is to store the parameter data into a HttpdCgiParameter node.

However, for large fields (such as files uploaded or large text areas) this method can be overridden by subclasses to provide a different object for handling the data. The returned writable object in `p_dest` could do anything, even process the data as it is received.

To identify the particular parameter the `state` object is passed to this method. This object contains all of the specifics for this particular multipart entity:

Members of HttpdMultipartCgiParser::State

Type: `HttpdMimeParser`
Name: *`mMimeParser`*
Description: Each entity in multipart *MIME* data has its own set of headers. This object is the parser used to parse those headers. Additional headers (such as `Content-Type`) can be extracted from this parser.
Type: `Attributes`
Name: *`mAttributes`*
Description: This structure contains various attributes about the current entry.

The `Attributes` structure contains the details for processing the current entry. It is defined as follows:

Members of HttpdMultipartCgiParser::Attributes

Type: char *
Name: *`mpName`*

Description: This is the name (as defined by the HTML `NAME` attribute of the `INPUT` element) of the parameter.
Type: char *
Name: *`mpFileName`*
Description: For input elements of type `FILE`, this is the client-specific name of the file.
Type: const char *
Name: *`mpContentDisposition`*
Description: This is the value of the `Content-Disposition` *MIME* header of this particular multipart entity.

An error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned. If no error is returned, it is expected that *`p_dest`* is pointed to a valid object.

CloseDestination

`int` **`HttpdMultipartCgiParser::OpenDestination`** `(HttpdMultipartCgiParser::State &`*`state`*`, HttpdWritable *`*`p_dest`*`, int `*`rc`*`);`

This method is called after all the data for a particular field is written to *`p_dest`* object obtained via OpenDestination.

Because this function performs double duty, cleaning up the resources used by *`p_dest`* and storing or processing data, the *`rc`* argument is used to indicate the success of reading the data. A non-zero value of *`rc`* indicates that the read was unsuccessful and whatever data was written to *`p_dest`* should be ignored (or undone if data was being processed as it was read). Cleanup of the resources should be performed in either case.

An error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned. If *`rc`* was non-zero, that value should be returned in place of success.

HandlePart

`int` **`HttpdMultipartCgiParser::HandlePart`** `(HttpdMultipartCgiParser::State &`*`state`*`, HttpdBoundaryReader &`*`reader`*`);`

The default implementation of this method calls `OpenDestination`, pumps the contents of the part into the opened destination, and then cleans up the destination.

For clients wishing to use the HttpdBoundaryReader interface directly (for example to use a "pull" model of processing the data) this method can be overridden.

Note

If success is returned then *`reader`* should have absorbed the boundary string. Do not return 0 if this is not the case.

Parse

`int` **`HttpdMultipartCgiParser::Parse`** `(void);`

This method invokes the parser. The multipart body is separated into entities and each one is written to an object provided by the overridable `OpenDestination` method. The default behavior of which is to append each parameter in a list of HttpdCgiParameter objects.

An error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned. Zero indicates success and that a valid parameter list can be obtained via `HttpdMultipartCgiParser::List`.

Important

The caller of this method must validate that the *MIME* type of the incoming `POST` request is in fact `multipart/form-data`.

`HttpdCgiWriter` Reference

Introduction

`HttpdCgiWriter` objects are used to generate query strings to objects that implement the HttpdWritable interface.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdCgiWriter`

`HttpdCgiWriter::HttpdCgiWriter` `(HttpdWritable *`*`p_target`*`, bool` *`compact_space`* `= false);`

Constructs a `HttpdCgiWriter` object that writes the query string to *`p_target`*. The `?` separator between the file path and the query string is not written by this class and, if desired, must be written manually before this object is constructed.

If *`compact_space`* is `true` then space characters (ASCII 0x20) are replaced with plus characters ("`+`").

`Write`

`int` **`HttpdCgiWriter::Write`** `(const char *`*`p_name`*`, const char *`*`p_value`*`);`

Append the name and value parameter to the query string being assembled in the target sink.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`WriteNode`

`int` **`HttpdCgiWriter::WriteNode`** `(const HttpdCgiParameter *`*`p_node`*`);`

Append the contents of *`p_node`* to the query string being assembled in the target sink.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`WriteList`

`int` **`HttpdCgiWriter::WriteList`** `(const HttpdCgiParameter *`*`p_list`*`);`

Append the contents of every node in *`p_list`* to the query string being assembled in the target sink.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Reset

void **HttpdCgiWriter::Reset** (void);

Reset the writer for a new string of parameters.

HttpdAttributeParser Reference

Introduction

Several extensions to the values of the *MIME* headers of an HTTP request are done using token/value pairs. These pairs typically (but not always) follow data terminated by a semicolon (`;`).

The `HttpdAttributeParser` class parses these kinds of attributes. Because most of these attributes are clauses that can be processed without much state information, the interface of this class is designed to allow easy, procedural looping of the name/value pairs.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdAttributeParser

HttpdAttributeParser::HttpdAttributeParser (const char **p_front*, const char **p_valueterm* = …);

Initialize the attribute parser to begin parsing the string pointed to by *`p_front`*.

If *`p_valueterm`* is specified then this is the set of characters that terminate an unquoted value. See `HttpdUtilities::DequoteToken` for more information.

NextAttribute

bool **HttpdAttributeParser::NextAttribute** (void);

This method should be called to obtain each successive name/value pair. After each call, the appropriate values are in the `mpKey` and `mpValue` data members.

This method returns true if there are more name/value pairs to be obtained or false if there are no more.

Public Data

mpKey

This data member contains the key (name) portion of the attribute pair. It points to internally allocated storage that is managed by the object. If the caller wishes to keep the string then this data member can be set to NULL before calling `NextAttribute` again and the buffer will not be freed. It is then the up to the new owner of this string to free it using `HttpdOpSys::Free`.

If `NextAttribute` returns true then this member will never be NULL.

mpValue

This data member contains the value portion of the attribute pair. For a standalone token this data member will be set to NULL. If not NULL it points to internally allocated storage that is managed by the object. If the caller wishes to keep the string then this data member should be set to NULL before calling `NextAttribute` again as with the `mpKey` member.

mpFront

This is a pointer within the input string. This pointer is advanced as the parse progresses. It can be used to do early termination by looking for characters or strings before calling `NextAttribute`.

HttpdCookies Reference

Introduction

This class provides a mechanism for sending `Set-Cookie` headers to clients and parsing `Cookie` headers from clients. Instances of `HttpdCookies` are associated with a particular `HttpdMimeParser` (which is part of a `HttpdRequest` object). Once associated, *cookies* associated with a request may be enumerated using a loop.

A static method, `HttpdCookies::SendCookie` is also provided to generate `Set-Cookie` headers to a client via the `HttpdDynamicOutput` class.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdCookies

`HttpdCookies::HttpdCookies` `(HttpdMimeParser &`*`mime_parser`*`);`

Initialize the cookie iterator to begin parsing the cookies associated with the *MIME* headers received in *`mime_parser`*. The `HttpdMimeParser` passed into this construct must have already completed its parsing phase (i.e. `HttpdMimeParser::Finish` must have already been called on the object).

NextCookie

`bool` **`HttpdCookies::NextCookie`** `(void);`

This method should be called to obtain each successive name/value cookie pair. After each call, the cookie name is available using the `Key` method and the value of the cookie is available using the `Value` method.

This method returns true if there are more name/value pairs to be obtained or false if there are no more.

It is important to realize that clients can send no cookies in a request. Therefore this method should always be called first (typically as the conditional of a while loop) to determine if the `Key` and `Value` methods should even be called.

Key

`const char * **HttpdCookies::Key** (void);`

This method should be called to obtain the name of the current cookie. The returned pointer is only valid for this iteration and its contents will change after the next call to `NextCookie`. This method never returns NULL.

Note

This method should not be called until `NextCookie` has been called and returned true.

Value

`const char * **HttpdCookies::Value** (void);`

This method should be called to obtain the value of the current cookie. The returned pointer is only valid for this iteration and its contents will change after the next call to `NextCookie`. This method may return NULL for value-less cookies.

Note

This method should not be called until `NextCookie` has been called and returned true.

SendCookie (Stream version)

`int **HttpdCookies::SendCookie** (HttpdWritable *p_out, const char *p_key, const char *p_value, …);`

This method generates a `Set-Cookie` header with one or more name/value pairs. The pairs are provided as a variable list of arguments. Either two valid pointers must be provided or a terminator (zero cast to a constant character pointer) can be passed.

This method is a static method and is not associated with any instances of `HttpdCookies`. The resulting header is sent to the *`p_out`* stream. When using `HttpdDynamicOutput` the *`p_out`* parameter should be obtained from the `HttpdDynamicOutput::Headers`.

It is important to remember that the terminator must always be included (even for sending a single pair):

```
HttpdCookies::SendCookie(p_out,
                         "SESSION_ID",
                         "123456",
                         (const char *)0);
```

When sending more than one header it is also important to remember that the key portion and value portion are distinct. The pairs are always presented in key then value order:

```
HttpdCookie::SendCookie(p_out,
                        "SESSION_ID",
                        "123456",
                        "USERID",
                        (const char *)user_id,
                        "SYSTEM",
                        "control",
                        (const char *)0);
```

It is also important that any pointers should be explicitly cast to constant character pointers to avoid any variable argument pitfalls.

`SendCookie` (Dynamic version)

`int` **`HttpdCookies::SendCookie`** `(HttpdDynamicOutput *`*`p_out`*`, const char *`*`p_key`*`, const char *`*`p_value`*`, …);`

This method generates a `Set-Cookie` header with one or more name/value pairs. It is called in a similar way to the overloaded version that takes a `HttpdWriteable` pointer. This version uses a `HttpdDynamicOutput` object as the target.

`HttpdAuthenticator` Reference

Introduction

HTTP provides an authentication framework that can handle multiple authentication schemes. `HttpdAuthenticator` provides a framework for authenticating requests with a minimum of programming effort. For example authentication can be performed during the `HttpdFileHandler::ProcessUri` phase of request processing.

The authentication framework is defined in a header file called `sem_auth.h`. In order to use any of these classes or methods, this header file must be included.

Note

`HttpdAuthenticator` is an abstract base class. It must be subclassed and provided with methods for getting credentials.

Public Methods

`Authenticate` (Default version)

`bool` **`HttpdAuthenticator::Authenticate`** `(HttpdRequest *`*`p_request`*`);`

This method should be called during request processing for any request which must be authenticated. If this method returns false, no further processing of the request should be performed; the correct response will be sent. If this method returns true then the request should be processed as normally.

This version of the `Authenticate` utilizes all of the enabled authentication schemes. The version below allows fine-grained control of what schemes are used and in what order they are presented.

`Authenticate` (Specific version)

bool **HttpdAuthenticator::Authenticate** (HttpdRequest **p_request*, const HttpdAuthSchemes **p_schemes*);

This method is identical to the default version of `Authenticate` (described above) except that precise control over what authentication schemes (and in what order) are used.

The array pointed to by *p_schemes* must be terminated by a value of `End`.

`Create`

int **HttpdAuthenticator::Create** (void);

This method initializes an `HttpdAuthenticator` class for use. It must be called successfully before any other methods can be accessed.

An error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned, or zero on success.

`SecureStrEqu`

bool **HttpdAuthenticator::SecureStrEqu** (const char **p_str1*, const char **p_str2*);

This static method compares *p_str1* with *p_str2*. If the strings are equal then true is returned; otherwise false is returned.

This method is more secure than `strcmp` because it defends against timing attacks. No matter how the *contents* of the strings differ the amount of CPU time this method takes to execute is constant. This implies that the time it takes to reject a valid password can not be used to guess successive characters of the correct password.

The default implementation of `ValidatePassword` calls this method to compare the provided password with the correct one. Other circumstances where timing attacks are possible should also use this method.

Note

The timing of this method is only consistent if the INC_PASSWD_BLINDING option is enabled.

The implementation of this method is tuned to use as few conditional branches as possible. Furthermore local variables are declared volatile to provide consistent behavior across compilers. However if security is a high concern then manual inspection of the generated assembly code for this method is recommended.

Protected Methods

`Realm`

void **HttpdAuthenticator::Realm** (HttpdRequest **p_request*, char **p_realm*);

Note

This method is pure virtual. It must be overridden in subclasses with the appropriate functionality.

This method is called to obtain the name of the realm for a given request. The provided buffer, *`p_realm`* is HTTPD_MAX_REALM_LENGTH bytes in length. The value of the HTTPD_MAX_REALM_LENGTH constant is controlled by the `MAX_REALM_LENGTH` build-time parameter.

`GetPassword`

`bool` **`HttpdAuthenticator::GetPassword`** `(const char *`*`p_user`*`, HttpdRequest *`*`p_request`*`, char *`*`p_buf`*`);`

Note

This method is pure virtual. It must be overridden in subclasses with the appropriate functionality.

This method is called to obtain the password of the user *`p_user`* for a given request. The provided buffer, *`p_buf`* is HTTPD_MAX_PASSWD_LENGTH bytes in length. The value of the HTTPD_MAX_PASSWD_LENGTH constant is controlled by the `MAX_PASSWD_LENGTH` build-time parameter.

If the user specified in *`p_user`* does not exist or there is an internal error getting the password then false should be returned. If *`p_buf`* is set to the correct password then true should be returned.

`ValidatePassword`

`bool` **`HttpdAuthenticator::ValidatePassword`** `(const char *`*`p_user`*`, HttpdRequest *`*`p_request`*`, const char *`*`p_provided_password`*`);`

This method is called to validate that *`p_provided_password`* is in fact a valid password for *`p_user`*. The default implementation calls `GetPassword` and compares the passwords.

Subclasses may override this method if they wish to customize the password matching behavior. For example supporting case-insensitive passwords. Another reason to override this method may be that the password can't be easily obtained and it can only be validated. For example if the password is stored as a one-way hash or backed by a RADIUS server.

If the password and username combination is not valid for *any* reason then this method should return false.

Note

This method may not be called for all authorization schemes. In particular the digest authentication scheme does not provide the password to the server. Authentication schemes where the provided password is not available call `GetPassword` directly.

Override this method only if you understand all of the consequences fully.

`DigestAuthHeader`

`void` **`HttpdAuthenticator::DigestAuthHeader`** `(HttpdRequest *`*`p_request`*`, bool` *`stale`*`);`

This method is called for an unauthorized client (for whatever reason) when digest authentication is enabled. The default behavior is to propose digest authentication by adding a `WWW-Authenticate` header for the digest method.

If this method is being called because credentials were supplied against a stale nonce then *`stale`* will be `true`.

AuthorizeDigest

`bool` **`HttpdAuthenticator::AuthorizeDigest`** `(HttpdRequest *`*`p_request`*`, const char *`*`p_resp`*`, const HttpdAuthSchemes *`*`p_schemes`*`);`

This method is called when a request includes a `WWW-Authenticate` header for the digest authentication method. It should return true if the request is authorized or false if the request was declined. The digest parameters (following the method name in the `WWW-Authenticate` header) are given to this method as *`p_resp`*.

BasicAuthHeader

`void` **`HttpdAuthenticator::BasicAuthHeader`** `(HttpdRequest *`*`p_request`*`);`

This method is called for an unauthorized client (for whatever reason) when basic authentication is enabled. The default behavior is to propose basic digest authentication by adding a `WWW-Authenticate` header for the current realm.

AuthorizeBasic

`bool` **`HttpdAuthenticator::AuthorizeBasic`** `(HttpdRequest *`*`p_request`*`, const char *`*`p_resp`*`, const HttpdAuthSchemes *`*`p_schemes`*`);`

This method is called when a request includes a `WWW-Authenticate` header for the basic authentication method. It should return true if the request is authorized or false if the request was declined. The parameters (following the method name in the `WWW-Authenticate` header) are given to this method as *`p_resp`*.

NotAuthorized

`void` **`HttpdAuthenticator::NotAuthorized`** `(HttpdRequest *`*`p_request`*`, const HttpdAuthSchemes *`*`p_schemes`*`);`

This method is whenever (and for whatever reason) a client has requested a resource that it did not present proper credentials for. The default behavior is to send a `HTTPD_RESP_UNAUTHORIZED` (`401`) response with the authentication challenges listed in *`p_schemes`*.

`HttpdSessionManager` Reference

Introduction

HTTP transactions are stateless. The `HttpdSessionManager` class maintains a collection of `HttpdSessionObject` objects. The session manager addresses the objects it manages by key string. The key string can be stored in the client as a cookie or hidden form field. `HttpdSessionManager` does not force a policy of how the key is stored on the client side.

The number of `HttpdSessionObject` that can be stored in the session manager is fixed at creation time. When a new session needs to be created and there is no room the oldest (inactive) session is purged to make room for the new one. To avoid sessions that are being used to process an active request from being deleted, session objects are reference counted.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

Create

```
int HttpdSessionManager::Create (size_t count);
```

Initialize the session manager to contain `count` session objects. This method must be called before any other methods of the object with the exception of the methods that configure background scrubbing: `MaxSessionAge`, `CycleTime`, and `ScrubbingBatchSize`.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`CycleTime` (setter)

```
void HttpdSessionManager::CycleTime (unsigned long cycle_time);
```

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. This method sets the time between scrubbing intervals (in milliseconds). During each scrubbing interval a batch (controlled by `ScrubbingBatchSize`) of sessions are examined. These two parameters control the amount of processor time devoted to scrubbing inactive sessions.

Setting this parameter to `0` disables background scrubbing for this instance of session manager. However if background scrubbing is to be disabled it must be done by calling this method with a parameter of `0` before `Create` is called. Alternatively, the default value is zero and the call to this method can be avoided.

Enabling background scrubbing is a security enhancement. The session manager is always free to eject an old session if no space can be found. However, old sessions are never timed out if background scrubbing is not enabled. This leaves open the possability of the session key being obtained and then utilized by an attacker.

`MaxSessionAge` (setter)

```
void HttpdSessionManager::MaxSessionAge (long max_age);
```

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. It sets the maximum amount of time (in seconds) that a session can live without being accessed. Once that time is exceeded the session is deleted.

Note

When INC_BACKGROUND_SESSION_PURGE is enabled and the cycle time is set to a non-zero value then this method must also be called to set the initial value before `Create` can be called.

`ScrubbingBatchSize` (setter)

```
void HttpdSessionManager::ScrubbingBatchSize (size_t batch_size);
```

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. It sets the size of a scrubbing batch. This is the number of sessions that are examined during a scrubbing cycle.

Note

When INC_BACKGROUND_SESSION_PURGE is enabled and the cycle time is set to a non-zero value then this method must also be called to set the initial value before `Create` can be called.

`CycleTime` (getter)

`unsigned long` **`HttpdSessionManager::CycleTime`** `(void);`

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. It returns the current interval between session scrubbing cycles (in milliseconds).

`MaxSessionAge` (getter)

`long` **`HttpdSessionManager::MaxSessionAge`** `(void);`

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. It returns the current maximum allowable session age (in seconds).

`ScrubbingBatchSize` (getter)

`size_t` **`HttpdSessionManager::ScrubbingBatchSize`** `(void);`

This method is only available if INC_BACKGROUND_SESSION_PURGE is enabled. It returns the current session scrubbing batch size.

`Insert`

`int` **`HttpdSessionManager::Insert`** `(HttpdSessionObject *`*`p_obj`*`);`

This method inserts *`p_obj`* into the session manager. If the object is inserted successfully (0 is returned) then the session was inserted. Upon successful return, the session will be given a reference count of `1` and should be unlocked (via `Unlock`) when the pointer is no longer needed (typically at the end of a HTTP transaction).

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`UnlockedInsert`

`int` **`HttpdSessionManager::Insert`** `(HttpdSessionObject *`*`p_obj`*`);`

This method is identical to `Insert` except the session manager mutex is not locked. Callers must obtain the lock prior to calling this method.

`Find`

`int` **`HttpdSessionManager::Find`** `(const char *`*`p_session_id`*`, HttpdSessionObject *&`*`p_obj`*`);`

This method uses the session identifier (obtained via the `HttpdSessionObject::SessionId` method) in *`p_session_id`* to locate the session object. This string is typically stored on the client either in a cookie or passed as a hidden form variable. If the session is still stored in the container its reference count is increased and its address is placed into `p_obj`.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

UnlockedReference

void **HttpdSessionManager::UnlockedReference** (HttpdSessionObject *p_obj);

This method increments the reference count of `p_obj`. Callers must lock the session manager mutex before calling this method. This is typically done when searching for a session object in some other manner (while holding the lock) and then referencing the object so the session manager lock can be released.

Unlock

void **HttpdSessionManager::Unlock** (HttpdSessionObject *p_obj);

Whenever an object is inserted (via `HttpdSessionManager::Insert`) or retrieved (via `HttpdSessionManager::Find`) its reference count is incremented to prevent it from being destroyed by another thread.

When the session object is no longer needed for the remaining processing of the request it should be unlocked using this method.

Delete

void **HttpdSessionManager::Delete** (HttpdSessionObject *p_obj);

If a session object is to be destroyed (such as a user logging out, for example) then a pointer to the session object can be passed to `Delete` instead of `Unlock` to destroy the object. The session object should be locked by at least one thread.

If the session object is in use by other threads then it is not destroyed until all threads using it unlock it (via `HttpdSessionManager::Unlock`).

Mutex

HttpdMutex &**HttpdSessionManager::Mutex** (void);

The `HttpdSessionManager` is thread-safe because a `HttpdMutex` is used to synchronize access to the list of session objects.

If session objects are tracked in a manner external to the `HttpdSessionManager` it may be desirable to have a single lock manage both lists. In these cases this method gives access to the lock used to maintain the session object list.

There are also non-sychronized versions of the accessor methods that can be called when the lock is obtained externally via this method.

`HttpdSessionObject` Reference

Introduction

This class is a base class for objects managed by the HttpdSessionManager class. This class overrides `operator new` and `operator delete` to allocate space using HttpdOpSys::Malloc.

In addition to some helper methods, the `HttpdSessionObject` class defines some protected data members that are for the use of the `HttpdSessionManager` class.

Public Methods

`SessionId`

```
void HttpdSessionObject::SessionId (char *p_session_id);
```

This method obtains the session identifier that can be used to track the session. This should only be called after a successful insertion of the session object into the manager.

The buffer pointed to by *`p_session_id`*, which must be at least HTTPD_SESSION_KEY_LEN characters in length, is filled in with the session identifier. This string is generally sent to the client (either as a cookie or hidden form variable) to identify the session object on subsequent requests (via the `HttpdSessionManager::Find` method).

`Deleted`

```
bool HttpdSessionObject::Deleted (void);
```

This method returns true if the object has been marked for deletion.

`HttpdDynamicOutput` Reference

Introduction

Some HTTP features designed to increase efficiency do not work well when the length of the content is unknown. In particular, persistent connections do not work without a `Content-Length:` header. Generating dynamic content is considerably easier when the length does not have to be known in advance. This is even true of Seminole's template system.

There are several approaches to this problem. The simplest is to close the connection whenever an object of unknown length is requested. This results in lower throughput and wasted bandwidth. Another option is to buffer dynamically generated content in memory at the server end. Once it is generated, the length of the buffered data is known and can then be sent out. Of course, this leads to increased memory consumption on the server as well as a delay in sending the content. The third solution uses chunked transfer encoding. This solution sends out the data in small chunks. The length of each chunk is sent along with the chunk so the receiver can keep in sync. This solution is almost ideal for dynamically generated content but it is only supported by HTTP/1.1 or higher.

Seminole includes HttpdContentSink and HttpdChunkedSink classes that handle the protocol mechanics of buffering and chunking content, respectively. The `HttpdDynamicOutput` class acts as a switchboard to select these different mechanisms and provide a uniform interface for generating dynamic content.

One of the major goals of Seminole is that it be small but also support as much of the HTTP protocol as possible. To achieve both of these goals, `HttpdDynamicOutput` uses conditional compilation to (optionally) avoid as much support code as possible. The INC_PERSISTENT_CONN, INC_BUFFER_OUTPUT, and INC_CHUNK_OUTPUT options control how much support code `HttpdDynamicOutput` requires.

An important question to ask is should content be written for `HEAD` requests. The answer is: it depends. If no content is written then there is no wasted effort in generating it — less CPU load. In this case the repsonse to the `HEAD` request will not include a `Content-Length` header which may be the reason the `HEAD` request was submitted.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdDynamicOutput`

`HttpdDynamicOutput::HttpdDynamicOutput` `(HttpdRequest *`*`p_request`*`, bool` *`is_head`*`);`

This constructor initializes the dynamic output engine. If the request should only require headers (a `HEAD` request, for example) then the parameter *`is_head`* should be set to `true`.

Note

For optimal memory utilization and efficiency it is best if the `HttpdDynamicOutput` object can be constructed *before* the `HttpdRequest::Respond` or `HttpdRequest::ResponseHeader` methods are called. Otherwise, the resulting headers may be out of sync with the response.

Furthermore, only one `HttpdDynamicOutput` instance should be associated with a request. Therefore the `HttpdDynamicOutput` should be created in the innermost scope that covers its use. Typically this is the point at which the handler has determined how to handle the request and dynamically generated output is necessary.

It is normal to construct an instance of this class on the stack and then pass a pointer to it down to the various routines that generate the content.

`Header`

`void` **`HttpdDynamicOutput::Header`** `(const char *`*`p_name`*`, const char *`*`p_value`*`);`

This method sends a *MIME* header to the output stream. The *`p_name`* should contain the name of the header without the colon or other separator characters. No processing is done on *`p_value`*, however, multi-line escapes can be included within *`p_value`* as long as it does not end with a CRLF (as this is supplied automatically by this method).

Note

This method can be called as many times as necessary and should follow the call to the `HttpdRequest::Respond` method of the request.

`HeaderComplete`

`void` **`HttpdDynamicOutput::HeaderComplete`** `(void);`

This method should be called after all headers have been written (via the `Header` method).

Body

`HttpdWritable *` **`HttpdDynamicOutput::Body`** `(void);`

This method obtains the object that should receive the dynamically generated content. It is impossible for this method to return NULL or result in an error.

Note

This method can be called at any point after construction of the `HttpdDynamicOutput` object. However, it is *very important* that no data be written to the object until after the `HeaderComplete` method is called.

Headers

`HttpdWritable *` **`HttpdDynamicOutput::Headers`** `(void);`

This method returns a pointer to a stream that can be used to dump header data to in place of using the `Header` method. Like the `Header` method, data should only be written to this stream after the call to `HttpdRequest::Respond` and before the call to `HeaderComplete`.

This method will never return NULL.

HttpdInboundTransfer Reference

Introduction

`HttpdInboundTransfer` is used to process received data from an HTTP client; such as `POST` requests.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdInboundTransfer

`HttpdInboundTransfer::HttpdInboundTransfer` `(HttpdRequest *`*`p_request`*`, int &`*`rc`*`);`

This function prepares the inbound transfer associated with *`p_request`*. The success of opening the transfer is placed into *`rc`*. If the status is non-zero (i.e. an error) then `HttpdInboundTransfer::Receiver` should not be called.

Receiver

`HttpdReceiver *` **`HttpdInboundTransfer::Receiver`** `(void);`

This function returns an interface for reading data from the transfer.

`HttpdOutboundTransfer` Reference

Introduction

`HttpdOutboundTransfer` is used to process received data from an HTTP server.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdOutboundTransfer`

`HttpdOutboundTransfer::HttpdOutboundTransfer` `(HttpdSocket &`*`socket`*`, HttpdMimeParser *`*`p_parser`*`, int &`*`rc`*`);`

This function prepares the outbound transfer associated with the socket and *MIME* parser. The success of opening the transfer is placed into *`rc`*. If the status is non-zero (i.e. an error) then `HttpdOutboundTransfer::Receiver` should not be called.

`Receiver`

`HttpdReceiver *` **`HttpdOutboundTransfer::Receiver`** `(void);`

This function returns an interface for reading data from the transfer.

`HttpdTracer` Reference

Introduction

The `HttpdTracer` class provides a simple debugging facility for Seminole This is especially important when integrating Seminole into an existing system. In order to remain “lean and mean” Seminole uses the C++ preprocessor (along with `HttpdTracer`) to show whats going on.

Tracing support is enabled by setting the INC_TRACING build option to a non-zero (true) value. If INC_TRACING is defined to be `0` then tracing has no runtime overhead impact whatsoever.

Seminole includes many built-in trace points at interesting locations that should allow easy bring-up of even the most complex configurations without resorting to a debugger (well a debugger on Seminole).

Using the Tracing Macros

In order to trace a particular block of code the tracer must be declared. This is done using the HTTPD_DIARY macro rather than a standard C++ declaration. Once declared the tracer object can be used to print informational messages using HTTPD_NOTE. Expression values can be logged using HTTPD_LOG. In addition, if the type of an expression needs to be forced to a particular type there are variants of HTTPD_LOG which include a type cast: HTTPD_LOGL for long integers, HTTPD_LOGUL for unsigned long integers, and HTTPD_LOGP for pointers.

The HTTPD_DIARY macro takes an argument that defines the minimum trace level required to display the messages. The trace level is controlled with the static member variable `HttpdTracer::mTraceLevel`. The trace level is divided into discrete ranges that roughly parallel the various operational phases of Seminole Trace messages for a particular diary will only be displayed if `mTraceLevel` is equal to or above the level associated with the HTTPD_DIARY macro call. The trace levels are defined with an enumeration inside the `HttpdTracer` class:

Tracing Levels

`NONE`	No tracing should be performed.
`STARTUP`	Tracing for the various startup phases such as the spawning of the acceptor and the installation of handlers.
`REQUESTS`	Tracing for incoming requests and the basic processing mechanism.
`AUTH`	The authentication phase is typically done after the incoming request. This trace level is after `REQUESTS` but takes place before `HEADERS`. Additional authentication may be performed later on, this trace level is merely a convention.
`HEADERS`	This tracing phase is typically associated with the processing the headers of an associated request.
`PREPROCESSING`	This phase is used to denote any additional processing before the real “meat” of request processing.
`LOGIC`	This phase denotes the core processing logic in the `Handle` method of the handler.
`RESPONSE`	This phase is used to denote the delivery of the HTTP response to the client and any logic (such as template evaluation) involved in this phase.
`POSTPROCESSING`	This phase is used to denote any additional processing after the `RESPONSE` phase. A good example is logging or auditing of requests which is generally performed after the response is delivered for performance reasons.
`CLIENT`	This level covers the operation of `HttpdClient` and its associated classes during HTTP client operations.
`ALL`	This tracing level covers all phases.

Important

Only one tracer can be declared in a single scope, so each scope should contain only one call to the HTTPD_DIARY macro. Typically a single call at the beginning of a routine is sufficient.

As a simple example, this function is adorned with tracing:

```
void MyFrobalizer(int a, char *p_address)
{
  bool free_server;

  HTTPD_DIARY(STARTUP);

  HTTPD_LOG(a);
  if (p_address == NULL)
  {
    HTTPD_NOTE("No address provided, getting it from the server");
    p_address = GetFromServer();
    free_server = true;
  }

  int connector = ConnectTo(p_address);
  int offset    = DefaultConnectorOffset();
  HTTPD_LOG(connector + offset);

  DoSomething(connector, offset);
  if (free_server)
    HttpdOpSys::Free(p_address);
}
```

As you can see the HTTPD_LOG macro conveniently takes an expression and logs it. Using the stringizing operator of the preprocessor your trace includes the expressions along with time stamps and file names and line numbers.

Chapter 3. Support Classes

There are many classes that are part of the public *API* in Seminole although they are also used “under the covers” to support other classes. Just as with the much of the core *API* these classes also may be used without a webserver instance if useful.

`HttpdFileSystem` Reference

Introduction

An `HttpdFileSystem` is an abstraction of a particular “namespace” of files. This class is derived and implemented by various file system providers.

Filesystem/backing store concepts can range from a fully hierarchical tree with long filenames to a flat namespace with very constrained naming conventions, or possibly a single binary image containing discrete chunks of data. Seminole abstracts filesystem services using an abstract interface built around the `HttpdOpSys`, `HttpdFileInfo`, `HttpdFile`, and `HttpdDirectory` objects. The abstraction is designed to be as generic as possible. For example, some filesystems have two distinct concepts when opening up a file:

- Locating the file and computing an “internal identifier” from the name.
- Actually transporting the file data from the storage medium to the requesting code.

Seminole separates the concept of the file metadata from the data. This makes opening a file a two-step process:

1. Build a `HttpdFileInfo` object that is attached to the file.
2. Open the file based on the `HttpdFileInfo` object and the requested access.

In the case of filesystems where these two concepts are a single atomic operation, the abstraction layer can simply keep a file name as part of the `HttpdFileInfo` object.

There can be any number of file systems present at the same time, all abstracted by `HttpdFileSystem` instances. Instances of this interface serve as factories for file info, file, and directory objects from a file system.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`FileInfo` (From path)

```
int HttpdFileSystem::FileInfo (const char *p_path, HttpdFileInfo &info);
```

This method obtains information about a file named *`p_path`* and places it into *`info`*. The *`info`* object can then be used to open the file (or directory) for access.

In addition, the data within the *`info`* object can be queried without the overhead of opening the file.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

`HttpdFileSystem` implementations should take note that (for efficiency) callers may use the same `HttpdFileInfo` object repeatedly to query information about multiple paths. As such implementations of this method should always be sure to set all the fields.

`FileInfo` (From parent & path tuple)

`int` **`HttpdFileSystem::FileInfo`** `(const HttpdFileInfo *`*`p_parent`*`, const char *`*`p_name`*`, HttpdFileInfo &`*`info`*`);`

This method obtains information about a file named *`p_name`* that is contained in the directory identified by *`p_parent`*. If *`p_parent`* is NULL then the root of the hierarchy of this filesystem is assumed. If *`p_parent`* is not NULL then it must be the obtained information for a directory. The gathered information is placed into the *`info`* object.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

`HttpdFileSystem` implementations should take note that (for efficiency) callers may use the same `HttpdFileInfo` object repeatedly to query information about multiple paths. As such implementations of this method should always be sure to set all the fields.

`OpenFile`

`int` **`HttpdFileSystem::OpenFile`** `(const HttpdFileInfo &`*`info`*`, int` *`mode`*`, HttpdFile *&`*`p_file`*`);`

Assuming that `IsDir` is not true for *`info`*, the associated file is opened. The address of the opened file system object is placed in *`p_file`*. For *`mode`*, it can be one of HttpdFileSystem::FILE_READ_ONLY or HttpdFileSystem::FILE_READ_WRITE depending on the desired access.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`OpenDirectory`

`int` **`HttpdFileSystem::OpenDirectory`** `(const HttpdFileInfo &`*`info`*`, HttpdDirectory *&`*`p_dir`*`);`

Assuming that `IsDir` is true for *`info`*, the associated directory is opened for iteration. The address of the opened directory object is placed in *`p_file`*.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`Open`

`int` **`HttpdFileSystem::Open`** `(const char *`*`p_name`*`, int` *`mode`*`, HttpdFile *&`*`p_file`*`);`

This method is a little helper that obtains file information for the file named *`p_name`* with the `FileInfo` and then opens the file.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`LoadFile` (ASCII)

`int` **`HttpdFileSystem::LoadFile`** `(const char *`*`p_filename`*`, char *&`*`p_result`*`);`

This helper method loads the contents of the file specified by *`p_filename`* into a null-terminated buffer. Upon success, *`p_result`* points to the file contents in allocated storage.

It is the caller's responsibility to free the buffer (using HttpdOpSys::Free).

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

`LoadFile` (binary)

`int` **`HttpdFileSystem::LoadFile`** `(const char *`*`p_filename`*`, char *&`*`p_result`*`, size_t &`*`size`*`);`

This version of the `LoadFile` is identical to the ASCII version with the exception of the size (in bytes) of the file is placed in the *`size`* parameter.

`Delete` (Parent & path tuple)

`int` **`HttpdFileSystem::Delete`** `(const HttpdFileInfo *`*`p_parent`*`, const char *`*`p_name`*`);`

This method deletes a file named *`p_name`* that is contained in the directory identified by *`p_parent`*. If *`p_parent`* is NULL then the root of the hierarchy of this filesystem is assumed. If *`p_parent`* is not NULL then it must be the obtained information for a directory.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

File systems must not implement a recursive delete. If the file to be deleted is a directory and it is not empty then it must not be removed and an error must be returned.

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

`Delete` (via HttpdFileInfo))

`int` **`HttpdFileSystem::Delete`** `(const HttpdFileInfo &`*`info`*`);`

This method deletes the file identified by *`info`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

File systems must not implement a recursive delete. If the file to be deleted is a directory and it is not empty then it must not be removed and an error must be returned.

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

MakeDirectory

`int HttpdFileSystem::MakeDirectory (const HttpdFileInfo *p_parent, const char *p_name);`

This method creates an empty directory named `p_name` that is contained in the directory identified by `p_parent`. If `p_parent` is NULL then the root of the hierarchy of this filesystem is assumed. If `p_parent` is not NULL then it must be the obtained information for a directory.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

MakeFile

`int HttpdFileSystem::MakeFile (const HttpdFileInfo *p_parent, const char *p_name, HttpdFile *&p_file);`

This method creates and opens an empty file named `p_name` that is contained in the directory identified by `p_parent`. If `p_parent` is NULL then the root of the hierarchy of this filesystem is assumed. If `p_parent` is not NULL then it must be the obtained information for a directory.

If successful the open file is returned in `p_file` which must be closed when no longer needed by the caller. The file is always opened for reading and writing.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

CopyFrom

`int HttpdFileSystem::CopyFrom (const HttpdFileInfo &from, const HttpdFileInfo *p_parent, const char *p_dest);`

This method creates a new file, named `p_dest`, from the contents of the file identified by `from`. The newly created file is placed in the directory identified by `p_parent`. If `p_parent` is NULL then the root of the hierarchy of this filesystem is assumed. If `p_parent` is not NULL then it must be the obtained information for a directory.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

MoveTo

`int HttpdFileSystem::MoveTo (const HttpdFileInfo &from, const HttpdFileInfo *p_parent, const char *p_to);`

This method relocates (or renames) the file or directory identified by *`from`* to *`p_dest`* in the directory identified by *`p_parent`*. If *`p_parent`* is NULL then the root of the hierarchy of this filesystem is assumed. If *`p_parent`* is not NULL then it must be the obtained information for a directory.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

GetQuota

`int HttpdFileSystem::GetQuota (const HttpdFileInfo &info, HttpdFileQuota &quota);`

If quota information is available for this filesystem then this method populates the fields of *`quota`* with quota information.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

Note

This method is only available if INC_FILE_QUOTAS is enabled.

Members of HttpdFileQuota

Type: unsigned long
Name: *`mAvailable`*
Description: The available writing space in units of 1000 bytes.
Type: `unsigned long`
Name: *`mUsed`*
Description: The available space used in units of 1000 bytes.

SupportsQuota

`int HttpdFileSystem::SupportsQuota (void); const`

If this filesystem supports quota information then this method returns `true`. Otherwise `false` is defined.

Note

This method is only available if INC_FILE_QUOTAS is enabled.

Protected Methods

CommonFileInfo

`int HttpdFileSystem::CommonFileInfo (const HttpdFileInfo &info);`

This is a helper routine for subclasses of `HttpdFileSystem`. It sets up fields in the `HttpdFileInfo` object *`info`* with values for parameters common to all file systems.

Public Data

This is an abstract interface class and therefore contains no data members of interest.

`HttpdFileInfo` Reference

Introduction

Generally, a distinction is made between a file's contents and metadata concerning the file. `HttpdFile` objects provide access to a file's contents, while `HttpdFileInfo` objects provide access to file metadata. `HttpdFileInfo` instances are generally created by the caller and populated by HttpdFileSystem::FileInfo.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`IsDir`

```
bool HttpdFileInfo::IsDir (void);
```

Determine whether this `HttpdFileInfo` object refers to a directory, on platforms where this concept exists.

Returns true if a directory, false if not.

`FileSystem` (getter)

```
HttpdFileSystem *HttpdFileInfo::FileSystem (void);
```

Returns the file system provider associated with this file.

`MimeType` (getter)

```
const char *HttpdFileInfo::MimeType (void);
```

Determine the *MIME* type of the file described by the parent `HttpdFileInfo` object.

Returns a pointer to a string containing the *MIME* type encoding upon success. The returned value should never be NULL if a `FileInfo` call returned success on this object.

Note

This method should not be considered an absolute guarantee of file type; some file systems (such as the platforms' native file system) do not provide any method for explicitly describing a file's contents other than direct inspection. For these file systems, this method provides at best an educated guess based on naming conventions, etc.

`Size` (getter)

unsigned long **HttpdFileInfo::Size** (void);

Determine the size in octets of the file described by the parent `HttpdFileInfo` object.

Returns the number of octets (on systems with 8-bit bytes, this also happens to be the number of bytes).

`LastModificationTime`

const HttpdTimeStamp * **HttpdFileInfo::LastModificationTime** (void);

This method returns the last time the file was modified.

`CreationTime`

const HttpdTimeStamp * **HttpdFileInfo::CreationTime** (void);

This method returns the time the file was created.

`FileSystem` (setter)

void **HttpdFileInfo::FileSystem** (HttpdFileSystem **p_fs*);

This method is used to set the associated file system provider of the file.

`ChangeLastModificationTime`

HttpdTimeStamp * **HttpdFileInfo::ChangeLastModificationTime** (void);

This method is used to set the last modification time of the file information. Normally this method is only used by providers of a file system interface.

`ChangeCreationTime`

HttpdTimestamp * **HttpdFileInfo::ChangeCreationTime** (void);

This method is used to set the creation time of the file information. Normally this method is only used by providers of a file system interface.

`Size` (setter)

void **HttpdFileInfo::Size** (unsigned long *sz*);

This method is used to set the size (in bytes) of the file information. Normally this method is only used by providers of a file system interface.

`IsDir` (setter)

void **HttpdFileInfo::IsDir** (bool *is_it*);

This method is used to set the directory flag of the file information. Normally this method is only used by providers of a file system interface.

`MimeType` (setter)

void **HttpdFileInfo::MimeType** (const char **p_type*, bool *must_free*);

This method is used to set the *MIME* type of the file information. If *`must_free`* is true, it is assumed that the storage for *`p_type`* was allocated with HttpdOpSys::Malloc. Normally this method is only used by providers of a file system interface.

`Location` (getter)

`HttpdParameter` **`HttpdFileInfo::Location`** `(void);`

This obtains the location property of the file. This is an internal value that should be used by a file system provider to track the referenced file.

The purpose of the location is to split apart the operation of finding a file from a catalog and to actually doing I/O from the file. Of course, for some operating systems (such as POSIX) this can store the file name if separating these two actions is impossible.

Because this data is specific to a file system provider only the associated provider should be used to open the file.

`Location` (setter)

`void` **`HttpdFileInfo::Location`** `(HttpdParameter` *`param`*`, bool` *`must_free`*`);`

This method is used to set the location tag of the file information. If *`must_free`* is true, it is assumed that the storage for mpVoid field of *`param`* was allocated with HttpdOpSys::Malloc. Normally this method is only used by providers of a file system interface.

`ETag` (setter)

`void` **`HttpdFileInfo::ETag`** `(const char *`*`p_tag`*`, bool` *`must_free`*`, bool` *`is_weak`* `= false);`

This method is only present if INC_ETAGS is enabled. If so, this method sets the ETag member to point to the new ETag in *`p_tag`*. If *`must_free`* is true, it is assumed that the string pointed to by *`p_tag`* was allocated with HttpdOpSys::Malloc and therefore must be freed when no longer needed. Normally this method is only used by providers of a file system interface.

Entity tags come in two flavors: weak and strong which affect how they compare. The *`is_weak`* argument can be used to indicate the specified tag is a weak one.

In most cases generating a completely unique entity tag for a given file is prohibitively expensive. Most file system implementations use meta-data to construct the entity tag rather than a hash function (such as MD5). In these cases implementations should be careful to not generate an entity tag with a high probability of not changing if the file contents can change.

If *`p_tag`* is NULL then the weak flag should be ignored.

`ETag` (getter)

`const char *`**`HttpdFileInfo::ETag`** `(void);`

This method is only present if INC_ETAGS is enabled. If so, this method returns the ETag of the file object if one exists. If no ETag is available for the file, NULL is returned.

`ETagIsWeak`

`bool` **`HttpdFileInfo::ETagIsWeak`** `(void);`

This method determines if the entity tag is weak.

`Attributes` (setter)

`void` **`HttpdFileInfo::Attributes`** `(HttpdCgiParameter *`*`p_attrs`*`);`

Every file can have various name-value pairs associated with it. This meta-data is managed using the HttpdCgiParameter class. If any attributes are available for a file this method stores the list in the `HttpdFileInfo` object. It is important to understand that this method does not make a copy of the attributes and once given to this method they should no longer be managed by the caller. If no attributes are available for this file it is safe to call this method with a *`p_attrs`* value of NULL. Normally this method is only used by providers of a file system interface.

`Attributes` (getter)

`HttpdCgiParameter *`**`HttpdFileInfo::Attributes`** `(void);`

This method obtains the attribute list for the file. If there are no attributes then NULL is returned. The `HttpdFileInfo` object owns the list and callers should refrain from modifications of the attribute list.

Public Data

`HttpdFileInfo` contains no publically accessible data members.

`HttpdFile` Reference

Introduction

An `HttpdFile` object represents a valid file "handle" suitable for performing I/O operations on. Its semantics are as consistent as possible across heterogeneous platforms, and this class should be used to perform file-related tasks in a portable manner.

The `HttpdFile` class is an abstract interface. File systems provide appropriate implementations of this interface. These specific implementations are accessed through `HttpdFileSystem::OpenFile` method and do not need to be created by users of this class.

If the INC_MODIFIABLE_FILESYSTEMS feature is enabled the interface specified by `HttpdFile` also includes HttpdWritable. Files that are writable can be used anywhere the `HttpdWritable` interface can be utilized.

Even though the compile-time feature enables methods for modifying files and filesystem structure this does not guarantee that a file (or filesystem) is modifiable. The default implementation of modification methods return a `HttpdOpSys::ERR_NOTREADY` error code.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`Read`

`int` **`HttpdFile::Read`** `(void *`*`p_buffer`*`, size_t &`*`sz`*`);`

Read `sz` bytes from the `HttpdFile` object, and store the result in the storage pointed to by `p_buffer`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes"). `sz` is updated to reflect the actual number of bytes read. If end-of-file is reached, success is returned and `sz` is set to `0`.

ReadObject

```
int HttpdFile::ReadObject (void *p_buffer, size_t sz);
```

Read exactly `sz` bytes from the `HttpdFile` object, and store the result in the storage pointed to by `p_buffer`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes"). If `sz` bytes could not be read, this method returns `HttpdOpSys::ERR_BADFORMAT`.

Write

```
int HttpdFile::Write (size_t sz, const void *p_buffer);
```

Write `sz` bytes from the storage pointed to by `p_buffer` to the `HttpdFile` object.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes"). If successful, all bytes were written by `Write()`.

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

SetSize

```
int HttpdFile::SetSize (unsigned long size);
```

This method sets the size of the file to `size` bytes. If the file is larger then it will be truncated. If the file is smaller then it will be grown.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Note

This method is only available if INC_MODIFIABLE_FILESYSTEMS is enabled.

Seek

```
int HttpdFile::Seek (long offset, int whence);
```

Change the current position of the seek pointer associated with the `HttpdFile`. If `whence` is set to `FILE_SEEK_START`, `offset` represents the new absolute position of the seek pointer. A value of `FILE_SEEK_CUR` adds `offset` to the seek pointer's current position. `FILE_SEEK_END` adds `offset` to the size of the file and sets the seek pointer to that value.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

`Tell`

`int` **`HttpdFile::Tell`** `(unsigned long &`*`offset`*`);`

Obtain the current position of the seek pointer associated with the file. The zero-based position is stored in *`offset`* on success.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`PushToSink`

`int` **`HttpdFile::PushToSink`** `(HttpdWritable *`*`p_sink`*`);`

This method writes the entire contents of the file to the object pointed to by *`p_sink`*. Before calling the file pointer should be at the begining of the file and is indeterminate after this operation.

The default implementation of this method simply transfers files in blocks of `XFER_BUF_SIZE` bytes. Implementations of the `HttpdFile` interface may override the default implementation if a more efficient approach is possible.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`PushFileSegment`

`int` **`HttpdFile::PushFileSegment`** `(HttpdWritable *`*`p_sink`*`, unsigned long` *`start_offs`*`, unsigned long` *`end_offs`*`);`

This method writes the specified window of the contents of the file to the object pointed to by *`p_sink`*. The range of bytes written starts at *`start_offs`* byte offset (inclusive) and ends at the byte position of *`end_offs`* (exclusive). After this call the file pointer is indeterminate.

The default implementation of this method simply transfers files in blocks of `XFER_BUF_SIZE` bytes. Implementations of the `HttpdFile` interface may override the default implementation if a more efficient approach is possible.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Public Data

Other than the constants mentioned in the `Seek()` entry, `HttpdFile` contains no publically available data members.

`HttpdDirectory` Reference

Introduction

For those filesystems which support the concept of hierarchical namespaces or file listings, `HttpdDirectory` objects provide the ability to traverse one directory's contents in a linear fashion. Like `HttpdFile` objects the `HttpdDirectory` object is opened using the `HttpdFileSystem::OpenDirectory` given a file info object.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

Name

`const char *HttpdDirectory::Name (void);`

Return the currently loaded directory entry in this `HttpdDirectory` object. The syntax of the resultant string is entirely system-dependent.

The provided string pointer is valid until the originating `HttpdDirectory` is closed.

Next

`bool HttpdDirectory::Next (void);`

Load the next directory entry in series within the parent `HttpdDirectory` object.

Returns true if successful, false if no further directory entries exist.

Close

`void HttpdDirectory::Close (void);`

Destroy the `HttpdDirectory` object and release any allocated resources. After calling this method the pointer to the `HttpdDirectory` is no longer valid.

Public Data

`HttpdDirectory` contains no publically accessible data members.

`HttpdReadOnlyMemoryFile` Reference

Introduction

The class `HttpdReadOnlyMemoryFile` implements the file interface against a read-only buffer.

Note

Only additional methods are described here. This class implements the methods in the `HttpdFile` class.

Public Methods

HttpdReadOnlyMemoryFile

`HttpdReadOnlyMemoryFile::HttpdReadOnlyMemoryFile (const void *p_data, size_t sz);`

Associates a file with *sz* bytes pointed to by *p_data*.

HttpdMemoryFile Reference

Introduction

The class HttpdMemoryFile implements the file interface against a data buffer.

Note

Only additional methods are described here. This class implements the methods in the HttpdFile class.

Public Methods

HttpdMemoryFile

HttpdMemoryFile::HttpdMemoryFile (void **p_buffer*, size_t *sz*);

Associates a file with *sz* bytes pointed to by *p_buffer*.

HttpdRedirectResponse Reference

Introduction

The HttpdRedirectResponse class coordinates sending back redirect responses to HTTP requests. For simple applications the HttpdRequest::Redirect method is more appropriate. Using this class additional *MIME* headers (such as Set-Cookie) can be appended to the redirect.

Instances of HttpdRedirectResponse encapsulate the state involved in sending out a redirect response. Under normal use the Begin method is called. If successful the response is partially complete and in the *MIME* header phase. Callers can then write out additional headers to the HttpdRequest object. After writing any additional headers, callers should invoke the End method to complete the response.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdRedirectResponse

HttpdRedirectResponse::HttpdRedirectResponse (HttpdRequest **p_request*, int *status*);

The constructor prepares the object to perform the redirect. The *p_request* parameter is a pointer to the current request. The type of redirect response is specified in *status*; see Supported HTTP Response Codes for possible values.

Begin

```
int HttpdRedirectResponse::Begin (const char *p_url);
```

Begin the response to the `HttpdRequest` object given to the constructor of this object. The *`p_url`* parameter is the target URL for the redirection. The URL does not have to be absolute.

Important

The return code indicates a success or failure of the operation (see Table 4.1, “OS Abstraction Layer Error Codes”). If `0` is returned the caller should generate any additional headers and invoke the `End` method.

Upon failure no further action should be taken as an appropriate error response is sent to the client before the failure return of this routine.

End

```
void HttpdRedirectResponse::End (void);
```

This method must be called after the *MIME* headers are sent to the client (assuming `Begin` returned success).

HttpdSocket Reference

Introduction

`HttpdSocket` serves as a container for protocol-specific network operations, and provides abstract access to a communication endpoint connected with a client.

If the INC_MULTIPLE_TRANSPORTS option is not enabled then the `HttpdSocket` is simply a synonym for the platform-specific `HttpdTcpSocket` object. If INC_MULTIPLE_TRANSPORTS is enabled then `HttpdSocket` acts as an abstraction to one or more *transport* layers.

The interface of `HttpdSocket` closely mirrors the Berkeley sockets *API*, and hence will be quite familiar to experienced UNIX® or WinSock programmers. It is expected that additional abstraction or separation of platform independent and dependent code will occur in this area, so its interfaces are subject to future change.

Generally, `HttpdSocket` itself is encapsulated by an `HttpdRequest` object, so it is often of little concern to programmers modifying Seminole within the existing framework (e.g. adding a handler).

Transport objects (i.e. `HttpdTcpSocket`) are derived from `HttpdSocketInterface`. The `HttpdSocketInterface` interface is ultimately derived from HttpdWritable and thus provides an interface for writing data.

The interface provided by `HttpdSocketInterface` closely parallels the methods provided by `HttpdSocket`. Implementors porting Seminole are encouraged to study the existing socket implementations for reference.

Public Methods

Initialize

```
static int HttpdSocket::Initialize (void);
```

Initialize the socket abstraction. This static method is called by `Httpd::Init` before any any socket (including the listening socket) is created.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method does not have to be idempotent. It is called once and only once by `Httpd::Init`.

Write

`int` **`HttpdSocket::Write`** `(size_t` *`nbytes`*`, const void *`*`ptr`*`);`

Given a pointer *`ptr`* to a block of storage *`nbytes`* bytes in length, attempt to write the data therein to a network endpoint (socket).

It is important to note that some network API's have semantics which make it possible for writes to return successfully, yet incomplete, as opposed to *blocking* until an error occurs or all data has been written. `Write` takes the latter approach, so Seminole programmers need not make allowances for it.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

EnterReadMode

`int` **`HttpdSocket::EnterReadMode`** `(void);`

Before the `HttpdSocket::ReadN`, `HttpdSocket::Read`, or `HttpdSocket::Gets` methods can be called, this method must be invoked to prepare the socket for reading.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

ReadN

`int` **`HttpdSocket::ReadN`** `(void *`*`ptr`*`, size_t` *`nbytes`*`, unsigned int` *`timeout`*`);`

Given a pointer *`ptr`* to a block of previously allocated storage, read *`nbytes`* bytes of data from a network endpoint (socket). If no data is received for *`timeout`* seconds, the read is aborted and `HttpdOpSys::ERR_NOTREADY` is returned.

It is important to note that some network API's have semantics which make it possible for reads to return successfully, yet incomplete, as opposed to *blocking* until an error occurs or all data has been read. `ReadN` takes the latter approach, so Seminole programmers need not make allowances for it. Success will only be returned if *`nbytes`* are actually received. If partial reads are desired, `Read` should be used instead.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

Note

The `EnterReadMode` method must be called before `ReadN` can be called.

Read

`int` **`HttpdSocket::Read`** `(void *`*`ptr`*`, size_t &`*`nbytes`*`, unsigned int` *`timeout`*`);`

Given a pointer *`ptr`* to a block of previously allocated storage, read up to *`nbytes`* bytes of data from a network endpoint (socket). The value of *`nbytes`* is updated with the actual number of bytes read. If no data is available then `Read` will *block* for up to *`timeout`* seconds. As soon as any data is received this function copies it into the buffer and returns.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

Note

The `EnterReadMode` method must be called before `Read` can be called.

Read (multiple wait version)

`int` **`HttpdSocket::Read`** `(void *`*`ptr`*`, size_t &`*`nbytes`*`, unsigned int` *`timeout`*`, HttpdSocketWaitHandle` *`wait_for`*`);`

This method only exists if the portability layer defines `HAVE_SOCK_WAIT` to `1`. If the portability layer and underlying operating system support waiting for other events in addition to a socket event then the *`wait_for`* parameter acts as an “escape hatch” to pass an object to wait on to the operating system (and/or network stack).

LeaveReadMode

`int` **`HttpdSocket::LeaveReadMode`** `(void);`

After reading on the socket is complete, this method must be invoked to allow write operations (via `Write`) on the socket.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

Gets

`bool` **`HttpdSocket::Gets`** `(char *`*`p_buf`*`, size_t` *`maxbuf`*`, unsigned int` *`timeout`*`);`

Given a pointer *`p_buf`* to a block of previously allocated storage, read one line from the communications endpoint represented by the parent `HttpdSocket` object, assuming that each line is terminated by a newline character (ASCII line feed). The value of *`maxbuf`* is used to advise `Gets()` of the maximum length of the storage pointed to by *`p_buf`*. The resulting string is terminated by an ASCII `NUL` character.

It is worth noting that a carriage return may be embedded in the buffer, as `Gets()` does not purge them.

If an entire input line is not received in *`timeout`* seconds this function should return false.

Returns true on success, false upon failure. This method should never return true unless the string *`p_buf`* contains at least one character.

AbortGets

`bool` **`HttpdSocket::AbortGets`** `(void);`

This method attempts to abort another thread on this socket blocked in the `Gets` method. If the thread is sucessfully unblocked then this method should return true. If the thread can not be aborted or is not blocked in `Gets` then false is returned. The return value does not have to be precise as there may be race conditions involved with this operation. The intention of this method is a “best effort” attempt.

Note

This method need only be implemented if INC_OVERLOAD_PROTECTION is non-zero.

Socket

`bool` **`HttpdSocket::Socket`** `(const char *`*`p_transport`*`);`

Initializes a communications endpoint, which can subsequently be used to receive a connection from clients, or to establish an outbound connection with a server.

Calling `Socket()` is a generally a prerequisite for calling any other method in `HttpdSocket` meaningfully.

Returns true on success, false upon failure.

The *`p_transport`* is the transport to be used for this socket and its children. If INC_MULTIPLE_TRANSPORTS is not enabled then this parameter should not be provided.

Close

`void` **`HttpdSocket::Close`** `(void);`

Destroys the communications endpoint associated with the parent `HttpdSocket` object. `Close()` does not perform an orderly cleanup of an active connection, so if “graceful” termination of a connection is desired, use the `Shutdown()` method instead.

Listen

`bool` **`HttpdSocket::Listen`** `(HttpdIpPort` *`port`*`, const char **`*`pp_options`*`);`

Causes a previously initialized communications endpoint to be placed into a listening state, so that network clients can connect to it. The local port designated by *`port`* is used to discriminate incoming connections.

`pp_options` contains a list of open-ended list of name/value pairs that can be used to configure the specifics of the various *transport* layers. The list must be terminated with a NULL pointer. If no socket options are desired then the default value of the parameter, `HttpdSocket::mEmptySocketOptions`, may be passed as this parameter.

The life-time of *`pp_options`* is not required to extend beyond the call to `Listen`. Therefore it is the responsabilty of the socket implementation to locally copy any information it may need from the *`pp_options`* array.

Although the options supported by the socket are dependant on the implementation of the portability layer most implementations handle a common subset of options. What follows is a general summary rather than a specification. Those implementing a new portability layer should attempt to follow existing practice.

Option	**Meaning**	**Example**
`bind`	This option binds the socket to a particular interface.	`bind:192.168.1.16`
`ipv6`	If the target supports IPv6 (`INC_IPV6_SUPPORT`) then this option configures the socket for communication on an IPv6 network.	`ipv6`

Option	Meaning	Example
`bind6`	If the target supports IPv6 (`INC_IPV6_SUPPORT`) then the socket is bound to a specific IPv6 listening address. This option is mutually exclusive with the `bind` and `ipv6` options.	bind6:19::12ab:00d1

`Listen` returns true on success, false upon failure.

Connect

`bool` **`HttpdSocket::Connect`** `(HttpdIpAddress` *`addr`*`, HttpdIpPort` *`port`*`, const char **`*`pp_options`*`);`

Causes a previously initialized communications endpoint to be connected with a remote system and process. The remote system's network address is provided in *`addr`*, while the remote port is provided in *`port`*.

`pp_options` contains a list of open-ended list of name/value pairs that can be used to configure the specifics of the various *transport* layers.

Returns true on success, false upon failure.

ConnectTo

`int` **`HttpdSocket::ConnectTo`** `(const char *`*`p_host`*`, HttpdIpPort` *`port`*`, const char *const *`*`pp_options`*`);`

This method connects to the specified port using the provided host name. Optional platform-specific parameters may be specified in *`pp_options`* to control how the socket is connected.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Shutdown

`void` **`HttpdSocket::Shutdown`** `(void);`

Destroys the communications endpoint associated with the parent `HttpdSocket` object. The currently established connection, if any, is first closed in an orderly fashion.

Accept

`bool` **`HttpdSocket::Accept`** `(HttpdParameter &`*`con`*`, HttpdIpAddress &`*`client_addr`*`);`

Accept incoming connections to the communications endpoint (previously prepared with `Listen()`) contained within the parent `HttpdSocket` object.

Upon successful acceptance of a new connection, the peer's network address is placed in *`client_addr`*, while a handle for the connection itself is placed in *`con`*. Once the new connection is appropriately dispatched (typically by the creation of a new server thread, process, or task), `Accept()` can be called again to set up the next incoming connection request on the original endpoint. Thus, each connection accepted creates a new, unique pair of endpoints.

Returns true on success, false upon failure.

Note

A newly accepted connection should be aborted by means of the HttpdSocket::Cancel method or initialized using the HttpdSocket::Socket method. When no longer needed *`client_addr`* should be disposed of by calling `HttpdSocketFoundation::FreeAddress`.

`Cancel`

`void` **`HttpdSocket::Cancel`** `(HttpdParameter` *`param`*`);`

After acceptance of a new connection via HttpdSocket::Accept, it is possible to find that the connection should be prematurely ended, either for administrative reasons or system errors. In that case, the new connection should be aborted by means of this method.

Note

The object used to invoke this method should be the listening socket that generated the HttpdParameter value.

`Socket`

`bool` **`HttpdSocket::Socket`** `(HttpdSocket *`*`p_listen`*`, HttpdParameter` *`param`*`);`

After acceptance of a new connection via HttpdSocket::Accept, this method takes the generated HttpdParameter value and initializes a socket object associated with the incoming connection. The listening socket that generated *`param`* should be passed in as *`p_listen`*.

Note

If this method fails no further methods should be called on the socket object.

`GetLocalAddress`

`int` **`HttpdSocket::GetLocalAddress`** `(HttpdIpAddress &`*`addr`*`);`

This method obtains the local address (near end) that a listening or accepted socket is associated with.

On success a `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

If successful it is the responsability of the caller to discard the IP address object returned with the `HttpdSocketFoundation::FreeAddress` method.

`ForceShutdown`

`void` **`HttpdSocket::ForceShutdown`** `(void);`

This method is called on a listening socket. When called it will unblock all waiting calls on any socket objects for any thread that were originated from this listening socket. Methods such as `Gets` and `Read` and `ReadN` should return with an error in the case of this method being called in another thread.

Note

When implementing this particular method it is important to pay close attention to the lifetimes of the sockets. In particular the listening socket may be destroyed before all of its children sockets are.

In this case if `ForceShutdown` is called and then the listening socket is immediately destroyed all of the child sockets must still have been released.

In addition, this method is not required to block while the other threads are released. The synchronization is instead handled by waiting for the worker threads to terminate.

After this method is called, all child sockets should remain as unreadable until the listening socket is closed.

`Transport`

`const HttpdTransport *` **`HttpdSocket::Transport`** `(void);`

This method returns a pointer to the transport object associated with the socket. This method should only be called after the socket has been initialized with the `HttpdSocket::Socket` method.

Note

This function is not available unless INC_MULTIPLE_TRANSPORTS is enabled.

Public Data

`mEmptySocketOptions`

`const char *mEmptySocketOptions[];`

Some socket calls (i.e. `HttpdSocket::Listen`) take a list of parameters. This variable is the default list of options if no extra parameters are needed.

`HttpdSocketInterface` Reference

Introduction

The class `HttpdSocketInterface` is the base class for all transports that are used by Seminole.

The following methods must be provided by subclasses of `HttpdSocketInterface`. The behavior of the transport-specific implementations should be identical to the definitions of the following methods in HttpdSocket.

- `static int` **`HttpdSocketInterface::Initialize`** (void);
- `int` **`HttpdSocketInterface::Write`** (size_t *nbytes*, const void **ptr*);
- `int` **`HttpdSocketInterface::EnterReadMode`** (void);
- `int` **`HttpdSocketInterface::Read`** (void **ptr*, size_t &*nbytes*, unsigned int *timeout*);
- `int` **`HttpdSocketInterface::Read`** (void **ptr*, size_t &*nbytes*, unsigned int *timeout*, HttpdSocketWaitHandle *wait_for*);

Note

This method is only provided if the portability layer defines `HAVE_SOCK_WAIT` to `1`.

- `int` **`HttpdSocketInterface::ReadN`** (void **ptr*, size_t *nbytes*, unsigned int *timeout*);
- `int` **`HttpdSocketInterface::LeaveReadMode`** (void);
- `bool` **`HttpdSocketInterface::Gets`** (char **p_buf*, size_t *maxbuf*, unsigned int *timeout*);
- `void` **`HttpdSocketInterface::Close`** (void);
- `bool` **`HttpdSocketInterface::Listen`** (HttpdIpPort *port*, const char ***pp_options*);
- `bool` **`HttpdSocketInterface::Connect`** (HttpdIpAddress *addr*, HttpdIpPort *port*, const char ***pp_options*);
- `void` **`HttpdSocketInterface::Shutdown`** (void);
- `bool` **`HttpdSocketInterface::Accept`** (HttpdParameter &*con*, HttpdIpAddress &*client_addr*);
- `void` **`HttpdSocketInterface::Cancel`** (HttpdParameter *param*);
- `int` **`HttpdSocketInterface::GetLocalAddress`** (HttpdIpAddress &*addr*);
- `bool` **`HttpdSocketInterface::AbortGets`** (void); (only if INC_OVERLOAD_PROTECTION is enabled).

Important

The methods defined in HttpdSocketInterface Public Methods are methods that must be provided in addition to those listed above.

Public Methods

Socket

`bool` **`HttpdSocketInterface::Socket`** `(void);`

Initialize the communications object. This method is almost always called before calling any other method of the class.

Returns true on success, false upon failure.

Socket

`bool` **`HttpdSocketInterface::Socket`** `(HttpdSocketInterface *`*`p_listen`*`, HttpdParameter `*`param`*`);`

After acceptance of a new connection via `HttpdSocketInterface::Accept` this method converts the HttpdParameter handle value and initializes a socket object associated with the incoming connection. The listening socket that generated *`param`* is passed in as *`p_listen`*.

Factory

`static HttpdSocketInterface * **HttpdSocketInterface::Factory** (void);`

This method only needs to be defined if INC_MULTIPLE_TRANSPORTS is enabled. Using the `operator new` that is provided in `HttpdSocketInterface`, this function should return an instance of the particular socket associated with the class.

The address of this method is registered with the HttpdTransport structure associated with this particular protocol.

HttpdSocketFoundation Reference

Introduction

The `HttpdSocketFoundation` class is the base class for anything related to the socket and networking abstraction provided by the portability layers. `HttpdSocketInterface` and `HttpdUdpServerSocket` are derived from this class although this class has no non-static members. The inheritance is solely to provide access to the utility routines in this namespace.

The utility routines are general purpose helpers for dealing with HttpdIpAddress values. They may be called anywhere they are necessary or useful not just from classes derived from `HttpdSocketFoundation`.

If the portability layer defines HTTPD_HAVE_BULKY_SOCKET_ADDRESSES then a few additional methods are available for use by portability layers to help manage large address objects. These methods should not be called by platform independent code.

Public Methods

CreateAddress

int **HttpdSocketFoundation::CreateAddress** (HttpdIpAddress &*addr*, const char **p_str_rep*);

This routine translates a string representation of an address to an `HttpdIpAddress` object. The format of the string representation may be platform dependent and is determined by the portability layer.

On success a `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

If successful it is the responsability of the caller to discard the IP address object created with the `HttpdSocketFoundation::FreeAddress` method.

This method should be implemented by the portability layer.

AddressEqual

bool **HttpdSocketFoundation::AddressEqual** (HttpdIpAddress *addr_1*, HttpdIpAddress *addr_2*);

This method should be used by platform independent code to determine if two `HttpdIpAddress` objects refer to the same address. If so then this method returns `true` otherwise `false` is returned.

`CopyAddress`

`int` **`HttpdSocketFoundation::CopyAddress`** `(HttpdIpAddress &`*`dest`*`, HttpdIpAddress` *`src`*`);`

This method copies an address object from *`src`* into the variable referred to by *`dest`*.

On success a `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Note

If successful it is the responsability of the caller to discard the IP address object in *`dest`* with the `HttpdSocketFoundation::FreeAddress` method.

`FreeAddress`

`void` **`HttpdSocketFoundation::FreeAddress`** `(HttpdIpAddress` *`addr`*`);`

This method frees an `HttpdIpAddress` when it is no longer needed. All address objects must eventually be released using this method. This is true even if the `HttpdIpAddress` originated from a method other than `CreateAddress` (such as the one returned from `HttpdSocketInterface::Accept`.

`CreateAddress` (Portability Layer Support)

`int` **`HttpdSocketFoundation::Create`** `(HttpdIpAddress &`*`addr`*`);`

This method is only available if the portability layer defines HTTPD_HAVE_BULKY_SOCKET_ADDRESSES to a non-zero value. As such this method should only be called by the portability layer code when it must create an address object and return it to the caller. A typical example of this is the socket object method `GetLocalAddress`.

On success a `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Note

If successful it is the responsability of the caller to discard the IP address object in *`dest`* with the `HttpdSocketFoundation::FreeAddress` method.

`HashAddress`

`size_t` **`HttpdSocketFoundation::HashAddress`** `(HttpdIpAddress` *`addr`*`);`

This method computes a hash value for *`addr`*. The hash value can be computed with any appropriate algorithm. Preferrably the algorithm should evenly distribute addresses around the hash space (the entire range of size_t).

`FormatAddress`

`void` **`HttpdSocketFoundation::FormatAddress`** `(HttpdIpAddress` *`addr`*`, char *`*`p_str`*`);`

This method converts *`addr`* to a string representation. The size of the buffer pointed to by *`p_str`* is guaranteed to be at least `HTTPD_IPADDR_STR_LEN` bytes in length by all callers. The portability layer must define this constant as appropriate.

`HttpdUdpServerSocket` Reference

Introduction

If the platform supports UDP sockets (`HAVE_UDP_SOCKETS` not equal to zero) then the portability layer should provide an implementation of this class which is used to send and receive datagram packets. If the platform supports multicast then the preprocessor symbol HTTPD_HAVE_UDP_MULTICAST should be defined to a non-zero value and it should be possible to use multicast addresses with this class.

This class abstracts a UDP socket that is capable of both sending and receiving packets on a port that is bound at creation time. As with the `HttpdTcpSocket` various parameters for the socket are specified as an array of strings. This allows platform specific options to be passed easily from the client application through protocol code to the socket layer.

Public Methods

`Socket`

```
int HttpdUdpServerSocket::Socket (HttpdIpPort &port, const char *const
*pp_options = mDefaultOptions);
```

This method must be called before the `HttpdUdpServerSocket` can be used. The *`port`* parameter specifies what port the UDP socket listens on. If any socket specific options are to be specified then *`pp_options`* should point to an array of parameter strings terminated by a `NULL`.

If *`port`* is set to `0` (and the system supports this concept) a free port is allocated and upon successful return the value of *`port`* is set to the allocated port.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

The options supported by the socket are dependant on the portability layer. However the included portability layers provide some general options that work across most platforms.

Option	**Meaning**	**Example**
`bind`	This option binds the socket to a particular interface.	`bind:192.168.1.16`
`mcast`	If the target supports multicast (`HAVE_UDP_MULTICAST`) then this option joins this socket into a particular multicast group. Optionally an interface address can be specified to join the multicast group on a particular interface.	`mcast:238.17.1.1` or with an interface address: `mcast:238.17.1.1,192.168.1.16`
`mc-loop`	If the socket is part of a multicast group this enables loopback of multicast packets. Any packets transmitted are	mc-loop:0

Option	Meaning	Example
	also queued for reception. The argument is zero to turn off loopback or non-zero to turn it on. This option is only available if `HAVE_UDP_MULTICAST` is enabled. Not all platforms support this option.	
`mc-ttl`	This sets the TTL for packets transmitted with a multicast address. Not all platforms support this option.	`mc-ttl:32`
`sndbuf`	This option sets the size of the send buffer (in bytes) that holds packet data until the necessary interface becomes available. Not all platforms support this option.	`sndbuf:65536`
`rcvbuf`	This option sets the size of the receive buffer (in bytes) that holds packet data received from network interfaces until it can be processed. Not all platforms support this option.	`rcvbuf:65536`

Close

```
void HttpdUdpServerSocket::Close (void);
```

This method shuts down any operations on the socket and releases any resources owned by the socket. No operations should be performed on the socket once this method is called.

ForceShutdown

```
void HttpdUdpServerSocket::ForceShutdown (void);
```

In order to halt a thread that may be suspended performing a read operation on the socket this method aborts the readers with an error code.

Note

Portability layers may only perform a “best effort” implementation of this method. So it care should be taken that shutdown can happen without this method being perfect.

ReadPacket

```
int HttpdUdpServerSocket::ReadPacket (void *p_buffer, size_t &len,
HttpdIpAddress &addr, HttpdIpPort &port, unsigned int timeout);
```

This method reads a packet from the socket. If no packet is available it will block for up to *`timeout`* milliseconds. If no packet is received within this time then `ERR_NOTREAD` is returned.

If a packet is received properly it is placed into the buffer pointed to by *`p_buffer`* and *`len`* is set to the length of the packet (in bytes). The source address and port are placed into *`addr`* and *`port`*, respectively.

Upon receiving a packet, `0` is returned. On timeout `ERR_NOTREADY` is returned. If the read operation is aborted by the `ForceShutdown` method then `ERR_SYSPERM` is returned. Otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

SendPacket

`int` **`HttpdUdpServerSocket::SendPacket`** `(const void *`*`p_buffer`*`, size_t` *`len`*`, HttpdIpAddress` *`addr`*`, HttpdIpPort` *`port`*`);`

This method is used to send responses to requests. It is not limited to sending packets on the port that the socket is bound to. Rather the port and destination address are specified for each packet sent and may differ for each packet.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

HttpdIpAddressBase Reference

When the representation of an IP address is large (i.e. more complex than a single scalar value) the portability layer may define HTTPD_HAVE_BULKY_SOCKET_ADDRESSES to request that Seminole handle the larger addresses in an efficient manner.

When bulky addresses are present the HttpdIpAddress type becomes a pointer to the `HttpdIpAddressObject` class.

The `HttpdIpAddressObject` is a platform specific class that is defined and implemented by the portability layer if address objects are large. The `HttpdIpAddressBase` class provides support for the storage and lifetime of its only intended superclass - `HttpdIpAddressObject`.

Note

`HttpdIpAddressBase` is only available if the portability layer defines HTTPD_HAVE_BULKY_SOCKET_ADDRESSES to a non-zero value.

An `HttpdAllocatorCache` is used to efficiently allocate `HttpdIpAddressObject` instances. It is required that `HttpdIpAddressObject` implement:

- `operator==`
- virtual destructor (if necessary)
- `operator=` (if necessary)

`HttpdIpAddressObject` is also free to implement any methods in addition to the above provided they are not called from platform independent code.

HttpdMemoryAllocator Reference

Introduction

`HttpdMemoryAllocator` provides a dynamic memory pool with an interface similar to the `malloc()` and `free()` functions provided by the standard C runtime system. It can be used for pooling memory in certain thread contexts or to provide dynamic heap allocation in the portability layer for an underlying operating system with no notion of dynamic memory.

Instances of this class are not thread-safe, and multiple accesses to it should be guarded by a mutual exclusion mechanism such as that provided by an HttpdMutex.

Public Methods

`Create`

`void` **`HttpdMemoryAllocator::Create`** `(void *`*`p_mem`*`, size_t `*`sz`*`);`

`HttpdMemoryAllocator` objects are not usable until this method initializes the memory allocator, given a pre-existing memory arena of size *`sz`* bytes pointed to by *`p_mem`*.

This method is guaranteed not to fail, and instead will generate assertions (if enabled) when given incorrect parameters.

`Allocate`

`void *` **`HttpdMemoryAllocator::Allocate`** `(size_t `*`sz`*`);`

Allocate new memory *`sz`* bytes in length.

Returns a pointer to a buffer of at least the requested size, taking into account host alignment requirements, or NULL upon error or exhaustion of the memory pool.

`Free`

`void` **`HttpdMemoryAllocator::Free`** `(void *`*`p_ptr`*`);`

Release a block of allocated memory pointed to by *`p_ptr`*.

`Reallocate`

`void *` **`HttpdMemoryAllocator::Reallocate`** `(void *`*`p_oldptr`*`, size_t `*`newsz`*`);`

Expand or shrink the size of the memory block pointed to by *`p_oldptr`*, to be *`newsz`* bytes in length.

Returns a revised pointer upon success, or NULL upon failure.

Important

If `Reallocate()` fails to change the size of a given block of memory, the original block remains valid and can be used normally. Therefore, it is important to keep track of the previous allocation and free it as necessary.

Public Data

`HttpdMemoryAllocator` contains no publically accessible data members.

`HttpdAllocatorCache` Reference

Introduction

`HttpdAllocatorCache` caches pre-allocated memory buffers for quick allocation of fixed size objects. The contents of the blocks are not guaranteed across allocations. The allocator is thread safe and may be accessed by multiple threads simultaneously.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`HttpdAllocatorCache`

`**HttpdAllocatorCache::HttpdAllocatorCache** (size_t *object_size*, size_t *max_depth*);`

This constructs an allocation cache for objects of *`object_size`* bytes. The *`max_depth`* parameter controls the maximum number of free objects that the cache will hold.

The object can not be used until the `Create` method is called first.

`Create`

`int **HttpdAllocatorCache::Create** (size_t *initial_depth* = 0);`

This method creates and initializes the cache. The cache populates itself with *`initial_depth`* objects.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Prune`

`void **HttpdAllocatorCache::Prune** (size_t *desired* = 0);`

This method reduces the size of the cache to *`desired`* objets. If less than *`desired`* objects exist in the cache then the method simple returns success.

`AllocateObject`

`void * **HttpdAllocatorCache::AllocateObject** (void);`

This method allocates an object from the cache. If the cache is empty then it attempts to allocate an object using HttpdOpSys::Malloc.

Returns a pointer to the newly allocated upon success, or NULL if there is insufficient memory to allocate the object.

`FreeObject`

`void **HttpdAllocatorCache::FreeObject** (void **p_object*);`

This method frees an object allocated from the cache. If *`p_object`* is NULL then this method performs no operation. If *`p_object`* is not NULL then it must have been a value returned from `AllocateObject`.

`PurgeAllCaches`

`void **HttpdAllocatorCache::PurgeAllCaches** (void);`

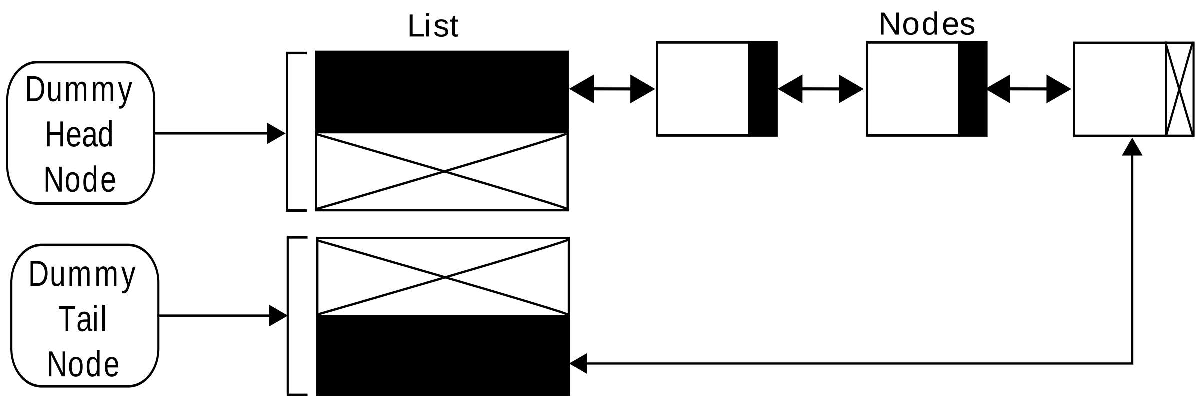
If INC_ALLOCATION_CACHE_PURGE is enabled then this static method frees all cached memory from all allocator caches. If INC_ALLOCATION_CACHE_PURGE is disabled then this method does nothing.

`HttpdList` and `HttpdListNode` Reference

Introduction

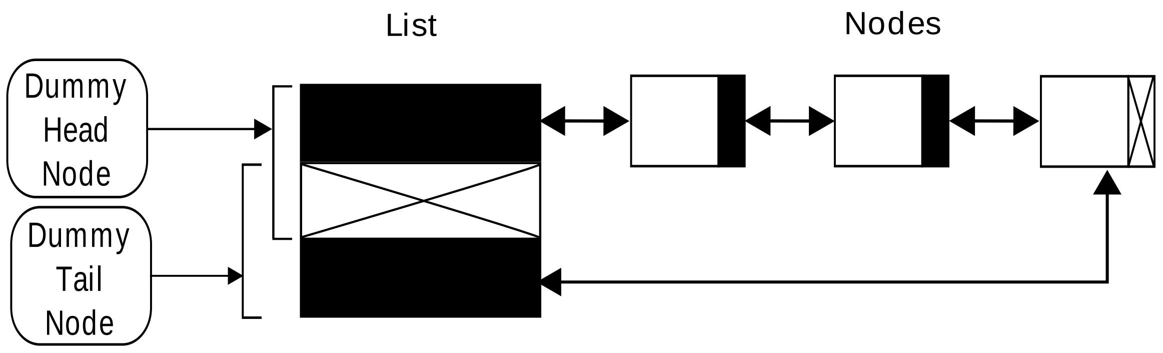
The `HttpdList` and `HttpdListNode` classes implement compact and efficient doubly-linked list container support. Lists can be made circular and insertions can be performed at any point. A very important feature is that `HttpdListNode` contains a backpointer to the object that owns it. This allows an object to be linked into several lists at the same time.

There are several strategies for doubly linked-lists. The most common approach is to use NULL as a value of a next or previous pointer to indicate that no node exists beyond the current one. This requires that many special cases for end of node be peppered all over the code for inserts and deletes. A workaround for this is to use two “dummy nodes” that are always present, even in an empty list.



Traditional approach to dummy nodes

The dummy nodes waste little space for the amount of code they save, but they still waste four pointers worth of space. We can optimize this further by overlapping the dummy nodes. This optimization reduces the overhead to three pointers per list. Without any dummy nodes a list would need to contain a minimum of a head and tail pointer so the dummy node overhead is minimal.



Compact dummy nodes

Public Methods (`HttpdListNode`)

`Owner` (Getter)

```
void *HttpdListNode::Owner (void);
```

Each node object maintains a backpointer to the owning object. This method obtains the value of the backpointer.

`Owner` (Setter)

```
void HttpdListNode::Owner (void *p_value);
```

Set the backpointer in the node to *p_value*.

`Next`

```
HttpdListNode *HttpdListNode::Next (void);
```

Get the address of the next node in the list.

`Prev`

```
HttpdListNode *HttpdListNode::Prev (void);
```

Get the address of the previous node in the list.

`InsertBefore`

```
void HttpdListNode::InsertBefore (HttpdListNode *p_pos);
```

Insert this node before the node specified by *p_pos*.

`InsertAfter`

```
void HttpdListNode::InsertAfter (HttpdListNode *p_pos);
```

Insert this node after the node specified by *p_pos*.

`Remove`

```
void HttpdListNode::Remove (void);
```

Remove the node from the list it is inserted in.

`MakeCircular`

```
void HttpdListNode::MakeCircular (void);
```

This method allows a node to be constructed that is considered a single, circular list. Other nodes can then be inserted around it.

Public Methods (`HttpdList`)

`Initialize`

`void` **`HttpdList::Initialize`** `(void);`

Initialize a list object. This method must be called before the list can be used.

`IsEmpty`

`bool` **`HttpdList::IsEmpty`** `(void);`

This method returns true if the list is empty (contains no nodes other than the dummy nodes). Otherwise the list is not considered empty and false is returned.

`AddToHead`

`void` **`HttpdList::AddToHead`** `(HttpdListNode *`*`p_node`*`);`

Insert *`p_node`* to the front of list.

`AddToTail`

`void` **`HttpdList::AddToTail`** `(HttpdListNode *`*`p_node`*`);`

Insert *`p_node`* to the rear of list.

`Head`

`HttpdListNode *`**`HttpdList::Head`** `(void);`

Return the front node on the list or NULL if the list is empty.

`Tail`

`HttpdListNode *`**`HttpdList::Tail`** `(void);`

Return the rear node on the list or NULL if the list is empty.

`CountChildren`

`size_t` **`HttpdList::CountChildren`** `(void);`

This method counts the number of nodes in the list and returns the value. For large lists this operation may take some CPU time.

`Concatenate`

`void` **`HttpdList::Concatenate`** `(HttpdList &`*`src`*`);`

This method concatenates all of the nodes in the list specified by *`src`* to the tail of this list. After this call, *`src`* is no longer a valid list and must be re-initialized (via `Initialize`) if it is to be used again.

`MakeCircular`

`void` **`HttpdList::MakeCircular`** `(void);`

This method removes the dummy nodes of the list from the nodes already linked in to the list. Thus, the nodes that were previously linked in the list object are turned into a circular chain of nodes. After this call, *`src`* is no longer a valid list and must be re-initialized (via `Initialize`) if it is to be used again.

It is important to get a pointer to at least one of the nodes in the list before calling this method. After this call completes the list object is no longer associated with any of the nodes in the list.

Iterating over lists

Iterating the contents of `HttpdList` objects can be error prone. Therefore a helper class, called `HttpdListIterator`, is provided to make this easier. Instances of `HttpdListIterator` are typically used as index variables in for-loops.

For example, to iterate the contents of a list from head to tail the following construct can be employed:

```
for(HttpdListIterator i(list.Head()); i.Continue(); i.Next())
{
  SomeClass *p_obj = (SomeClass *)(void *)i;
  p_obj->DoSomething();
}
```

The conditional state of the loop is always provided by the `HttpdListIterator::Continue` method. Notice the way we obtain the object pointer. Casting the iterator to void * is equivalent to calling `HttpdListNode::Owner` to obtain the data pointer from the node.

Traversing the list from tail to head is follows a similar structure:

```
for(HttpdListIterator i(list.Tail()); i.Continue(); i.Prev())
{
  HttpdListNode *p_node = (HttpdListNode *)i;
  ProcessNode(p_node);
}
```

Here we cast the `HttpdListIterator` object to a HttpdListNode *. This obtains the current node we are iterating over. Deleting nodes from a list during traversal deserves special attention. If we wanted to remove the node from the list during iteration we would have to do something like this:

```
HttpdListIterator i(list.Head());
while (i.Continue())
{
  // We must store the pointer to the node here as we may be
  // modifying the list later on.
  HttpdListNode *p_node = (HttpdListNode *)i;

  bool need_adj = NeedsAdjustment((SomeObject *)p_node->Owner());

  // Advance the iterator before we alter the node.
  i.Next();

  // Now we can fiddle with the node.
```

```
      if (need_adj)
      {
        p_node->Remove();
        adjustment_list.AddToHead(p_node);
      }
    }
```

The iterator can be repositioned to an arbitrary node with the `HttpdListIterator::Reposition` method.

HttpdBitSet Reference

Introduction

The `HttpdBitSet` class acts as a pointer to HttpdBitWord where each bit can be individually manipulated or examined. This class is mainly useful for maintaining arrays of boolean values or for small set membership. No memory allocation or range checking is done by this class, it really does function just like a native pointer.

To use the `HttpdBitSet` it must first be assigned storage of a suitable size. The the static method `Size` can be used to compute the size of the storage required. The store (as a pointer to HttpdBitWord) can be assigned to the `HttpdBitSet` object.

To set a bit the operator `+=` is used. To clear a bit the operator `-=` is used. To check the value of a bit the `HttpdBitSet` object can be dereferenced like an array.

Note

The storage provided is accessed as an array of `HttpdBitWord` objects. Therefore the storage must have alignment that is appropriate for these accesses.

```
HttpdBitSet    bs;
HttpdBitWord   storage = 0;

bs = &storage; // Assign the storage.

// Set bits #9 and #11.
bs += 9;
bs += 11;

// Now print out all the set bits.
for(unsigned int i = 0; i < 16; i++)
  printf("Bit %d is %s\n", i, bs[i] ? "set" : "clear");
```

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

Size

`size_t` **`HttpdBitSet::Size`** `(size_t` *`bits`*`);`

This static method returns the number of bytes required for an `HttpdBitSet` object to hold the number of bits specified by *`bits`*.

Elements

`size_t` **`HttpdBitSet::Elements`** `(size_t` *`bits`*`);`

This static method returns the number of `HttpdBitWord` entries in the storage needed to hold the number of bits specified by *`bits`*.

RemoveLeadingSet

`size_t` **`HttpdBitSet::RemoveLeadingSet`** `(size_t` *`bits`*`);`

This operation is like a left shift on the entire bitmap. The portion of the bitmap affected is limited to *`bits`* bits in length. The number of bits the bitmap is shifted is computed automatically such that the first bit in the resulting bitset is the first (leftmost) `0` (i.e. unset) bit in the bitset.

The number of bits shifted out is returned.

Storage

`HttpdBitWord *`**`HttpdBitSet::Storage`** `(void);`

This static method returns the pointer to the storage the `HttpdBitSet` is using.

`HttpdMacroProcessor` Reference

Introduction

`HttpdMacroProcessor` is a utility class for doing string substitutions. An input string is written to either a dynamic string or an HttpdWritable interface. Tokens in the input string are parsed and a pure virtual method is called to replace the macro.

Substitutions are broken up into an array of strings — much like a POSIX command line. In fact two forms of quoting are available as well. The argument vector is then used by the `Command` method to perform a substitution. For example, consider the following macro string:

```
The user is $(age -years) years old $(today "\n\x1b") and has a bank account
balance of $$ $(account 'John Q Public' 1234).
```

Notice that doubling the special character, `$` in this case is used as a literal escape. Otherwise it is required that a macro begin with a left parenthesis following the special character. Notice in the above example that a macro may contain three types of tokens: non-quoted strings, single-quoted strings, and double-quoted strings.

Non-quoted strings must not have a "`)`" or whitespace character in them. Either of those characters is a delimiter either ending the substitution or delimiting the next argument.

Strings quoted with a single quote character can have any character in them except for a single quote character. This is the most general form of quoting.

Strings quoted with a double quote character allow ANSI C style escape sequences.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdMacroProcessor`

```
HttpdMacroProcessor::HttpdMacroProcessor (char special = '$');
```

This constructs the macro processor object with *`special`* as the delimiter character.

Expand (sink version)

```
int HttpdMacroProcessor::Expand (HttpdWritable *p_target, const char
*p_macro);
```

This function writes *`p_macro`* to *`p_target`* expanding tokens delimited by the special character in the process.

Upon success `0` is returned. Otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Expand (string version)

```
int HttpdMacroProcessor::Expand (char *&p_output, const char *p_macro);
```

This function copies *`p_macro`* into a string pointed to by *`p_output`*. The resultant string is allocated dynamically and should be freed using HttpdOpSys::Free by the caller if successful.

Upon success `0` is returned. Otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Protected Methods

`Command`

```
int HttpdMacroProcessor::Command (void);
```

This pure virtual method is called when a substitution has been parsed and is to be replaced. The substitution is broken apart into a vector of arguments — the `mArgCount` and `mArgs` protected member variables. The substituted text (if any) should be written to `mpTarget`.

WriteString

`int` **`HttpdMacroProcessor::WriteString`** `(size_t` *`args`*`, const char *`*`p_string`*`);`

This function formats *`p_string`* as a macro substitution. This method should be called from the implementation of `Command`.

The remaining arguments (starting at offset *`args`*) allow various transformations to be performed on *`p_string`* before it is written. The following transformations are possible:

- `html` escapes characters that are HTML tokens.
- `uri` encodes the string using the `HttpdUtilities::UriEncode` routine.
- `unuri` decodes the string using the `HttpdUtilities::UriDecode` routine.
- `unuri+` decodes the string using the `HttpdUtilities::UriDecode` routine with the *`plus_xlat`* parameter set to `true`.
- `c-ascii` encodes the string using the section called “`CQuoteString`” with the `STR_QUOTE_C` mode.
- `js-utf8` encodes the string using the section called “`CQuoteString`” with the `STR_QUOTE_JSON` mode.
- The `remove-chars` attribute causes any characters in its value to be removed from the formatted string.
- `remove` removes any characters from the string as specified by the next argument.
- `filter` removes any characters from the string as not specified by the next argument.

Upon success `0` is returned. Otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdCgiMacroProcessor` Reference

Introduction

The `HttpdCgiMacroProcessor` is a subclass of `HttpdMacroProcessor`. This class takes a `HttpdCgiParameter` list and uses the contents of that list for expanding macros. When a value is found in the list it is written using `HttpdMacroProcessor::WriteString`.

Public Methods

HttpdCgiMacroProcessor

`HttpdCgiMacroProcessor::HttpdCgiMacroProcessor` `(HttpdCgiParameter *`*`p_params`*`, char` *`special`* `= '$');`

This constructs the macro processor object where *`p_params`* is used to satisfy the value of the substitutions. The character *`special`* specifies the delimiter character.

`HttpdHtmlQuoter` Reference

Introduction

`HttpdHtmlQuoter` is a helper class for HTML-escaping strings. Although this task can be accomplished with the `HttpdUtilities::HtmlQuote` method using this class is more efficient. The `HtmlQuote` method always copies the resulting string to dynamically allocated storage, even if there are no characters to be escaped.

The `HttpdUtilities::NeedsHtmlQuoting` method scans a string for characters that need quoting. If none are found then the call to `HtmlQuote` can be avoided. However care must be taken to free the allocated memory only if `HtmlQuote` is actually called.

This class handles these details, automatically freeing allocated memory when it is destroyed. The typical use case for this class is to be allocated on the stack for the duration that the quoted string is needed. For example:

```
int              rc;
HttpdHtmlQuoter  quoter(some_string, rc);
if (rc != 0)
{
  // Handle the error!
  return;
}

rc = p_stream->Printf("<code>%s</code>\n", quoter.Quoted());
if (rc != 0)
{
  // Things just aren't good today.
  return;
}
```

`HttpdDataSource` Reference

Introduction

The `HttpdDataSource` is a base class that represents a source of data. Examples are things like flash chips, files, and memory buffers. The interface exposed by this class is realized by the HttpdMemoryDataSource class provided by Seminole. For specialized data sources such as banked flash chips, external files, or even network sources user-written implementations of this interface may be created.

Public Methods

ReadAt

`int` **`HttpdDataSource::ReadAt`** `(void *`*`p_data`*`, size_t` *`sz`*`, unsigned long` *`offset`*`);`

This pure virtual function is the interface for reading data from the source. On success, *`sz`* bytes are written to the buffer pointed to by *`p_data`* starting at *`offset`* bytes from the start of the data from the source.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

`ReadValue` (32-bit)

`int` **`HttpdDataSource::ReadValue`** `(HttpdUint32 &`*`val`*`, unsigned long` *`offset`*`);`

This method reads a 32-bit unsigned value from the source starting at offset *`offset`* from the start of the data in the source. The decoding is performed by the HttpdUtilities::AssembleU32 routine.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes"). The returned value is placed in the *`val`* argument.

`ReadValue` (16-bit)

`int` **`HttpdDataSource::ReadValue`** `(HttpdUint16 &`*`val`*`, unsigned long` *`offset`*`);`

This method reads an unsigned 16-bit value from the source starting at offset *`offset`* from the start of the data in the source. The decoding is performed by the HttpdUtilities::AssembleU16 routine.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes"). The returned value is placed in the *`val`* argument.

`AddressOf`

`const void *` **`HttpdDataSource::AddressOf`** `(unsigned long` *`offset`*`, size_t` *`sz`*`);`

Some sources of data are accessible directly via a memory address. For those kinds of sources this method allows access to the memory space. If a `HttpdDataSource` does not support access via a pointer this function can safely return NULL and the data will be accessed using the `ReadAt` and `ReadValue` methods.

If the data can be mapped for *`sz`* bytes starting at *`offset`* from the start of the data in the source, the mapped address should be returned. The returned address should be valid until `ReleaseAddress` is called on the returned pointer.

This function should only be implemented for cases where access through a pointer would be faster than calls to `ReadAt`. For example, allocating a buffer and reading the contents of a file into it is not really any more efficient than having the data read into a buffer provided to `ReadAt`.

However, an implementation data source backed by something like a Disk-On-Chip® from M-Systems with a memory-mapping window would implement this method in a special way to avoid the copy if sufficient mapping window is available.

`ReleaseAddress`

`void` **`HttpdDataSource::ReleaseAddress`** `(const void *`*`p_addr`*`);`

This method releases any resources associated with a mapped address obtained from `AddressOf`.

`HttpdMemoryDataSource` Reference

Introduction

The `HttpdMemoryDataSource` is a class that abstracts an addressable region of memory as a data source (see HttpdDataSource).

A very typical use of this class is to provide an interface between content data stored in an array by the bin2c tool and the HttpdRomFileSystem. The initialized array from **bin2c** is accessed via a `HttpdMemoryDataSource` that is provided to an instance of `HttpdRomFileSystem`.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Public Methods

`HttpdMemoryDataSource`

`HttpdMemoryDataSource::HttpdMemoryDataSource` `(void *`*`p_data`*`, size_t `*`sz`*`);`

This function initializes a memory data source that points to *`p_data`* and is *`sz`* bytes in length.

`HttpdFileDataSource` Reference

Introduction

The `HttpdFileDataSource` is a class that abstracts a `HttpdFile` object as a data source (see HttpdDataSource).

There are two distinct implementations of this class depending on the INC_CACHING_FILE_DATA_SOURCE option. If this option is disabled then it is assumed that read and seek operations on a file are very fast.

If the INC_CACHING_FILE_DATA_SOURCE is enabled then the `HttpdFileDataSource` object maintains a cache of buffers to avoid having to read continually from the file. This is especially desirable if `HttpdFile` objects have high overhead performing seeks and reads.

The `HttpdFileDataSource` arbitrates access to the underlaying file object so that the data source may be used in a thread-safe manner. This allows a single file object to service many threads using the data source; as is common when the data source is used to back an instance of the HttpdRomFileSystem.

If your platform provides either virtual memory (and an interface similar to POSIX `mmap()`) or has large amounts of directly addressable storage then consider using the HttpdMemoryDataSource class instead. Otherwise the `HttpdFileDataSource` object is doing the same work that the memory manager within the operating system is doing; and performing this work twice is less efficient.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Caching

The caching version of `HttpdFileDataSource` maintains a set of fixed-size cache buffers in a hash table for easy access. The hash is indexed on the logical address of the block and the ordering of the nodes within each bucket is explicitly from most recently used (list head) to least recently used (list tail).

Once the maximum number of buffer blocks are resident in the cache any access outside the resident blocks requires that a block be evicted. The eviction process rotates through all the buckets to avoid punishing any particular group of blocks. Buffers are evicted starting from the end of the list which is where the least recently used blocks reside.

In order to support mapped access (see `HttpdDataSource::AddressOf`) some blocks are "pinned" and are never removed from the cache until they are "un-pinned."

Tuning the cache is a matter of understanding the costs of the `HttpdFile` object backing the `HttpdFileDataSource` and the access pattern of the data source. In most cases the HttpdRomFileSystem package will be accessing the data source. The *ROM* filesystem performs two types of accesses: Small random accesses for searching the meta-data and large consecutive accesses to the data once found. The `HttpdRomFileSystem` class attempts to map data areas directly.

It is best to keep the cache block size (FILE_DATASRC_CACHE_SIZE) a multiple of the underlaying filesystem block size as well as a power of two (to avoid lengthy division). The FILE_DATASRC_MAX_PINNED setting should be increased for lots of concurrent access as multiple threads attempt to map different regions of data.

The FILE_DATASRC_HASH_BUCKETS and FILE_DATASRC_MAX_CACHE_BLOCKS parameters should be increased for large amounts of data.

Public Methods

`HttpdFileDataSource`

`**HttpdFileDataSource::HttpdFileDataSource** (HttpdFile *p_file);`

This function initializes a file data source backed by *`p_file`*. The file must be reserved exclusively for the use of the data source as long as the `HttpdFileDataSource` object exists. In addition the lifetime of *`p_file`* must meet or exceed the lifetime of the `HttpdFileDataSource` object.

`Create`

`int **HttpdFileDataSource::Create** (void);`

This method should be called once before the `HttpdFileDataSource` object is used. If this method returns failure then the object should not be used.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdContentSink` Reference

Introduction

The `HttpdContentSink` implements the interface of HttpdWritable. Data written to the `HttpdContentSink` is buffered up in memory for later use. The buffering is done in such a way that when the contents are written out to a sink (which is typically a socket) the writes are in chunks of `SINK_BUFFER_SIZE` bytes.

The most common use is to direct the output of an operation such that the output data is queued for later transmission. In fact, once stored within the `HttpdContentSink` the data can be sent multiple times.

Another common use of `HttpdContentSink` is to buffer up data so that a correct `Content-length` header can be sent as the result of an HTTP request. If a sink that can convert the content into a null-terminated C string is desired consider using a `HttpdStringSink`.

The `HttpdContentSink` class provides a guaranteed atomic behavior for writes. If a write will not succeed the stored content within the sink remains unchanged as if the `Write` method was not called. This behavior allows recovery from a failure when using the `HttpdContentSink`. In fact this is how the INC_BUFFER_OVERFLOW_RECOVERY option is implemented.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`ContentLength`

`size_t` **`HttpdContentSink::ContentLength`** `(void);`

This function returns the number of bytes queued in the `HttpdContentSink` at the current moment.

`SendData`

`int` **`HttpdContentSink::SendData`** `(HttpdWritable *`*`p_data`*`);`

This function writes the queued data to the interface specified by *`p_data`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Purge`

```
void HttpdContentSink::Purge (void);
```

This method removes all stored content from the sink.

`HttpdBatchWriter` Reference

Introduction

>

The `HttpdBatchWriter` implements a filter that can be applied to a `HttpdWritable` object to batch up smaller writes into larger ones.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Many TCP implementations do not perform well when many small writes of various sizes are performed on a socket. This can be avoided by enabling the Nagle algorithm although this results in higher latency. The `HttpdBatchWriter` is a filter that sits on top of a `HttpdSocket` sink and normalizes the size of the writes to the socket to `XFER_BUF_SIZE` bytes.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdBatchWriter`

```
HttpdBatchWriter::HttpdBatchWriter (HttpdWritable *p_target);
```

Initialize the batch writer. The batched data is written to *`p_target`* periodically.

`Flush`

```
int HttpdBatchWriter::Flush (void);
```

This method flushes the pending data that has not been batched yet.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

Pending data is not flushed if the `HttpdBatchWriter` is destroyed; although all allocated resources are released. Therefore it is important to call `Flush` before releasing this object in the event of successful request processing.

`HttpdNullSink` Reference

Introduction

The `HttpdNullSink` implements the interface of HttpdWritable. Data written to the `HttpdNullSink` is destroyed.

The most common use (from a user point of view) for this strange class is for multipart *MIME* file handling. To ignore a particular entity of a *MIME* multipart message, the address of an instance of this class can be passed to `HttpdBoundaryReader::Read` as the *`p_target`* parameter.

Because there is no instance specific data in this object, an instance of `HttpdNullSink` is available as a singleton from `HttpdNullSink::Null`. This instance is valid for the lifetime of the system and may be used any time after global constructors are executed.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`Null`

```
static HttpdWritable *HttpdNullSink::Null (void);
```

This function returns a writable object that simply ignores any data written to it. The pointer is never NULL.

`HttpdStringSink` Reference

Introduction

The `HttpdStringSink` implements the interface of HttpdWritable. Data written to the `HttpdStringSink` is stored as a regular, contiguous, zero-terminated string.

This class differs from a `HttpdContentSink` in that the buffer is contiguous and can be used as a null-terminated C string. If the content is only to be stored and then written out to another sink consider using a `HttpdContentSink` as it is more efficient.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

String

```
const char * HttpdStringSink::String const (void);
```

This function returns a (read-only) pointer to the current contents of the sink. If the sink is empty a pointer to the empty string is returned.

Buffer

```
char * HttpdStringSink::Buffer const (void);
```

This function returns a pointer to the current contents of the sink. Unlike the `String` method if the sink is empty the return value is undefined. This method is slightly more efficient than calling `String`, however.

TakeBuffer

```
char * HttpdStringSink::TakeBuffer (void);
```

This method removes the current string in the sink and reset the sink to contain an empty string. A pointer to the string that contains the contents of the sink. If the sink is empty, NULL is returned. After this call the string is owned by the caller. It is the responsibility of the caller to release it using HttpdOpSys::Free when no longer needed.

Length

```
size_t HttpdStringSink::Length (void);
```

This method returns the size (in bytes) of the string data. The return value is the number of bytes written to the buffer not the actual size of the buffer. The size of the buffer may be larger than the number of bytes written.

Clear

```
void HttpdStringSink::Clear (void);
```

This function removes any content written to the sink. The allocated buffer is not returned to the system however and is re-used when more data is written to the sink.

ClearAndRelease

```
void HttpdStringSink::ClearAndRelease (void);
```

This function removes any content written to the sink. Additionally, the allocated buffer is returned to the system.

Prepare

```
void HttpdStringSink::Prepare (size_t size);
```

This method pre-allocates *`size`* bytes of free space within the sink. This is useful for avoiding heap fragmentation if the size of the data being written to the sink is known in advance.

It is okay to call this method at any time during the life of the object. This method will never reduce the size of the allocated free space if it is greater than *`size`*.

`ReleaseBuffer`

```
void HttpdStringSink::ReleaseBuffer (void);
```

This method releases any unused buffer space in the sink. It is a good idea to call this method for long-lived sinks that will not be modified once built.

It is okay to call this method at any time during the life of the object. Writing additional data to the sink will simply reallocate the buffers if needed.

`HttpdBufferWriter` Reference

Introduction

The `HttpdBufferWriter` class is similar in purpose to the `HttpdStringSink` class. This class implements the interface of HttpdWritable such that data written to the interface is stored in a fixed size buffer. Attempts to write more data than the buffer has available results in an error. The buffer is not null terminated and is not allocated or managed by this class.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`Count`

```
size_t HttpdBufferWriter::Count const (void);
```

This function returns the number of bytes written into the buffer. The count is reset to zero when a new buffer is assigned.

`Buffer`

```
void HttpdBufferWriter::Buffer (void *p_buffer, size_t max_size);
```

This method assigns a new buffer as the target for written data. The *`p_buffer`* parameter points to the address of the new buffer. The *`max_size`* parameter is the maximum number of bytes that may be written into the buffer.

The buffer set by this method remains the target for writing until either the `HttpdBufferWriter` instance is destroyed or a new buffer is set with this method.

Setting a new buffer resets the number of bytes written counter

`HttpdFifo` Reference

Introduction

This class implements a dynamically sized buffer that can be used to capture streamed data for analysis. The expected use of this class is that data is removed from the buffer as it is processed. A typical example would be to process data in a streaming fashion where processing must be delayed until a certain amount of data has arrived.

Use of this class provides a very efficient solution to producer/consumer type problems. `HttpdFifo` implements the `HttpdWritable` interface in addition to a zero-copy interface that can directly access the internal buffer of the FIFO. The latter interface is ideal for processing data from a `HttpdReceiver` object efficiently.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

`HttpdFifo`

`HttpdFifo::HttpdFifo` `(size_t` *`initial_buffer`* `= 0, size_t` *`max_buffer`* `=` *`infinity`*`);`

This function constructs the `HttpdFifo` object. If *`initial_buffer`* is not `0` then this is the number of bytes allocated initially. If *`max_buffer`* is specified then this is the maximum amount of data that may be buffered before the methods of this object return an error condiition.

`AvailableWriteBuffer`

`size_t` **`HttpdFifo::AvailableWriteBuffer`** `const (void);`

This method returns the number of bytes that the current write buffer can take without a reallocating memory.

`TransferSize`

`size_t` **`HttpdFifo::TransferSize`** `const (void);`

This method returns the ideal size to use for the buffer window specified to the `GetWriteBuffer()` method. It avoids lots of repeated small (inefficient) writes by rounding the transfer size up if necessary.

GetWriteBuffer

`void *HttpdFifo::GetWriteBuffer (size_t window);`

This method returns a pointer to the buffer for writing data. The buffer will be at least *`window`* bytes in size. If NULL is returned then there is insufficient memory to open the buffer to the specified size.

After writing upto *`window`* bytes to the returned buffer `Produce()` should be called with the number of bytes actually written.

Produce

`int HttpdFifo::Produce (size_t count);`

This method registers that *`count`* bytes were written to the write buffer returned by `GetWriteBuffer()`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Used

`size_t HttpdFifo::Used const (void);`

This method returns the number of bytes available in the FIFO for reading (i.e. used buffer space).

ReadData

`void *HttpdFifo::Used const (void);`

This method returns a pointer to the FIFO data. The pointer points to all of the data available: the number of bytes returned by `Used()`.

Consume

`void HttpdFifo::Consume (size_t count);`

This method removes *`count`* bytes from the FIFO. Typically this method is invoked after processing data accessed by the pointer returned from `ReadData()`.

Read

`size_t HttpdFifo::Read (void *p_buffer, size_t count);`

This method moves up to *`count`* bytes from the FIFO to *`p_buffer`*. If fewer than *`count`* bytes are available then the actual number of bytes read is returned.

ReleaseBuffer

`void HttpdFifo::ReleaseBuffer (void);`

This method releases any memory allocated by the FIFO if the FIFO is empty. The FIFO can be used after this at which point it will reallocate the buffer automatically.

Finish

`int` **`HttpdFifo::Finish`** `(void);`

This method is to be called when no more data is written to the tokenizer. In this class it simply returns `0` but subclasses may override it to provide additional functionality.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

ReadBody

`int` **`HttpdFifo::ReadBody`** `(HttpdRequest *`*`p_request`*`, unsigned int `*`time_out`*` = HTTPD_CGI_TIMEOUT);`

This method reads an document that is provided with the entity body of *`p_request`*. If the entity body is fully digested then `Finish()` is automatically called.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

HttpdCountingSink Reference

Introduction

The `HttpdCountingSink` implements the interface of HttpdWritable. Data written to the `HttpdCountingSink` is discarded but a running total of the number of bytes written is kept. This class is especially useful when generating the `Content-Length` headers.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

WrittenSize (Getter)

`size_t` **`HttpdCountingSink::WrittenSize`** `(void); const`

This method determines how many bytes have been written into the sink object.

WrittenSize (Setter)

`void` **`HttpdCountingSink::WrittenSize`** `(size_t `*`sz`*`);`

This method sets the current byte count of the sink.

HttpdChunkedSink Reference

Introduction

The `HttpdChunkedSink` implements the interface of HttpdWritable. Data written to the `HttpdChunkedSink` is reformatted to the HTTP chunked transfer encoding.

This transfer encoding is only necessary for dynamically generated content where the length is unknown before it is generated. Unless chunked encoding is used, persistent connections can not be maintained with dynamically generated content.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

HttpdChunkedSink

```
HttpdChunkedSink::HttpdChunkedSink (HttpdWritable *p_out);
```

This constructor initializes the sink. The output of the sink is sent to *`p_out`*. This parameter should almost always be the socket from a request object.

Open

```
int HttpdChunkedSink::Open (void);
```

This method should be called before any data is written to the object. Not calling this method when a sink is not used avoids memory waste for the chunking buffers.

An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned. However, this error code can be ignored as it is not fatal. Should `Open` fail writes to this object will simply return an error.

Finalize

```
int HttpdChunkedSink::Finalize (void);
```

This method should be called after all data is written to the object. No more data should be written after `Finalize` is called. The only reason for not calling this is if the socket the sink is attached to is being abandoned due to error.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdRomFileSystem` Reference

Introduction

`HttpdRomFileSystem` provides an interface for the abstract class `HttpdFileSystem` which provides an abstract interface for a file system. The structure of the filesystem is stored in a packed form generated by the SCPG tool.

The packed content generated by SCPG must be stored in some form of read-only storage and provided to the `HttpdRomFileSystem` via a HttpdDataSource class.

This class can be used independently of Seminole. However, the file system semantics implemented by this class are really oriented for HTTP style transactions. Files are directly associated with *MIME* types and there is no concept of a "current working directory." The *ROM* filesystem is designed to have full path names for the most efficient file lookup.

The *ROM* filesystem also allows named attributes on a per-file basis if the INC_ROM_ATTRIBUTES configuration option is enabled.

Even if your embedded platform has a flash filesystem it is probably optimal to use a `HttpdRomFileSystem` contained in a single file holding all of the web content. There are several reasons for this:

- Many flash filesystems do not deal well with the kinds of access patterns that HTTP requests generate.
- Websites are composed of many small files. Sophisticated flash filesystems that perform wear-leveling and bad block handling (e.g. YAFFS or jffs2) keep a large amount of meta-data per file. Storing web content in these filesystems can waste a large amount of space.
- `HttpdRomFileSystem` provides highly optimized versions of the `PushToSink` and `PushFileSegment` methods. These operations are fundamental to web serving.
- Traditional filesystem semantics (e.g. POSIX) do not keep track of content types or other meta data while `HttpdRomFileSystem` does.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

`HttpdRomFileSystem` is thread safe provided the underlaying data source is reentrant. For performance reasons there is no locking within `HttpdRomFileSystem`. Therefore multiple threads may be opening files against the `HttpdRomFileSystem` although each individual open file object may only be used by one thread at a time.

Public Methods

`Mount`

```
int HttpdRomFileSystem::Mount (HttpdDataSource *p_source);
```

This method should be called once after the construction of the `HttpdRomFileSystem`. Given a valid *ROM* file system image contained in *`p_source`*, the *ROM* filesystem becomes active.

No accesses to the filesystem should be made until it is mounted without error.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdReceiver` Reference

Introduction

`HttpdReceiver` is an abstract interface that is used to read data from an HTTP inbound transfer. It has similar methods to a socket for reading data. This interface is mainly used when reading data from `POST` requests.

Public Methods

`HttpdReceiver`

`HttpdReceiver::HttpdReceiver` `(HttpdSocket &`*`p_request`*`);`

This function constructs the abstract portion of the `HttpdReceiver` object.

`ReadUntil`

`bool` **`HttpdReceiver::ReadUntil`** `(char` *`term`*`, char *&`*`p_buffer`*`, size_t` *`bufsz`*`, unsigned int` *`timeout`*`);`

This function reads bytes from the transfer until either *`term`* is seen; in which case *`term`* is not stored in the resulting buffer. The method returns if the timeout period elapses.

In order to avoid excessive memory allocations on entry *`p_buffer`* should point to a statically allocated buffer that is *`bufsz`* bytes in size. If the amount of data to be read exceeds the statically allocated buffer size then a dynamic buffer will be allocated. When this method returns if *`p_buffer`* no longer points to the statically allocated buffer then it must be freed by the caller.

This method returns true if at least one byte of data was returned. The returned data is terminated by a zero byte. If no data was received then false is returned.

`Read`

`int` **`HttpdReceiver::Read`** `(void *`*`p_buf`*`, size_t &`*`nbytes`*`, unsigned int` *`timeout`*`);`

This function reads upto *`nbytes`* from the transfer into *`p_buf`*. If no data is received for *`timeout`* seconds then an error is returned. Upon successful return *`nbytes`* will be set to the number of bytes read, which may be less than the requested amount.

`Pump`

`int` **`HttpdReceiver::Pump`** `(HttpdFifo *`*`p_fifo`*`, unsigned int` *`timeout`*`);`

This function transfers all of the received data into *`p_fifo`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

ReadN

int **HttpdReceiver::ReadN** (void **p_buf*, size_t *nbytes*, unsigned int *timeout*);

This function reads exactly *nbytes* from the transfer into *p_buf*. If not enough data is received for *timeout* seconds then an error is returned.

Gets

int **HttpdReceiver::Gets** (char **p_buf*, size_t *nbytes*, unsigned int *timeout*);

This function reads one line from the receiver, assuming that each line is terminated by a newline character (ASCII line feed). The value of *maxbuf* should be the size of the buffer, *p_buf*.

As with the `HttpdSocket` version of this method, it is worth noting that a carriage return may be embedded in the buffer, as `Gets()` does not purge them.

If the entire line is not received by the specified timeout (in seconds) then `HttpdOpSys::ERR_NOTREADY` is returned. If the line would exceed the available buffer size then `HttpdOpSys::ERR_LIMITRCHD` is returned. For an empty string `HttpdOpSys::ERR_BADFORMAT` is returned.

Callers should keep in mind that it is possible that this method returns some other error code surfaced from the underlying socket layer.

More

bool **HttpdReceiver::More** (unsigned int *timeout*);

This function returns true if there is likely to be more data available in the transfer. If the transfer is complete then false is returned.

Pump

int **HttpdReceiver::Pump** (HttpdWritable **p_sink*, unsigned int *timeout*);

This function transfers the body of the HTTP transaction (the received content) into *p_sink*. If data is not received in *timeout* seconds then the transfer is aborted with an error.

If successful `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdBoundaryReader` Reference

Introduction

The `HttpdBoundaryReader` class is used for processing *MIME* multipart messages. These are used for encapsulating many kinds of data; in particular, HTTP file uploads are done using multipart *MIME*. Multipart *MIME* separates components with a unique boundary string that is obtained from encapsulation headers. This class does not parse these headers.

Instances of `HttpdBoundaryReader` are associated with a receiver and can be used either by “pulling” the data or pushing the data into a subclass of HttpdWritable.

Public Methods

`HttpdBoundaryReader`

`HttpdBoundaryReader::HttpdBoundaryReader` `(HttpdReceiver *`*`p_receiver`*`, const char *`*`p_boundary`*`, int &`*`rc`*`);`

The `HttpdBoundaryReader` must be provided with a reference to the receiver to read from (*`p_receiver`*) and the boundary string (*`p_boundary`*). The *`rc`* parameter is set to an error status after the constructor returns if there was a problem initializing.

Note

The boundary should be found (using HttpdUtilities::FindBoundary before this class is constructed.

Read (pull model)

`int` **`HttpdBoundaryReader::Read`** `(const void *&`*`p_buffer`*`, size_t &`*`len`*`, unsigned int` *`timeout`*`);`

This function reads from the associated receiver, waiting for up to *`timeout`* milliseconds for data.

If there is data to be read then *`len`* is set to the number of bytes that were read and *`p_buffer`* is pointed to the data and 0 is returned.

The returned pointer is valid until the next call to this method or the destruction of the object.

If the boundary is found then `HttpdBoundaryReader::HTTPD_MIME_BOUNDARY` is returned. In this case callers should call HttpdUtilities::IsLastBoundary to complete the boundary parsing and determine if another part of the multipart entity is present.

Otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Read (push model)

`int` **`HttpdBoundaryReader::Read`** `(HttpdWritable *`*`p_target`*`, unsigned int` *`timeout`*`);`

This method writes the contents of the current part of the multipart message into *`p_target`*. If no data is received in *`timeout`* milliseconds the operation is aborted and an error code is returned.

As with the pull version of `Read` if success is returned (a return value of 0) then HttpdUtilities::IsLastBoundary should be called to complete the boundary parsing.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

Data is written into the sink in chunks no larger than the boundary size. If the target stream does not perform well with small writes then the HttpdBatchWriter class can be used to increase the write size.

`HttpdMuxFileSystem` Reference

Introduction

To support the modular construction of systems, the `HttpdMuxFileSystem` class allows multiple separate `HttpdFileSystem` objects to be combined into a single object. Each filesystem is addressed with a specific prefix that is assigned at registration time. `HttpdMuxFileSystem` provides an abstract interface for a file system although the `OpenFile` method is not used. The `HttpdMuxFilesystem` class fills in the correct filesystem in the `HttpdFileInfo` object when the `HttpdMuxFilesystem::FileInfo` method is called.

A good example for the use of this class is an embedded device with slots that allow additional modules to be inserted. It would be convenient if each module could contain its own filesystem image for configuration of its specific parameters. With this approach new modules can be developed without even having to update the software on the embedded device.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`Mount`

```
int HttpdMuxFileSystem::Mount (const char *p_prefix, HttpdFileSystem
*p_fs);
```

This method adds *`p_fs`* to the translation table addressed by the prefix string in *`p_prefix`*.

No accesses to the filesystem should be made until all calls to `Mount` complete without error. Once the filesystem is accessed no more prefixes should be added.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Chapter 4. Portability Layer Reference

Platform Specific Definitions

The definitions in `sem_sys.h` are specific to the target OS. The specifics of the TCP implementation are defined in `sem_syssock.h`. Seminole comes with reference implementations for several operating systems. The reference implementations do not have to be used, and the class specifications in this section can be implemented in any way necessary.

It is also okay to change the reference code to work around any special platform needs. Unless the target platform is very different from any of the other targets it is suggested that an existing portability layer be taken as a base when attempting a new port. The source code for all of the reference applications are in `src/targets/`*`OS-NAME`*. New portability layers should be placed in the same parent directory as the existing ones.

Although it is not necessary, application code may also make use of the portability layer if desired. It is also not strictly necessary for the portability layer to use the operating system to implement its services. For example instead of implementing the memory allocation service in terms of `malloc` and `free` the `HttpdMemoryAllocator` class within Seminole can be used to allocate from a statically declared chunk of storage.

`HttpdOpSys` Reference

Introduction

`HttpdOpSys` serves as an abstraction layer between Seminole's platform-independent code and the specific interfaces offered by the host operating system. Primarily, it provides generic memory and process management facilities, since these are the most basic requirements for Seminole. Like the `HttpdUtilities` class all members of this class are static and there is no need to ever instantiate this class.

More detailed architectural discussion of Seminole's portability mechanisms can be found in the section called “Operating Environment Abstraction Layers”.

When requesting services from the underlying host operating system through `HttpdOpSys`, it is possible for internal errors to eventually be returned to the caller within Seminole. However, these errors are abstracted to generic equivalents which are descriptive of the error condition, but do not depend on any platform-specific representation. The possible OS errors are listed in Table 4.1, “OS Abstraction Layer Error Codes”.

Table 4.1. OS Abstraction Layer Error Codes

Constant	Meaning
`ERR_NOTFOUND`	File, directory, or entity not found
`ERR_SYSPERM`	Administrative permission denied
`ERR_NOTREADY`	Device, resource, or unit not ready
`ERR_LIMITRCHD`	Maximum limit or capacity reached
`ERR_IO`	Low-level or hardware I/O error
`ERR_WRONGTYPE`	Inappropriate type or target for operation

Constant	**Meaning**
`ERR_OUTOFMEM`	Ran out of memory
`ERR_BADPARAM`	Invalid or out-of-range parameter
`ERR_BADFORMAT`	The provided data is corrupted or not in a valid format.
`ERR_NOSPACE`	There is insufficient permanent storage to complete this operation.
`ERR_UNKNOWN`	Unknown/untranslatable error
`ERR_USER`	This is the base number for error codes in components that use `HttpdOpSys`. Some components in Seminole need these numbers to return extended error or status codes and use this number as a starting base.

Public Methods

`Init`

`int` **`HttpdOpSys::Init`** `(void);`

This static method initializes the operating system abstraction layer. No other services from `HttpdOpSys` can be utilized before this method is called and returns success.

Returns an error code from Table 4.1, “OS Abstraction Layer Error Codes” on failure or zero on success.

Note

This method does not have to be idempotent. It is called once and only once by `Httpd::Init`.

`Malloc`

`void *`**`HttpdOpSys::Malloc`** `(size_t` *`sz`*`);`

Allocate new memory *`sz`* bytes in length.

Returns a pointer to a buffer of at least the requested size, taking into account host alignment requirements, or NULL upon error.

If desired implementations can make use of the HTTPD_MALLOC_RETRY_LOOP and HTTPD_MALLOC_RETRY_TAIL macros to add a retry mechanism for allocations. These macros clear the allocation caches if INC_ALLOCATION_CACHE_PURGE is enabled. A typical implementation in the portability layer would be:

```
void *HttpdOpSys::Malloc(size_t size)
{
  HTTPD_MALLOC_RETRY_LOOP
  {
    void *p_buffer = malloc(size);
    if (p_buffer != NULL)
      return (p_buffer);
    HTTPD_MALLOC_RETRY_TAIL
  }

  return (NULL);
}
```

Free

`void` **`HttpdOpSys::Free`** `(void *`*`p_ptr`*`);`

Release a block of allocated memory pointed to by *`p_ptr`*.

Realloc

`void *`**`HttpdOpSys::Realloc`** `(void *`*`p_oldptr`*`, size_t` *`newsz`*`);`

Expand or shrink the size of the memory block pointed to by *`p_oldptr`*, to be *`newsz`* bytes in length.

Returns a revised pointer upon success, or NULL upon failure.

Important

If `Realloc()` fails to change the size of a given block of memory, the original block is invalidated and cannot be used. Therefore, if the block being resized has pointers to other objects embedded within, it is better to use the `SafeRealloc` method and explicitly free the original pointer.

As with `Malloc()` implementations may opt to use the HTTPD_MALLOC_RETRY_LOOP and HTTPD_MALLOC_RETRY_TAIL macros to add a retry mechanism.

SafeRealloc

`void *`**`HttpdOpSys::SafeRealloc`** `(void *`*`p_oldptr`*`, size_t` *`newsz`*`);`

This method is similar to `Realloc`. It expands or shrinks the size of the memory block pointed to by *`p_oldptr`*, to be *`newsz`* bytes in length.

Returns a revised pointer upon success, or NULL upon failure.

Unlike `Realloc`, if this method returns NULL the original block pointed to by *`p_oldptr`* is not released. It must be explicitly released. This gives callers a chance to perform further cleanup before releasing the allocated memory block.

Fork

`bool` **`HttpdOpSys::Fork`** `(void (*`*`p_func`*`)(HttpdParameter p1, HttpdParameter p2, HttpdParameter p3), HttpdParameter` *`p1`*`, HttpdParameter` *`p2`*`, HttpdParameter` *`p3`*`, HttpdParameter` *`p3`*`, HttpdPriorityHint` *`pri_hint`*`);`

Create a new process, job, or task (depending on the host platform), which will immediately enter the function *`p_func`*, passing it *`p1`*, *`p2`*, and *`p3`* as arguments.

`pri_hint` serves as a characterization to the underlying operating system of the type of work the new thread of execution will be performing, so that it can be scheduled accordingly. The behavior this hint

elicits is completely dependent on the operating system abstraction layer being used; while all layers must support the standard values of *`pri_hint`*, they are not actually required to take any action on it. These standard values are listed in Table 4.2, “Fork() Priority Hints”. The existence of other values should not be relied upon, since the operating system abstraction layer is only required to support the listed values.

Table 4.2. Fork() Priority Hints

Constant	Meaning
`HTTPD_PRI_WORKER`	Standard worker thread
`HTTPD_PRI_ACCEPTOR`	Webserver connection acceptor thread
`HTTPD_PRI_SESSION_SCRUBBER`	Session table scrubbing thread
`HTTPD_PRI_DISCOVERY`	Discovery server thread

Returns true upon successful process creation, false upon failure.

TaskSleep

```
void HttpdOpSys::TaskSleep (unsigned int msec);
```

This method suspends the calling thread for *`msec`* milliseconds. Ideally, the operating system should schedule other tasks during the interval.

Now

```
void HttpdOpSys::Now (HttpdOpSys::TimeStamp &now);
```

This method obtains the current time as measured from some arbitrary epoch and places it into *`now`*. This notion of “current time” is not necessarily connected with the actal wall-clock time. Instead it is used to measure time realtive to other values of the same clock.

On some systems the wall-clock time is either not available or changes in a manner that does not reflect the passage time (e.g. is periodically adjusted to some other reference clock). In these cases this routine can be implemented to provide a “pure” time measurement source.

The TimeStamp type must be defined by `HttpdOpSys` as an abstract type that can represent this measured time.

DiffTime

```
int HttpdOpSys::DiffTime (const HttpdOpSys::TimeStamp &t1, const
HttpdOpSys::TimeStamp &t0);
```

This method computes the signed difference, in seconds, between the time values given by *`t1`* and *`t0`* by subtracting *`t0`* from *`t1`*.

This method is similar to the standard library routine `difftime` except that it does not return a floating point value (which is frequently inappropriate for embedded systems).

Randomize

```
void HttpdOpSys::Randomize (void);
```

This method is called by Seminole just before it is about to obtain entropy (via the `HttpdOpSys::Entropy` method) to potentially give some additional randomness to the obtained data.

In particular the timing of when this function is called is typically a function of the requests delivered to Seminole Although this is not an ideal source of entropy (since it can be manipulated externally) in systems with few other sources of entropy it can be helpful.

Note

This function may be called from multiple threads because it is not called at startup but rather when a stream of entropy is needed.

Entropy

`unsigned int` **`HttpdOpSys::Entropy`** `(unsigned int` *`max_val`*`);`

This method should return a random value between `0` and *`max_val`* (inclusive). Ideally the data should be totally random as it may be used for cryptographic purposes. However, the only real source of true randomness is from specialized hardware (such as an avalanche noise). For cost-sensitive applications it may be necessary to gather entropy from other sources such as the time between keypresses or the input of an analog-to-digital converter.

Therefore the implementation of this method (and the associated `HttpdOpSys::Randomize` method) is considered to be very platform specific.

NativeFileSystem

`HttpdFileSystem *` **`HttpdOpSys::NativeFileSystem`** `(void);`

Some operating systems have their own native file systems (well, most actually). On these systems, the native filesystem is abstracted as a file system interface (`HttpdFileSystem`).

On operating systems that do not have a native file system available this routine shall return NULL.

OpenSystemFile

`int` **`HttpdOpSys::OpenSystemFile`** `(const char *`*`p_filename`*`, HttpdDataSource *&`*`p_source`*`);`

This routine maps a native operating system file to a HttpdDataSource abstraction.

The address of the created data source object is placed in *`p_source`*. When the object is no longer needed it should be released with HttpdOpSys::CloseSystemFile.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This routine is optional and does not have to be implemented. If implemented, the symbol HTTPD_HAVE_NATIVE_FILE_SOURCES should be defined to a non-zero (true) value.

CloseSystemFile

`void` **`HttpdOpSys::CloseSystemFile`** `(HttpdDataSource *`*`p_source`*`);`

This releases the file mapping created with HttpdOpSys::OpenSystemFile. The data source pointed to by *`p_source`* is no longer valid after this method is called.

Note

This routine is optional and does not have to be implemented. If implemented, the symbol HTTPD_HAVE_NATIVE_FILE_SOURCES should be defined.

Public Data

`HttpdOpSys` contains no publically accessible data members.

`HttpdTcpSocket` Reference

Introduction

The `HttpdTcpSocket` class provides an implementation of the required interface for a *transport* abstraction. It inherits the interface defined by HttpdSocketInterface. In particular, the TCP protocol is implemented via this interface.

Seminole does not provide its own TCP/IP stack. It is expected to be part of the host operating system or support package. Thus, this class is not defined as part of Seminole proper. Rather, it is part of the portability layer.

`HttpdSslSocket` Reference

Introduction

The `HttpdSslSocket` class provides an implementation of the required interface for a *transport* abstraction. It inherits the interface defined by HttpdSocketInterface. In particular, the SSL protocol is implemented via this interface.

Seminole does not provide its own SSL stack. Thus, this class is not defined as part of Seminole proper. Rather, it is part of the portability layer.

On most platforms, the OpenSSL [http://www.openssl.org/] library is used. The `HttpdSslSocket` interface uses the primitives of the OpenSSL™ library to manage secure connections.

Unlike normal TCP traffic, SSL traffic requires lots of configuration information. In particular, digital certificates and keys must be provided to the SSL engine. These are passed through the *`pp_options`* parameters to the `Listen` and `Connect` methods.

For OpenSSL™ implementations of `HttpdSslSocket` the following parameters can be specified:

Table 4.3. OpenSSL Socket Options

`key:`*`filename`*	Specify the RSA keys. This key is used for the certificate validation as well as encryption of the session if ephemeral keying is not used. The key file should be in the PEM format.
`cert:`*`filename`*	Specify the digital certificate used to identify the server. The “common name” field of the certificate should be the hostname that the server is addressed by. The certificate file should be in the PEM format.

`pem:`*`filename`*	Specify a PEM file containing both the RSA keys and the certificate.
`cipher:`*`cipher selection`*	Specify the suite of ciphers to use. This is the list parsed by the OpenSSL™ `SSL_CTX_set_cipher_list` function.
`dh-512:`*`filename`*	For ephemeral keying the Diffie-Hellman key-agreement protocol is used. This protocol requires some specific random numbers that are computationally intensive to generate. This option loads the 512-bit version of the parameters from the specified PEM file.
`dh-1024:`*`filename`*	This specifies the 1024-bit version of the Diffie-Hellman key used for ephemeral keying of the session.
`dh-reuse:`	Reuse Diffie-Hellman keys. This adds security at the expense of CPU time. In most cases the added security benefits out weigh the additional overhead. However this option may be specified on especially low-end processors to quicken response times.
`rand-egd:`*`EGD socket path`*	Load entropy (randomness) from a socket managed by an EGD daemon. This is only supported on POSIX platforms.
`rand-file:`*`size,filename`*	Load entropy from a file. The size (and comma) are optional. If specified only that many bytes will be read from the file. If the size is not specified the contents of the entire file are analyzed. A good example for the use of the size would be the `/dev/urandom` device available on some POSIX systems. This device generates an endless source of entropy so the size must be specified or else the server will never start.

Although the configuration of SSL may seem daunting at first, the makecert tool automatically generates most of the files needed to support SSL

`HttpdMutex` Reference

Introduction

Instances of `HttpdMutex` are used to protect shared objects from the effects of being accessed by multiple threads.

Note

The implementation of `HttpdMutex` is provided by the portability layer. It is important to keep in mind that under some operating systems mutexes may not be “recursive.”

A recursive mutex (also called a “counting” mutex) is one that can be taken by a thread that already owns the mutex without deadlocking. Seminole does not require that mutexes are recursive — however the target platform may only provide for recursive mutexes. For maximum portability code that makes use of `HttpdMutex` objects should not assume they can be taken recursively.

Public Methods

HttpdMutex

HttpdMutex::HttpdMutex (void);

This initializes the mutex object. The mutex object is not usable though until the `Create` method is called.

~HttpdMutex

HttpdMutex::~HttpdMutex (void);

Release all associated resources with the mutex object. It is important that the mutex not be obtained when it is destroyed.

Create

int **HttpdMutex::Create** (void);

This method should be called once after construction of the object. It registers the mutex object with the operating system and must be called before the `Lock` or `Unlock` methods are called.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Lock

void **HttpdMutex::Lock** (void);

This method requests exclusive access to the mutex (and the object protected by the mutex). Only one thread at a time will return from this call. The rest will remain queued until the mutex is unlocked.

Unlock

void **HttpdMutex::Unlock** (void);

This method releases exclusive access to the mutex. If other threads are pending on access, another thread should be allowed to take the mutex as this thread releases it. It is up to the scheduling policy of the host operating system to determine when threads are allowed to obtain the mutex.

If the host operating system provides per-mutex selectable scheduling policies then in general a FIFO scheduling policy is the best for Seminole. Priority inversion protection is also not really required and may be disabled on mutexes used by Seminole if it provides a performance boost.

HttpdEventSemaphore Reference

Introduction

Instances of `HttpdEventSemaphore` is used to allow one thread to wait for a signal from another thread.

Note

This class only exists if the portability layer defines HTTPD_HAVE_THREADS to a non-zero value.

Public Methods

`HttpdEventSemaphore`

`HttpdEventSemaphore::HttpdEventSemaphore` `(void);`

This initializes the semaphore object. The semaphore object is not usable though until the `Create` method is called.

`~HttpEventSemaphore`

`HttpdEventSemaphore::~HttpdEventSemaphore` `(void);`

Release all associated resources with the semaphore. No threads should be waiting on the semaphore when it is destroyed.

`Create`

`int` **`HttpdEventSemaphore::Create`** `(void);`

This method should be called once after construction of the object. It registers the semaphore object with the operating system and must be called before the `Wait` or `Signal` methods are called.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Wait`

`void` **`HttpdEventSemaphore::Wait`** `(void);`

This method suspends the calling thread until the semaphore object is signaled (via the `Signal` method). Only one thread at a time should wait on the semaphore object.

Once released the semaphore is reset to a non-signaled state.

`Wait` (with timeout)

`int` **`HttpdEventSemaphore::Wait`** `(unsigned long` *`msec`*`);`

This method suspends the calling thread until the semaphore object is signaled (via the `Signal` method) or *`msec`* milliseconds have elapsed. Only one thread at a time should wait on the semaphore object.

Once released the semaphore is reset to a non-signaled state.

Upon success, `0` is returned. If the wait times out, the value `HttpdEventSemaphore::ERR_TIMEOUT` is returned. If an operating system error prevents the operation from succeeding, then a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Signal

```
int HttpdEventSemaphore::Signal (void);
```

This method allows the waiting thread to continue executing. If a thread is not yet waiting on the semaphore object then the semaphore object is marked as signaled and the thread will not be suspended during the `Wait` method.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Chapter 5. Generating Dynamic Content with Templates

Understanding the Template Engine

Why Templates?

Although other methods exist for generating dynamic content they are often difficult to modify and bulky. Templates enforce a strict separation between content and code. This separation is especially important when the content developer is not the same person as the engineer writing the code backing the application.

Some template systems make the mistake of providing almost a complete programming language. This gives template authors too many freedoms and often results in a large amount of the program logic within the template. The Seminole template system provides only three constructs: substitution, conditional inclusion, and iteration. This keeps the content developers “honest” by forcing the actual program logic to reside in the application layer.

Sometimes it is okay to break the rules. For these cases Seminole includes many pre-built template commands that are quite flexible and generic. The penalty for breaking the rules and using these pre-built commands is twofold. First, there is the increase in code size brought about by the generic code. Second is the increased CPU overhead used during formatting. It is up to the designer of the content to determine if the tradeoff is worthwhile.

Compiled Templates

For efficiency and reliability reasons, templates in Seminole are compiled. The template-specific markup is processed by SCPG and encoded into a binary form. The binary form is then executed by `HttpdTemplateProcessor` and its supporting classes.

This results in much more efficient template execution because the portions of the template skipped due to conditional evaluation do not have to be parsed. In traditional template systems that process the template file, all portions (even those that are in the false part of a conditional) must be parsed.

Template Syntax

Template directives are denoted by the `%{` opening token and the `}%` closing token. These directives can appear anywhere in the template file. Therefore they can even be used inside of quoted HTML attributes without problems.

Table 5.1. Template Directives

Directive	Description
`eval`	This is the most basic directive. It is used like a function call in procedural languages. The C++ code can substitute any string for this directive. Of course, it can also be used to perform an operation and substitute the empty string for this directive. The directive must be followed by a symbol name to identify the operation:

Directive	Description
	`Hello user %{eval:username}%.<p>`

`loop`	This directive is used to repeat a body of the template zero or more times. The loop body can contain text and any other directives.
	
	`<h2>User names</h2>`
	`<ul>`
	` %{loop:usertable}%`
	`  <li>%{eval:username}%`
	` %{endloop}%`
	`</ul>`
	
	Loops are often used in conjunction with `eval` directives because loops can bind certain variables (such as username above) as a loop index.

`if`	Templates can have conditionals with the `if` directive:
	
	`The system stores data on a flash chip`
	`%{if:has_hard_disk}%and hard disk%{endif}%.`
	
	Conditionals can also have if-else blocks as well:
	
	`You must connect a the system to`
	`%{if:ethernet_model}%`
	` an ethernet`
	`%{else}%`
	` a token ring`
	`%{endif}%`
	`network uplink.`
	
	If-else chains can even be done:
	
	`You must connect the system to`
	`%{if:ethernet_model}%`
	` an ethernet`
	`%{elseif:token_ring_model}%`
	` a token ring`
	`%{elseif:bri}%`
	` an ISDN basic rate`
	`%{else}%`
	` a magical`
	`%{endif}%`
	`  network uplink.`

Directive	Description
`ifnot`	This conditional executes the contents of its body if the specified condition is not true: `The system is %{ifnot:ready}% busy%{endif}%.` No else clause can be used with this statement.

Some template directives can contain name/value attribute pairs just like an HTML tag. In these cases the syntax and quoting rules are similar to HTML. The following tags can have attributes:

- eval
- loop
- if
- else
- elseif

For example:

```
Your password is %{eval:password
                    set_insecured  = 1
                    comment        = "the magic word"
                    tagtype        = "&lt;a href&gt;"
                    salt           = "&#65;&#66;&#67;"}%
```

There are four attributes associated with the `eval` of password. Notice that HTML quoting rules apply. However, only a small subset of entity names are allowed:

- `&quot`
- `&lt`
- `&gt`
- `&amp`
- `&#XXX` (Character `XXX`)

Programming Template Interfaces

All of the definitions for the template processor are in the `sem_template.h`. This file automatically includes `seminole.h` if it has not been included already.

All of the names referenced in template directives must eventually reference some application specific code in C++. Each of these directives get instantiated into an object when interpreted. This “command object” is then passed to a method in the class `HttpdSymbolTable`. There is one receiver method for each command type, `HandleEval`, `HandleLoop`, `HandleCond` for each symbol table. These various methods are overridden in subclasses to implement the specific operations that templates can employ.

Rather than maintain one instance of `HttpdSymbolTable` (or subclass), the template processor maintains a stack of them. This provides a simple scoping mechanism that is especially useful for loops. A particular name is sent to each instance starting from the most recently added to the least recently added until it is properly handled.

Each symbol table in the chain is given a chance to either handle the symbol, fail the request with a fatal error or let outer symbol tables attempt to resolve the symbol. If no symbol table resolves the symbol processing of the template is halted and an error is returned.

To handle a symbol implementations of `HandleEval` or `HandleLoop` should return 0 while `HandleCond` should return either `HTTPD_TEMPLATE_FALSE_CASE` or `HTTPD_TEMPLATE_TRUE_CASE`. To stop any further searches for the name and fail the template processing any of the symbol table methods can return `HTTPD_TEMPLATE_UNKNOWN_NAME`. If a symbol table method wishes to continue the search to outer symbol scopes `HTTPD_TEMPLATE_NOT_HANDLED` should be returned.

The name in the template file can also be prefixed with one or more carets (`^`) to indicate previous levels of lexical scope. For example, if the current loop defines an evaluation label of username and we are nested in this loop three times then we can get to the username of the first loop with a `^^`:

```
%{loop:user_table}%
 …
 %{loop:user_table}%
 …
  %{loop:user_table}%
  …
  The current top-level user is %{eval:^^username}%.
  …
  %{endloop}%
 …
%{endloop}%
…
%{endloop}%
```

Referencing previous scopes can be helpful when more than one scope handles the same name.

The stack of symbol tables is maintained by a helper class called `HttpdTemplateScope`. This class uses constructors and destructors to keep the template scope in sync with C++ lexical scope. Another helper class, `HttpdSymbolMap` provides easy access to select C++ variables from templates. A combination of these two classes, `HttpdScopedSymbolMap` provides the combined functionality of `HttpdTemplateScope` and `HttpdSymbolMap`.

The error code `HTTPD_TEMPLATE_UNKNOWN_NAME` should be returned by the top-most symbol table if the named action does not exist. Although if no symbol table handles the request an error of `HTTPD_TEMPLATE_UNKNOWN_NAME` will be returned.

Returning `HTTPD_TEMPLATE_UNKNOWN_NAME` causes the template engine to stop searching any further for a symbol table willing to handle the symbol. The `HTTPD_TEMPLATE_NOT_HANDLED` return code indicates that a symbol table does not handle this name however the search should also be applied to previous scopes.

When using templates with the `HttpdFileHandler` request handler, the top-most symbol table is already implemented with a few bonuses as well. This `HttpdFSTemplateShell` handles file service requests and provides processing for include files.

`HttpdFSTemplateShell` also provides a static helper routine, called `Execute` that can be called from the `DoFile` phase of `HttpdFileHandler`. This helper handles all of the setup work necessary to execute a template from a subclass of `HttpdFileHandler`.

The demonstration code (`main.cpp`) provides a good example of subclassing `HttpdFileHandler` to add both authentication and template processing. The *MIME* type `x-server-internal/template` should be used to identify files that require template processing. However, this is only a convention and it can be circumvented if necessary. In fact, the entire symbol table can be made different based upon *MIME*.

`HttpdSymbolTable` Reference

Introduction

The `HttpdSymbolTable` class is a base class that accepts template commands and executes them. The default implementation simply returns HTTPD_TEMPLATE_NOT_HANDLED for all commands.

This class is designed to be subclassed and to handle application specific actions during template processing. Only the methods for the commands that must be handled need to be overridden.

The typical method for implementing one of the handler methods is to call the `Name` method of the supplied command pointer and then determine if this is one of the names that should be handled. It is probably best to implement this as a simple chain of if-else statements:

```
int MySymbolTable::HandleEval(HttpdEvalCommand *p_eval)
{
  const char *p_name = p_eval->Name();

  if (strcmp(p_name, "user_name") == 0)
    return (DoUserName(p_eval));
  else if (strcmp(p_name, "home_dir") == 0)
    return (DoHomeDir(p_eval));
  else
    return (HTTPD_TEMPLATE_NOT_HANDLED);
}
```

Because overriding this class for each scope can cause quite a few classes to be defined, for the simple cases of accessing a variable the `HttpdSymbolMap` helper class can be used instead of subclassing this class.

Public Methods

`HandleEval`

`int` **`HttpdSymbolTable::HandleEval`** `(HttpdEvalCommand *`*`p_eval`*`);`

An eval command needs to be executed by the template engine. The command should be analyzed by this method and handled if appropriate. If not appropriate, the value HTTPD_TEMPLATE_NOT_HANDLED should be returned.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

HandleLoop

```
int HttpdSymbolTable::HandleLoop (HttpdLoopCommand *p_loop);
```

An loop command needs to be executed by the template engine. The command should be analyzed by this method and handled if appropriate. If not appropriate, the value HTTPD_TEMPLATE_NOT_HANDLED should be returned.

The function `Iterate` method of *`p_loop`* should be called each time the body of the loop should be evaluated. It is also very useful to add a new lexical scope during the iterations for variables that change as the loop progresses.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

HandleCond

```
int HttpdSymbolTable::HandleCond (HttpdConditionalCommand *p_cond);
```

A conditional command needs to be executed by the template engine. There are three possible outcomes for processing a conditional command:

- The condition is true, indicated by returning HTTPD_TEMPLATE_TRUE_CASE
- The condition is false, indicated by returning HTTPD_TEMPLATE_FALSE_CASE
- The operation failed or should not be handled, indicated by returning the appropriate error code.

Note

This method should never return `0`. This is an ambiguous result to the template engine.

ReturnBool

```
int HttpdSymbolTable::ReturnBool (bool value);
```

This helper function maps the *`value`* to the appropriate return value for handling template conditionals.

HttpdPrefixSymbolTable Reference

Introduction

The `HttpdPrefixSymbolTable` class is a small wrapper that adds functionality for named prefixes to the `HttpdSymbolTable` abstract interface.

With many symbol tables active simultaneously it can be difficult to differentiate between them. The `HttpdPrefixSymbolTable` class allows command names to be given an easily recognizable prefix. There is no additional implementation to the command handlers from `HttpdSymbolTable`.

The registered prefix can then be used to address all commands. For example:

```
%{eval:buffer-show}%
```

would match an object with a prefix of `buffer` and the string `show` would be returned from the `Command` method.

Public Methods

HttpdPrefixSymbolTable

`**HttpdPrefixSymbolTable::HttpdPrefixSymbolTable** (const char *p_prefix);`

This initializes a `HttpdPrefixSymbolTable` object. The lifetime of the *`p_prefix`* string must be equal to or exceed the lifetime of this class as it is not copied internally.

Prefix

`const char * **HttpdPrefixSymbolTable::Prefix** (void);`

This method returns the prefix that was used to initialize the object.

Command

`const char * **HttpdPrefixSymbolTable::Command** (const HttpdTemplateCommand *p_command);`

Given a command object this function determines if it matches the prefix of this object. If so the remaining portion of the command object (following the prefix) is returned. Otherwise, NULL is returned and no further processing should be performed.

HttpdTemplateCommand Reference

Introduction

`HttpdTemplateCommand` serves as the base class for all of the command classes:

- `HttpdEvalCommand`
- `HttpdLoopCommand`
- `HttpdConditionalCommand`

The public methods in this class are available from any command and should be called in one of the handler methods of `HttpdSymbolTable` derivatives.

Public Methods

Name

`const char * **HttpdTemplateCommand::Name** (void);`

Returns the name of the of the command. For example, in a template directive such as:

```
  %{eval:user_name_string}%
```

the returned value would be the string `user_name_string`.

Note

This method will never fail or return NULL by the time the command is passed to a a handler method. Therefore it is safe for callers to always assume a valid name.

Attribute

`const char * HttpdTemplateCommand::Attribute (const char *p_name);`

If the specified attribute of the command exists, its value is returned. Otherwise NULL is returned.

Attributes

`HttpdCgiParameter * HttpdTemplateCommand::Attributes (void);`

Returns a list of the parsed attributes. For example, in a template directive such as:

```
%{eval:user_name_string  class = "logged-in"
                         mode  = local
                         id    = 65 }%
```

the parameters get encoded by SCPG when the template is compiled. This method reads the encoded attributes (via the `AttributeString` method) and parses them out into a `HttpdCgiParameter` list.

If no attributes exist or there is an error loading the attributes the value NULL is returned.

Caution

The returned list must *not* be released by the caller. It is owned by the command object and will be released when command processing is completed. Do not keep pointers to the nodes of the list or the strings contained within them; make a copy if necessary.

Output

`HttpdWritable * HttpdTemplateCommand::Output (void);`

Returns the associated output object that is the results of the template. This is commonly needed when processing eval commands. For example:

```
int MySymbolTable::HandleEval(HttpdEvalCommand *p_eval)
{
  if (strcmp(p_eval->Name(), "user_name") == 0)
    return (p_eval->Output()->Printf("user%d", userId));
  else
    return (HTTPD_TEMPLATE_NOT_HANDLED);
}
```

Note

There is always an output stream; therefore this method will never return NULL.

Processor

`HttpdTemplateProcessor * **HttpdTemplateCommand::Processor** (void);`

This method returns a pointer to the associated template processor object.

HttpdEvalCommand Reference

Introduction

An `HttpdEvalCommand` represents an evaluation command in the template. It is derived from `HttpdTemplateCommand` and possesses its public interface. See Template Command Objects.

Note

This class is never instantiated in application code. It is created during template execution and passed to the various `HandleEval` methods of the symbol tables.

Public Methods

Format

`int **HttpdEvalCommand::Format** (const char *p_string);`

There are many different rules for escaping strings when dealing with HTTP and HTML. This helper routine will format a string with support for some basic attributes that help deal with the quoting issues. The following attributes are supported:

- The `quote` attribute will perform quoting in the specified order using one of the following tokens:

`html`	Characters that are HTML tokens such as & or < are escaped using the `HttpdUtilities::HtmlQuote` routine.
`uri`	The string is encoded using the `HttpdUtilities::UriEncode` routine.
`unuri`	The string is decoded using the `HttpdUtilities::UriDecode` routine.
`unuri+`	The string is decoded using the `HttpdUtilities::UriDecode` routine. With the *`plus_xlat`* parameter set to `true`.
`c-ascii`	The string is encoded using the section called “`CQuoteString`” with the `STR_QUOTE_C` mode. Enclosing quotation marks are not automatically appended.
`js-utf8`	The string is encoded using the section called “`CQuoteString`” with the `STR_QUOTE_JSON` mode. Enclosing quotation marks are not automatically appended. This mode is especially useful for placing strings within JavaScript functions or encoding data in JSON format.

- The `remove-chars` attribute causes any characters in its value to be removed from the formatted string.
- The `filter-chars` attribute causes any characters not in its value to be removed from the formatted string.
- The `trim-front` attribute causes leading whitespace to be removed.
- The `trim-rear` attribute causes trailing whitespace to be removed.
- The `trunc` attribute limits the maximum number of output characters.

For example to HTML-quote and remove all trailing and leading whitespaces for a table field limiting the output to a maximum of 32 characters use the following template directive:

```
<td>%{eval:symbol quote="html"
                  trunc="32"
                  trim-front trim-rear}%</td>
```

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

FormatInteger

`int` **`HttpdEvalCommand::FormatInteger`** `(long` *`value`*`);`

`int` **`HttpdEvalCommand::FormatUnsigned`** `(unsigned long` *`value`*`);`

This function performs flexible formatting of *`value`*. Most of the standard mechanisms of the `printf()` family of functions can be employed with the appropriate attributes. In addition to the common attributes the following type-specific attributes may be used:

`hex`	The converted value is output in hexadecimal using lower-case alphabetic characters.
`HEX`	The converted value is output in hexadecimal using upper-case alphabetic characters.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

FormatFloat

`int` **`HttpdEvalCommand::FormatFloat`** `(double` *`value`*`);`

This function performs flexible formatting of *`value`*. Most of the standard mechanisms of the `printf()` family of functions can be employed with the appropriate attributes. In addition to the common attributes the following type-specific attributes may be used:

`format` = *`X`*	This specifies the formatting style (as per `printf()`) where *`X`* is one of `f`, `g`, `G`, `e`, or `E`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This method is only available if the target porting layer defines the HAVE_FLOATING_POINT preprocessor symbol to a non-zero value.

Common Formatting Attributes

Attribute	Effect
`width` = *`value`*	Sets the minimum field width (in characters). If the converted value has fewer characters than the field width, it will be padded according to the other formatting attributes.
`prec` = *`value`*	An optional precision. If this attribute is omitted, the precision is taken as zero. This value gives the minimum number of digits to appear for integral conversions, the number of digits to appear after the decimal point for scientific notation, or the maximum number of significant digits for floating point values.
`alt`	If this attribute is present then an “alternative” form is used for the value. This is identical to using the “#” modifier with the `printf()` functions. For hexadecimal output this results in a leading `0x` prefix. For floating point conversions typically a decimal point is always printed.
`zero`	If this attribute is present then zero padding rather than space padding is used: The converted value is padded with zeros rather than blanks.
`left`	The presence of this attribute indicates the converted value is to be left justified: The converted value is padded on the right instead of the left.
`blank`	This attribute specifies that a blank should be left before a positive number. This attribute is analogous to a space in a `printf()` style format string.
`plus`	This attribute causes a sign to always be placed before a the produced number.

`HttpdLoopCommand` Reference

Introduction

An `HttpdLoopCommand` represents a loop command in the template. It is derived from `HttpdTemplateCommand` and possesses its public interface. See Template Command Objects. In addition, this class has additional methods that handlers can use during the processing of this command.

Note

This class is never instantiated in application code. It is created during template execution and passed to the various `HandleLoop` methods of the symbol tables.

Public Methods

`Iterate`

`int HttpdLoopCommand::Iterate (void);`

This method causes the body of the loop to be evaluated. It may be called as many times as the loop body needs to be executed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If the return value of this method is not `0` then the return value should be returned without modification from the handler method.

`Counter`

`unsigned long HttpdLoopCommand::Counter (void); const`

During the execution of the loop the `HttpdLoopCommand` object keeps a counter of the number of iterations executed. The counter starts at `0` and is incremented by one for each iteration. This method returns the counter value.

`HttpdConditionalCommand` Reference

Introduction

An `HttpdConditionalCommand` represents an conditional command in the template. It is derived from `HttpdTemplateCommand` and possesses its public interface. See Template Command Objects.

Note

This class is never instantiated in application code. It is created during template execution and passed to the various `HandleCond` methods of the symbol tables.

Public Methods

`Test` (String Version)

`int HttpdConditionalCommand::Test (const char *p_command, const char *p_string);`

This method implements a series of standard tests on *`p_string`* depending on the value of *`p_command`*. The following tests are supported:

- The `empty` condition is true if *`p_string`* is empty.
- The `blank` condition is true if there are no non-whitespace characters in *`p_string`*.
- The `length` condition is used for comparing the length of *`p_string`* to the value in the attribute `len`. The attribute `is` can be one of `<` (less than), `>` (greater than), or `=` equal to.
- For numbers represented as strings the `number` symbol can be used to perform rudimentary comparisons. As with the `length` a relational operator is specified in the `is` attribute. A `value`

attribute holds the value being compared against and the `base` attribute holds the base of the numbers being compared. If a base of `0` is specified then the base of the numbers is determined using the standard C syntax for numbers.

- For more complex string matching the `match` symbol can be used. A `pattern` attribute is used to apply the string to the HttpdUtilities::MatchPattern method. If the attribute `not` is present then the test is considered true if the pattern does not match. Otherwise the test is considered true if the pattern matches the string.

The return value of this function should be returned from the `HandleCond` method.

`Test` (Integer version)

`int` **`HttpdConditionalCommand::Test`** `(const char *`*`p_command`*`, long `*`value`*`);`

This method implements a series of standard tests on *`value`* depending on the value of *`p_command`*. An attribute with a name of `to` gives the value to compare against. The following tests are supported:

- The `lt` condition is true if *`value`* is less than the value of the `to` attribute.
- The `le` condition is true if *`value`* is less than or equal to the value of the `to` attribute.
- The `eq` condition is true if *`value`* is equal to the value of the `to` attribute.
- The `ne` condition is true if *`value`* is not equal to the value of the `to` attribute.
- The `gt` condition is true if *`value`* is greater than the value of the `to` attribute.
- The `ge` condition is true if *`value`* is greater than or equal to the value of the `to` attribute.
- The `div` condition is true if *`value`* is divisable by the value of the `to` attribute. If the `rem` attribute is provided then this condition is true if the remainder is the value of that attribute.

The return value of this function should be returned from the `HandleCond` method.

`Test` (Unsigned version)

`int` **`HttpdConditionalCommand::Test`** `(const char *`*`p_command`*`, unsigned long `*`value`*`);`

This method implements a series of standard tests on *`value`* depending on the value of *`p_command`*. All of the test of the signed version (see above) are supported as well as an additional test of `bits`. This test requires an additional attribute, `mask`. The bits that are set in `mask` are compared with the value of the `to` attribute for equality. If equal then the condition is considered true.

The return value of this function should be returned from the `HandleCond` method.

`Test` (Floating-point version)

`int` **`HttpdConditionalCommand::Test`** `(const char *`*`p_command`*`, double `*`value`*`);`

This method implements a series of standard tests on *`value`* depending on the value of *`p_command`*. The following tests are supported:

- The `whole` condition is true if *`value`* is a whole number.

- The `lt` condition is true if *`value`* is less than the value of the `to` attribute.
- The `le` condition is true if *`value`* is less than or equal to the value of the `to` attribute.
- The `eq` condition is true if *`value`* is equal to the value of the `to` attribute. Because comparing floating-point values for equality is not well defined an additional attribute, `prec`, may set the epsilon value that defines the tolerance of equality.
- The `ne` condition is true if *`value`* is not equal to the value of the `to` attribute. An optional attribute, `prec`, may set the epsilon value that defines the tolerance of inequality.
- The `gt` condition is true if *`value`* is greater than the value of the `to` attribute.
- The `ge` condition is true if *`value`* is greater than or equal to the value of the `to` attribute.

The return value of this function should be returned from the `HandleCond` method.

Note

This method is only available if the target porting layer defines the HAVE_FLOATING_POINT preprocessor symbol to a non-zero value.

`HttpdTemplateScope` Reference

Introduction

`HttpdTemplateScope` is a very interesting class. It has no methods beyond its constructor and destructor. Its purpose is to temporarily add a lexical scope to the current template processor. The constructor inserts the new scope and the destructor removes it.

This behavior means that the lexical scope of the templates (roughly) follows the lexical scope of the C++ handler code. The most common use for this class is to temporarily add additional symbols during the evaluation of a loop. A typical construct might be:

```
int MySymbolTable::HandleLoop(HttpdLoopCommand *p_loop)
{
  if (strcmp(p_loop->Name(), "session_table") == 0)
  {
    AppLoopVariables   vars;
    HttpdTemplateScope loop_scope(p_loop->Processor(), &vars);

    while (!vars.Done())
    {
      int rc = p_loop->Iterate();
      if (rc != 0)
        return (rc);
    }
  }
  else
    return (HTTPD_TEMPLATE_NOT_HANDLED);
}
```

In the above example the scope becomes active when the if body is entered. The scope is removed when the block is exited. If another loop were to be executed during the call to `Iterate` (even the same loop again) then additional scopes can be entered. The caret (`^`) can be used to access obscured variables in previous scopes.

Public Methods

`HttpdTemplateScope`

`HttpdTemplateScope::HttpdTemplateScope` `(HttpdTemplateProcessor *`*`p_proc`*`, HttpdSymbolTable *`*`p_symbols`*`);`

The constructor of this class opens a new lexical scope in the chain of symbol tables in the template processor. The first parameter, *`p_proc`* is typically obtained by calling the `Processor` method of a command object. The second parameter, *`p_symbols`* is a pointer to the symbol table that is to be temporarily installed. The symbol table must be unique per scope and is typically instantiated as a local variable along with this object.

`HttpdTemplateProcessor` Reference

Introduction

`HttpdTemplateProcessor` is the engine that reads a template file, parses the contents, instantiates the command objects and calls the various handler methods of the various symbol tables.

Of course, the most important function `HttpdTemplateProcessor` must do is to keep track of all the things that are internal to the template processor. Thus, a pointer to this object is almost always required (explicitly or implicitly through another object) when performing any template operations.

There should be one instance of `HttpdTemplateProcessor` for each template file being processed. In the case of include files a mechanism is provided for cloning a new processor object from an existing one.

In the case of a newly created (non-cloned) template processor the symbol scope is empty and (at a minimum) a top-level symbol table must be installed before processing a template file. The top-level symbol table must never return the value `HTTPD_TEMPLATE_NOT_HANDLED` from the handler methods. Additionally, more scopes (in addition to the top scope) may be installed before starting template processing, if necessary.

`HttpdTemplateProcessor` Internals

The template processor uses what is often called the “visitor pattern”. The compiled template file is divided into blocks. Each block begins with a byte called the “op code”. The value of the op code determines the remaining format of the block.

When a template is to be executed, the processor starts processing the blocks in the file sequentially starting at the beginning of the file. The op code is analyzed and processed. When a block that represents an encoded command is to be processed a “visitor” object is constructed and is then asked to process the remainder of the block.

The visitor objects are used to encapsulate the transient information for processing a command. The visitor objects are the objects that are passed to the handler methods of the symbol tables. There is a unique handler method for each type of command object so there is little extra overhead at runtime for dispatching different commands to the same “name”.

Public Methods

`HttpdTemplateProcessor` (Clone constructor)

`HttpdTemplateProcessor::HttpdTemplateProcessor` `(HttpdTemplateProcessor` `*`*`p_clone`*`, bool` *`isolate`*`);`

This constructor initializes a template processor using an already existing object (*`p_clone`*) as a base. The constructed object can process a new template file but can use a related scope.

If *`isolate`* is false then the newly constructed object has the exact same lexical scope available to it as *`p_clone`*. If *`isolate`* is true then the new object only shares the top-level (first installed) symbol table with its predecessor. In either case, the newly created object does not need a top-level symbol table installed.

`StartProcessing`

`int` **`HttpdTemplateProcessor::StartProcessing`** `(HttpdFile *`*`p_input`*`, HttpdWritable *`*`p_output`*`);`

This method invokes the processing of the input file specified by *`p_input`*. The output is sent to *`p_output`*.

If the object was not constructed with the clone constructor then a top-level symbol table must be inserted (with the help of the `HttpdTemplateScope` class) before `StartProcessing` can be called.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Note

During the execution of this method (such as in the symbol tables), the `HttpdTemplateProcessor` object may be casted to a HttpdWritable * type to obtain the output pointer.

`Top`

`HttpdSymbolTable *` **`HttpdTemplateProcessor::Top`** `(void);`

This method returns the top-level symbol table, regardless of lexical scope. Often the first lexical scope contains important state information that may be useful to other symbol tables.

Caution

This method should only be called after at least one lexical scope level has been established.

`HttpdFSTemplateShell` Reference

Introduction

`HttpdFSTemplateShell` is a top-level symbol table designed to be used when subclassing `HttpdFileHandler`. This class has several duties:

- Act as an anchor to the `HttpdFileHandler::RequestState` object.
- Handle an evaluation command called `include_file` that can be used for pre-canned headers and footers.
- Catch symbols that undefined by returning HTTPD_TEMPLATE_UNKNOWN_NAME for unknown names.

This class is typically created during the `DoFile` method of `HttpdFileHandler` subclasses. The example `main` provided demonstrates handling template object with `HttpdFSTemplateShell`.

By default the *MIME* type for the expanded content is always `text/html`. This can be overridden if the `INC_TEMPLATE_MIME_TYPES` feature is enabled. When enabled files carrying the `expanded_mime_type` attribute will use the value of that attribute as the *MIME* type.

Public Methods

HttpdFSTemplateShell

```
HttpdFSTemplateShell::HttpdFSTemplateShell
(HttpdFileHandler::RequestState &state);
```

Constructor for `HttpdFSTemplateShell`. A reference to *`state`* is kept inside the shell object. It is therefore important that the lifetime of the shell is a subset of the lifetime of the file request.

Note

After construction the shell can be inserted into a template processor via a `HttpdTemplateScope`. This is rarely necessary, however, because the entire functionality of processing a template is implemented in the `Execute` (static) method of this class.

State

```
HttpdFileHandler::RequestState & HttpdFSTemplateShell::State (void);
```

This method returns the stored reference to the request state. This is really a convenience mechanism so that any handler in the chain can obtain a reference to the per-request state.

TopState

```
HttpdFileHandler::RequestState & HttpdFSTemplateShell::TopState (void);
```

Given a pointer to a template command obtain the request state object. Subclasses of `HttpdFileHandler` can use the `mpData` member to store application-specific data. This helper allows easy access to that pointer from the command object.

Note

This is a static method and does not require an instance of `HttpdFSTemplateShell`.

Caution

This method should only be called after at least one lexical scope level has been established.

`Execute`

`int` **`HttpdFSTemplateShell::Execute`** `(HttpdFileHandler::RequestState &`*`state`*`, HttpdSymbolTable *`*`p_symbols`*`);`

This method handles all of the setup work involved in processing a request from the `DoFile` of `HttpdFileHandler`. It creates a template processor object, sets up the input and output, installs a top-level symbol table, places *`p_symbols`* in the scope list, and finally executes the template.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This is a static method and does not require an instance of `HttpdFSTemplateShell`.

`HttpdFSTemplateRequest` Reference

Introduction

The `HttpdFSTemplateRequest` class is subclass of `HttpdTemplateProcessor`. The purpose of the class is to provide a simple way of using the `HttpdFSTemplateShell` class with handlers subclassed from `HttpdFileHandler`.

When a request for a template comes in to a subclass of `HttpdFileHandler` an instance of this class can be associated with the `HttpdFileHandler::RequestState` object. Once associated symbol table instances may be attached to the `HttpdFSTemplateRequest` as necessary. Finally, the `Execute` method is called to evaluate the associated template.

All of the machinery for opening the template file, creating the shell scope and handling errors is wrapped up inside the methods of this class. If only one symbol table is needed then an even easier approach may be used: Simply call the static `HttpdFSTemplateShell::Execute` method.

Public Methods

`HttpdFSTemplateRequest`

`HttpdFSTemplateRequest::HttpdFSTemplateRequest` `(HttpdFileHandler::RequestState &`*`state`*`);`

Constructor for `HttpdFSTemplateRequest`. A reference to *`state`* is kept inside the request object. It is therefore important that the lifetime of the request is a subset of the lifetime of the file request.

`Execute`

`int` **`HttpdFSTemplateRequest::Execute`** `(void);`

This method actually executes the template. It is assumed that all initial symbol tables have been attached to the `HttpdFSTemplateRequest` object before calling this method.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdConstantSymbolTable` Reference

Introduction

This class introduces a simple way to map string constants into templates. It is useful to avoid the proliferation of many nearly identical template files.

A simple mapping of evaluation name to string constant is specified during object construction via an array of HttpdPair structures.

Public Methods

`HttpdConstantSymbolTable`

`HttpdConstantSymbolTable::HttpdConstantSymbolTable` `(const HttpdPair *`*`p_table`*`, size_t `*`num_pairs`*`);`

Initialize the constant symbol table to use the symbols and values specified by *`p_table`*. No copy of the table is made, therefore the lifetime of the table must meet or exceed the lifetime of the `HttpdConstantSymbolTable` object.

Note

The table specified by *`p_table`* is searched using a binary search algorithm. The table must be in sorted order.

`HttpdSymbolMap` Reference

Introduction

Constructing code to handle fetching of variables for templates that do reporting can become tedious and result in code bloat. `HttpdSymbolMap` acts as a generic symbol table implementation for dealing with C or C++ structures. When constructed it is passed an array of `HttpdSymbolEntry` structures that define the layout of the fields in an application specific structure.

Note

Because a binary search is used on the `HttpdSymbolEntry` array, the elements must be sorted on the name field.

`HttpdSymbolEntry` associates a field name with a byte offset to locate the data item in question and function pointers for processing the commands. Because it uses function pointers to handle the actions it can be extended easily without subclassing. The `HttpdSymbolMap` class contains several static methods that handle most common data types.

Table 5.2. `HttpdSymbolMap` Default Handlers

Method	Template Directive	Data Type	Description
`EvalString`	`eval`	const char *	A pointer to a NUL-terminated string is formatted “as is” excluding the NUL byte.

Method	Template Directive	Data Type	Description
`EvalStringBuffer`	`eval`	const char array	An array of characters (NUL-terminated string)
`EvalUlong`	`eval`	unsigned long	The value is displayed as decimal with as many digits as required.
`EvalHexUlong`	`eval`	unsigned long	The value is displayed as hexadecimal with as many digits as the maximum possible value would take.
`EvalLong`	`eval`	long	The signed value is displayed in decimal with as many digits as required. An optional leading minus indicates a negative value.
`CondBool`	All conditional directives	bool	A C++ bool value is examined and used as the basis for the template condition.

Given the following example object from an application using Seminole:

```
struct UserInfo
{
  UserInfo       *next;
  const char     *name;
  unsigned long   userid;
  long            balance;
  bool            logged_on;
};
```

The following array of `HttpdSymbolEntry` objects describes the above structure:

```
const HttpdSymbolEntry UserInfo_map[] =
{
  {
    "balance",
    offsetof(UserInfo, balance),
    HttpdSymbolMap::EvalLong,      // Evaluation
    NULL,                          // Looping
    NULL                           // Conditional
  },

  {
    "logged_on",
    offsetof(UserInfo, logged_on),
    NULL,                          // Evaluation
```

```
          NULL,                              // Looping
          HttpdSymbolMap::CondBool           // Conditional
        },

        {
          "name",
          offsetof(UserInfo, name),
          HttpdSymbolMap::EvalString,        // Evaluation
          NULL,                              // Looping
          NULL                               // Conditional
        },

        {
          "userid",
          offsetof(UserInfo, userid),
          HttpdSymbolMap::EvalHexUlong,      // Evaluation
          NULL,                              // Looping
          NULL                               // Conditional
        }
      };
```

Using the above symbol map we can design a template for displaying user records in an HTML table:

```
      <table>
       <th>
        <td>Name</td>
        <td>Balance</td>
        <td>User ID</td>
        <td>Logged On</td>
       </th>
       %{loop:user_table}%
        <tr>
         <td>%{eval:name}%</td>
         <td>%{eval:balance}%</td>
         <td>%{eval:userid}%</td>
         <td>%{if:logged_on}%yes%{else}%no%{endif}%</td>
        </tr>
       %{endloop}%
```

Writing the code to then traverse the user_table loop becomes trivial:

```
  int MySymbolTable::HandleLoop(HttpdLoopCommand *p_loop)
  {
    if (strcmp(p_loop->Name(), "user_table") == 0)
    {
      UserInfo *ui;

      for(ui = userList; ui->next != NULL; ui = ui->next)
      {
        HttpdSymbolMap     map(UserInfo_map,
                               HTTPD_NUMELEM(UserInfo_map),
                               (const void *)ui);
        HttpdTemplateScope loop_scope(p_loop->Processor(), &map);

        int rc = p_loop->Iterate();
        if (rc != 0)
          return (rc);
      }
    }
    else
      return (HTTPD_TEMPLATE_NOT_HANDLED);
  }
```

Note

Because the usage of `HttpdSymbolMap` and `HttpdTemplateScope` together is so common, the above example can be simplified with the `HttpdScopedSymbolMap` that acts as an automatically scoped `HttpdSymbolMap`.

Public Methods

`HttpdSymbolMap`

`HttpdSymbolMap::HttpdSymbolMap` `(const HttpdSymbolEntry *`*`p_table`*`, size_t` *`num_elem`*`, const void *`*`p_base`*`, int` *`not_found`* `= HTTPD_TEMPLATE_NOT_HANDLED);`

The constructor of this class initializes the symbol table from the sorted table of elements (*`p_table`*). The number of elements in the table should be passed as *`num_elem`*. The parameter *`p_base`* should point to the object associated with the symbols.

The optional *`not_found`* argument is the result code to return if the symbol is not found. Ordinarily the default of `HTTPD_TEMPLATE_NOT_HANDLED` is appropriate. However, if the symbol map is the terminal symbol table then `HTTPD_TEMPLATE_UNKNOWN_NAME` should be used to prevent further searching.

`HttpdScopedSymbolMap` Reference

Introduction

`HttpdScopedSymbolMap` is wrapper class for `HttpdSymbolMap` that automatically inserts and removes its self from the lexical scope of an instance of `HttpdTemplateProcessor`.

This class makes certain constructs, such as looping over a list of structures, much easier. The code in the HttpdSymbolMap example can be simplified down to:

```
int MySymbolTable::HandleLoop(HttpdLoopCommand *p_loop)
{
  if (strcmp(p_loop->Name(), "user_table") == 0)
  {
    UserInfo *ui;
    for(ui = userList; ui->next != NULL; ui = ui->next)
    {
      HttpdScopedSymbolMap     map(p_loop->Processor(),
                                   UserInfo_map,
                                   HTTPD_NUMELEM(UserInfo_map),
                                   (const void *)ui);

      int rc = p_loop->Iterate();
      if (rc != 0)
        return (rc);
    }
  }
  else
    return (HTTPD_TEMPLATE_NOT_HANDLED);
```

Public Methods

`HttpdScopedSymbolMap`

`HttpdScopedSymbolMap::HttpdScopedSymbolMap` `(HttpdTemplateProcessor *`*`p_processor`*`, const HttpdSymbolEntry *`*`p_table`*`, size_t `*`num_elem`*`, const void *`*`p_base`*`);`

The constructor of this class initializes the symbol table from the table defined by *`p_table`* and *`num_elem`*. The new symbol table is inserted in the top of the scope list of *`p_processor`*. The object is removed from the scope list when it is destroyed.

CGI-template Interfacing

Introduction

There may be cases when repeated HTML forms must be generated for "wizard" like interfaces. In these situations it may be helpful to propagate CGI parameters into a template. There are three classes with slightly different interfaces that can be used to accomplish this goal.

- `HttpdCgiSymbols` provides generic template support for any kind of name-value pair storage.
- `HttpdCgiListSymbols` provides template support for a singly-linked list of `HttpdCgiParameter` objects. These lists are returned from the CGI parser routines. Because the lists are ordered this class also supports looping directives.
- `HttpdCgiHashSymbols` provides template support for a `HttpdCgiHash` object. Unlike linked-lists, hash objects have quicker lookup performance. However, they are not ordered in any meaningful way so no looping directives are defined.

All of these classes are derived from `HttpdPrefixSymbolTable` so that multiple instances can be active simultaneously with different lists and no ambiguity.

For all of these commands the parameter name is specified with the `name` attribute. If no `name` attribute is specified then the current iteration (if any) position is used.

- The `val` symbol evaluates to the contents of the specified parameter. If the `default` attribute is specified and no parameter exists for that name then the its value is used instead.

- The `current` symbol is the current iteration point (if any).
- The `exists` condition is true if the specified variable exists.
- The `for-each` loop is only available when using the `HttpdCgiListSymbols` class. It iterates through each `HttpdCgiParameter` node in order.

Note

Formatting and conditionals are done using the `HttpdEvalCommand::Format` and `HttpdConditionalCommand::Test` methods. Any behavior those methods support are supported by these classes.

Public Methods

`HttpdCgiSymbols`

`HttpdCgiSymbols::HttpdCgiSymbols` `(const char *`*`p_prefix`*`);`

This constructs the basic CGI interface class for templates. The class responds to commands with the specified prefix. This class is an abstract base class and can not be instantiated directly. Instead, it must be subclassed and the `Find` method implemented.

`HttpdCgiListSymbols`

`HttpdCgiListSymbols::HttpdCgiListSymbols` `(HttpdCgiParameter *`*`p_list`*`, const char *`*`p_prefix`*` = "cgi");`

This constructs the CGI interface class that uses a linked-list of `HttpdCgiParameter` nodes pointed to by *`p_list`*.

`HttpdCgiHashSymbols`

`HttpdCgiHashSymbols::HttpdCgiHashSymbols` `(HttpdCgiHash &`*`hash`*`, const char *`*`p_prefix`*` = "cgi");`

This constructs the CGI interface class that uses a hash table implemented by `HttpdCgiHash`.

Protected Methods

`Find`

`HttpdCgiParameter *`**`HttpdCgiSymbols::Find`** `(const char *`*`p_key`*`);`

This method should return a pointer to a `HttpdCgiParameter` object where the `mpKey` member is equal to the string in *`p_key`*. If no such parameter exists then this routine may return NULL.

`HttpdLoopCounterSymbols` Reference

Introduction

This class is a simple helper that can be used to expose the loop counter of a `HttpdLoopCommand` object to templates. The principle use of this class is for numbering rows of a table or highlighting alternating entries of a list.

The typical usage for this class is to allocate it on the stack during the handling of a loop command. This class automatically stacks itself on the symbol scope of the associated loop command. For example:

```
int SomeSymbolTable::HandleLoop(HttpdLoopCommand *p_loop)
{
  HttpdLoopCounterSymbols lcs(p_loop);

  …
}
```

Public Methods

HttpdLoopCounterSymbols

HttpdLoopCounterSymbols::HttpdLoopCounterSymbols (const HttpdLoopCommand *_p_loop_, const char *_p_prefix_ = mDefaultPrefix);

This method constructs the loop-counter symbol table and installs it in the scope of the *p_loop* command. For the duration of this object (which must be less than the lifetime of the HttpdLoopCommand object) the loop counter may be accessed within the loop body.

The *p_prefix* function specifies the prefix used for accessing the loop counter. The default value is loop-counter. Evaluating this symbol results in formatting the numerical value of the loop counter. If the attribute bias is present then this value is added to the counter before formatting. See the integer formatter reference for details.

The loop counter can also be tested by using template conditionals on the prefix string followed by a relational test as specified in the unsigned value conditionals section.

For example to number a list of entries in a table (starting at 1) and having every other row use an alternate background color consider the following fragment:

```
%{loop:table-objects}%
 <tr
  %{if:loop-counter-div to=2}%
   bgcolor="#555555"
  %{else}%
   bgcolor="#bbbbbb"
  %{endif}%
 >
  <td>%{eval:loop-counter bias=1}%</td>
  <td>%{eval:some-symbol}%</td>
 </tr>
%{endloop}%
```

Public Data

The HttpdLoopCounterSymbols defines a static data member called mDefault with the following definition:

```
static const char mDefault[] = "loop-counter";
```

This variable is used as the default command prefix for the symbols of the `HttpdLoopCounterSymbols` symbol table.

Chapter 6. Processing XML

“Streamy” Processing of XML

Seminole includes a set of classes to assist in parsing XML documents. Unlike some parsers `HttpdXmlParser` is not a validating parser. The parser also operates in a “streamy” fashion. This means that the document can be pumped piecemeal into the parser as it arrives. The `HttpdXmlParser` is derived from HttpdFifo. This allows `POST` requests to easily drive the parser with `HttpdReceiver` or `HttpdBoundaryReader` objects.

There are two different models of processing XML. The first model, implemented by `HttpdXmlParser`, calls various overridable methods of the parser as syntax is recognized. The second model, implemented by `HttpdXmlDomBuilder`, converts the entire document into an in-memory structure.

In memory constrained environments or when dealing with very large documents the callback-driven approach is preferrable as there is no intermediate representation stored in memory. If the document can not be processed until the entire document is parsed then the latter approach makes more sense. In fact it is only natural that `HttpdXmlDomBuilder` is implemented using `HttpdXmlParser` as its base class. The events trigger the construction of the tree structure.

The XML framework is defined in a header file called `sem_xml.h`. In order to use any of these classes or methods, this header file must be included.

`HttpdXmlAttribute` Reference

Introduction

This class represents an attribute on `HttpdXmlNode` and `HttpdXmlDomNode` objects.

Public Methods

`FreeList`

`void` **`HttpdXmlAttribute::FreeList`** `(HttpdXmlAttribute *`*`p_list`*`);`

Destroys a `HttpdXmlAttribute` list, and frees its resources.

`Find`

`HttpdXmlAttribute` **`*HttpdXmlAttribute::Find`** `(HttpdXmlAttribute *`*`p_list`*`, const char *`*`p_name`*`);`

Find the named attribute the specified node forward. Typically this method is called from the first node in the list but it can be used to walk a list with multiple parameters of the same name.

On success this method returns a pointer to the found node. NULL is returned on error.

`FindValue`

`const char *`**`HttpdXmlAttribute::FindValue`** `(const HttpdXmlAttribute *`*`p_list`*`, const char *`*`p_name`*`);`

Find the named parameter from the current node forward.

On success this method returns the value of the found node. NULL is returned if the node can not be found.

`FindValue` (Namespace version)

`const char *`**`HttpdXmlAttribute::FindValue`**` (const HttpdCgiParameter *`*`p_list`*`, const char *`*`p_name`*`, const char *`*`p_namespace`*`);`

Find the the parameter named *`p_name`* that belongs to the namespace *`p_namespace`*.

On success this method returns the value of the found node. NULL is returned if no node satisfies the search criteria.

Note

This method is only present if the INC_XML_NAMESPACES feature is enabled.

`CopyList`

`int `**`HttpdXmlAttribute::CopyList`**` (HttpdXmlHost &`*`host`*`, HttpdXmlAttribute *&`*`p_result`*`, const HttpdXmlAttribute *`*`p_src`*`);`

This static method copies all of the nodes pointed to by *`p_src`* into a new list. The pointer to the first node in the list is placed into *`p_result`* if successful.

The *`host`* object provided should be the same one used for the XML document that will be holding this attribute list.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Public Data

`mpNext`

`HttpdXmlAttribute *mPair`;

This member points to the next attribute if any or NULL if there are no attributes.

`mpName`

`char *mpName`;

This is the name of the attribute.

`mpValue`

`char *mpValue`;

This is the value of the attribute.

`mpNamespace`

`const char *mpNamespace`;

This member is the namespace of the attribute. This member is only present if INC_XML_NAMESPACES is enabled.

`mpSelector`

```
char *mpSelector;
```

This member is the name of the selector used to give this attribute its namespace. It may be NULL if there is no selector used. This member is only present if INC_XML_NAMESPACES is enabled.

`HttpdXmlHost` Reference

Introduction

The `HttpdXmlHost` class contains infrastructure needed to manage the lifetime of an XML document. It is important that the lifetime of the `HttpdXmlHost` meets or exceeds the lifetime of any XML data structures.

A single `HttpdXmlHost` object may be shared between multiple XML data structures. In fact this is more efficient in terms of memory usage.

`HttpdXmlTokenizer` Reference

Introduction

The `HttpdXmlTokenizer` class is used for tokenizing XML style documents. This pure abstract class is derived from `HttpdFifo` and calls various methods when tokens are written to the FIFO. This class is typically not used directly. Rather it serves as a base for the `HttpdXmlParser` class.

Public Methods

`HttpdXmlTokenizer`

`HttpdXmlTokenizer::HttpdXmlTokenizer` `(size_t` *`initial_buffer_size`* `= 0, size_t` *`max_buffer_size`* `=` *`infinity`*`);`

This method constructs the tokenizer. The *`initial_buffer_size`* and *`max_buffer_size`* arguments control the size of the `HttpdFifo` buffer.

`Finish`

`int` **`HttpdXmlTokenizer::Finish`** `(void);`

This method should be called when no more data is written to the tokenizer. It validates that all of the written data that has been digested.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

Protected Methods

TranslateEntity

virtual int **HttpdXmlTokenizer::TranslateEntity** (const char **p_entity*, HttpdWritable **p_target*);

This method is called to process entity references. The value of the entity named in *p_entity* should be written to *p_target*.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

StartText

virtual int **HttpdXmlTokenizer::StartText** (void);

This method is called when non-tag content is encountered. It should set the protected data membermpTarget to a writable object that will receive the content. The default implementation selects the null sink as the target.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

FinishText

int **HttpdXmlTokenizer::FinishText** (void);

This method is called at the end of non-tag content. It may clean up any action done by StartText. The default implementation does nothing.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

BeginDoctype

virtual int **HttpdXmlTokenizer::BeginDoctype** (void);

This method is called when the special <!DOCTYPE tag is opened. Subclasses must ensure they call the default implementation at some point to keep the tokenizer state consistent.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

EndDoctype

virtual int **HttpdXmlTokenizer::EndDoctype** (void);

This method is called when the special <!DOCTYPE tag is closed. Subclasses must ensure they call the default implementation at some point to keep the tokenizer state consistent.

Upon success, 0 is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

ParameterEntity

`virtual int` **`HttpdXmlTokenizer::ParameterEntity`** `(const char *`*`p_entity`*`);`

This method is called when a parameter entity reference (i.e. `%entity;`) is tokenizer.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Token

`virtual int` **`HttpdXmlTokenizer::Token`** `(char` *`ch`*`);`

This pure virtual method is called for various single character separator tokens. At a minimum `<`, `>`, `=`, `/`, and in some cases `?`, `[` and `]` tokens are detected.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

String

`virtual int` **`HttpdXmlTokenizer::String`** `(const char *`*`p_string`*`);`

This pure virtual method is called when a quoted string is tokenized.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

TakeQuotedString

`char *`**`HttpdXmlTokenizer::TakeQuotedString`** `(void);`

This method may be called within `String` to obtain a dynamically allocated (via HttpdOpSys::Malloc) copy of the quoted string. Calling this method may in some cases be more efficient than copying the string directly.

Upon success a pointer to the quoted string is returned. Upon failure NULL is returned.

Identifier

`virtual int` **`HttpdXmlTokenizer::Identifier`** `(const char *`*`p_id`*`);`

This pure virtual method is called when an identifier within a tag is tokenized.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Error

`virtual int` **`HttpdXmlTokenizer::Error`** `(int` *`error_type`*`, …);`

This method is called for malformed input. The *`error_type`* parameter determines how many string parameters follow.

Error Type	**Additional Parameters**
`XML_ERR_EXPECTED`	What was tokenized followed by what was expected.
`XML_ERR_UNEXPECTED`	The unexpected item.
`XML_ERR_UNKNOWN_ENTITY`	The undefined entity name.
`XML_ERR_EARLY_EOF`	None.
`XML_ERR_DUP_NS_SELECTOR`	Selector name.
`XML_ERR_UNKNOWN_NS_SELECTOR_ON_ATTR`	Selector name followed by attribute name.
`XML_ERR_EMPTY_NS_SELECTOR`	Attribute name.
`XML_ERR_UNKNOWN_NS_SELECTOR_ON_NODE`	Selector name followed by tag name.
`XML_ERR_RESERVED_SELECTOR`	Selector name followed by namespace.
`XML_ERR_UNCLOSED_TAG`	Tag name

The default implementation returns `HttpdOpSys::ERR_BADPARAM` and ignores the additional parameters. The protected data member `mLineNumber` is the current line number within the document where the error occured.

`HttpdXmlParser` Reference

Introduction

The `HttpdXmlParser` class is used for processing XML documents. It is subclassed (via `HttpdXmlTokenizer`) from `HttpdFifo` and shares the same public interface for receiving data. Only the additional methods are documented here.

Public Methods

`HttpdXmlParser`

`HttpdXmlParser::HttpdXmlParser` `(HttpdXmlHost &`*`host`*`, HttpdUint8` *`flags`* `= 0, size_t` *`initial_buffer_size`* `= 0, size_t` *`max_buffer_size`* `=` *`infinity`*`);`

This method constructs the parser. The *`initial_buffer_size`* and *`max_buffer_size`* arguments control the size of the `HttpdFifo` buffer.

If *`flags`* has the `HttpdXmlParser::XML_OPT_ANONYMOUS_CLOSE` bit set then the SGML close-tag shortcut (`</>`) optimization is supported.

The *`host`* object is used to manage resources during the parse. Any objects that remain after the parse (such as nodes and attribute lists) refer to memory allocated within this object. As such it is important to ensure that the lifetime of *`host`* is greater than any objects that survive the parsing operation.

The object can not be used until the `Create` method is called first.

`Create`

`int` **`HttpdXmlParser::Create`** `(void);`

This method creates and initializes the parser.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Finish

`int` **`HttpdXmlParser::Finish`** `(void);`

This method should be called after the entire document has been written to the parser. It validates that the parse has digested all data and that all of the state machines are in their appropriate idle states.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

Protected Methods

ProcessingInstruction

`virtual int` **`HttpdXmlParser::ProcessingInstruction`** `(const char *`*`p_instruction`*`, const char *`*`p_attribute`*`, const char *`*`p_value`*`);`

This method is called when a processing instruction directive is encountered. For example the following processing directive:

```
<?xml version="1.0" standalone='yes'?>
```

Would result in two distinct calls to this method. The first call would have the following parameters:

Parameter	**Value**
p_instruction	xml
p_attribute	version
p_value	1.0

The second call would be as follows:

Parameter	**Value**
p_instruction	xml
p_attribute	standalone
p_value	yes

The default behavior of this method is to simply return `0` (success).

RootBody

`virtual int` **`HttpdXmlParser::RootBody`** `(HttpdWritable *&`*`p_target`*`);`

This method is called when textual content is present for the root of the document. Under normal circumstances this is considered invalid XML However to allow XML fragments to be parsed this method may optionally place the address of a writable object in *`p_target`*.

The default behavior of this method is to set *p_target* to `HttpdNullSink::Null()` and return `0` (success).

CloseRootBody

`virtual int` **`HttpdXmlParser::CloseRootBody`** `(void);`

This method is called to complete the processing of textual content at the root of the document. Subclasses may use it to complete any actions performed in `RootBody`.

The default behavior of this method is simply to return `0` (success).

AllocateNode

`virtual int` **`HttpdXmlParser::AllocateNode`** `(const char *`*`p_tag`*`, HttpdXmlNode *&`*`p_node`*`);`

This method is called to allocate a new node when an opening tag construct is seen. The default implementation allocates an instance of `HttpdXmlNode`. Subclasses may override this method to return subclasses of `HttpdXmlNode` with specialized behavior. This is how "event driven" parsing works. Subclasses of `HttpdXmlNode` are defined to handle each particular state. This factory method returns the appropriate object depending on the tags seen. These subclasses then handle the parse of the entire tag as necessary.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

InnermostNode

`HttpdXmlNode *`**`HttpdXmlParser::InnermostNode`** `(void);`

This method returns the innermost node currently being parsed. This method may be called during `AllocateNode` to get additional context if needed. The return value is never NULL. Additionally it is invalid to call this method during `RootBody` and `CloseRootBody`.

IsPath

`bool` **`HttpdXmlParser::IsPath`** `(const char *`*`p_path`*`);`

This method tests if the current state of the document matches *p_path*. Just like a filesystem path the components of the path are separated with `/`. The path can either be relative or absolute if it begins with a leading `/`.

Just like the `InnermostNode` method this method is intended to be called primarily from implementations of `AllocateNode`. Consider the following document:

```
<a>
 <b>
  <c>
   <d>foo</d>
  </c>
 </b>
</a>
```

If we are constructing node `d` then the absolute path would be `/a/b/c`. A matching relative path would be `b/c`.

If the path matches the current point of the parse true is returned. Otherwise false is returned.

`HttpdXmlNode` Reference

Introduction

The `HttpdXmlNode` class represents a tag, its attributes, and content. Instances of this class are created by `HttpdXmlParser::AllocateNode`.

Once the parser creates the node it calls various methods to process the content of the tag. The default implementation of `HttpdXmlNode` records attributes and throws away the contents of the tag. However this behavior can be altered via subclassing.

Public Methods

`HttpdXmlNode`

`HttpdXmlNode::HttpdXmlNode` `(const char *`*`p_tag`*`, int &`*`rc`*`);`

This constructs the node and saves a copy of *`p_tag`* internally. Upon success *`rc`* is set to `0`; otherwise *`rc`* is set to a system dependent error value (see Table 4.1, "OS Abstraction Layer Error Codes").

`Tag`

`const char *`**`HttpdXmlNode::Tag`** `(void);`

This method returns a pointer to the tag name of this node.

Note

The return value can never be NULL.

`Namespace`

`const char *`**`HttpdXmlNode::Namespace`** `(void);`

This method returns a pointer to the namespace of this node. It never returns NULL.

Note

This method is only available if the INC_XML_NAMESPACES feature is enabled.

`Selector`

`const char *`**`HttpdXmlNode::Selector`** `(void);`

This method returns a pointer to the selector used to assign the namespace of this node. If no selector was used this method returns NULL.

Note

This method is only available if the INC_XML_NAMESPACES feature is enabled.

Protected Methods

`BodySink`

`virtual int` **`HttpdXmlNode::BodySink`** `(HttpdXmlParser *`*`p_parser`*`, HttpdWritable *&`*`p_target`*`);`

This method is called to obtain a writable object to process the body contents of this tag. If it returns success then *`p_target`* should point to an object that can receive the tag contents.

The default behavior of this method is to assign `HttpdNullSink::Null()` to *`p_target`* and return `0` (success).

`CloseBodySink`

`virtual int` **`HttpdXmlNode::CloseBodySink`** `(void);`

This method is called to complete the processing of the textual content of the node. Subclasses may use it to complete any actions performed in `BodySink`.

The default behavior of this method is simply to return `0` (success).

`Attribute` (First Pass)

`virtual int` **`HttpdXmlNode::Attribute`** `(HttpdXmlParser *`*`p_parser`*`, const char *`*`p_name`*`, const char *`*`p_value`*`);`

This method is called for each attribute as it is parsed. The attribute data can be processed however is desired. However if this method returns `HTTPD_ERR_SAVE_ATTRIBUTE` then the attribute is converted into a `HttpdXmlAttribute` object and passed to the `AttributesComplete` method that takes the list of saved attributes. Returning this value is possibly more efficient than constructing the `HttpdXmlAttribute` directly in this method.

The default implementation of this method simply returns `HTTPD_ERR_SAVE_ATTRIBUTE` in all cases.

If `0` is returned the attribute is considered processed and no `HttpdCgiParameter` object is constructed; otherwise a system dependent error value may be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

Even if the INC_XML_NAMESPACES feature is enabled there is no namespace information available when this method is called. If the namespace of an attribute is important the default implementation of this method should be used and processing should be done in the `AttributesComplete` method.

`AttributesComplete`

`virtual int` **`HttpdXmlNode::AttributesComplete`** `(HttpdXmlParser *`*`p_parser`*`, HttpdXmlAttribute *`*`p_saved_attribs`*`);`

This method is called after all of the attributes for this tag have been processed. After this method is called no further calls to the `Attribute` method will be made.

The default implementation simply returns `0` (success).

`Close`

virtual int **HttpdXmlNode::Close** (HttpdXmlParser **p_parser*);

This method is called after a tag is entirely processed. No further calls will be made to any of the other virtual methods of this object after this method is called. If this method returns the special status code `HTTPD_ERR_DELETE_NODE` the parser will automatically delete this object. If `0` is returned then some other object must eventually delete this object.

The default implementation simply returns `HTTPD_ERR_DELETE_NODE` since by default the nodes are not kept after they are processed.

`HttpdXmlDomBuilder` Reference

Introduction

The `HttpdXmlDomBuilder` class is derived from `HttpdXmlParser` to construct a tree of `HttpdXmlDomNode` objects representative of the document. Unlike `HttpdXmlParser` this class is not intended to be subclassed; rather it provides a simple interface to consume a whole document.

The document is converted to an in-memory tree representing the content of the document. This data structure, called a *DOM* or Document Object Model, is composed of a collection of one or more `HttpdXmlDomNode` objects. The tree can be manipulated and then seralized back to XML with the the section called “`HttpdXmlDomWriter` Reference” class.

Public Methods

`HttpdXmlDomBuilder`

HttpdXmlDomBuilder::HttpdXmlDomBuilder (HttpdXmlHost &*host*, HttpdUint8 *flags* = 0, size_t *initial_buffer_size* = 0, size_t *max_buffer_size* = *infinity*);

This method constructs the parser. The *`initial_buffer_size`* and *`max_buffer_size`* arguments control the size of the `HttpdFifo` buffer.

The *`flags`* argument is passed to the `HttpdXmlParser` constructor.

The *`host`* object is used to manage resources for the resulting *DOM* tree. during the parse. As such it is important to ensure that the lifetime of *`host`* is greater than the *DOM* tree.

The object can not be used until the `Create` method is called first.

`Create`

int **HttpdXmlDomBuilder::Create** (void);

This method creates and initializes the builder.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Root

`HttpdXmlDomNode *`**`HttpdXmlDomBuilder::Root`**` (void);`

This method returns a pointer to the root node of the document.

Lookup

`const char *`**`HttpdXmlDomBuilder::Lookup`**` (const char *`*`p_path`*`);`

This method gets the content of the requested node. Just like a filesystem path the components of *`p_path`* are separated with `/` characters. The path is always absolute to the root of the document and should not begin with a `/` unless the root node of the document is being requested.

By default the body contents of the node is returned. However an attribute value may be selected with a `&` suffix. If the path does not reference a node in the tree then NULL is returned.

Consider the following document:

```
<a>
 <b status="enabled">
  <c id="mynode"/>
  <c id="othernode">Some Data</c>
  <d>This is node D!</d>
  <c>Final node</c>
 </b>
</a>
```

Consider the following queries:

Query	**Value**
`a/b/d`	`This is node D!`
`a/b&status`	`enabled`
`a/b/c&id`	`mynode`
`a/b/c&id`	`mynode`
`a/b/c>1`	`Some Data`
`a/b/c>1&id`	`othernode`
`a/b/c>2`	`Final node`

If the INC_XML_NAMESPACES option is enabled then there is an additional syntax to restrict a particular tag to a namespace. If no namespace is specified then the namespace is ignored. Consider the following XML document with namespace designations:

```
<a xmlns:A="alpha:" xmlns:B="beta:">
```

```
  <A:b status="enabled">
   <c id="mynode"/>
   <B:c id="othernode">Some Data</B:c>
   <d>This is node D!</d>
   <c>Final node</c>
  </A:b>
 </a>
```

To select the node `c` that is in the `beta:` namespace the path would be `a/(alpha:)b/(beta:)c`.

LookupNode

`HttpdXmlDomNode *`**`HttpdXmlDomBuilder::LookupNode`**` (const char *`*`p_xml_path`*`);`

This method returns a pointer to the node specified by *`p_xml_path`* similar to the `Lookup` method. However the & specifier for attributes is not allowed in the path string.

If the node can not be found NULL is returned.

Set

`int `**`HttpdXmlDomBuilder::Set`**` (const char *`*`p_xml_path`*`, const char *`*`p_value`*`);`

This method sets the element specified by *`p_xml_path`* to *`p_value`*. Both node bodies and attributes can be set with this method.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdXmlDomNode` Reference

Introduction

Instances of `HttpdXmlDomNode` are created and arranged into a tree structure by the related `HttpdXmlDomBuilder` class. This class is not mean to be subclassed but rather the tree of objects traversed as needed. This class is derived from `HttpdXmlNode` and supports its public interface.

Each node contains the tag, its attributes, the body content of the tag, and a list of child nodes. The tree is formed because every child node can have a list of zero or more children. A special node is created for the root of the tree. The root node always has no attributes and has a tag name of `<root>`. The body contains any text that is outside the root tags during the parse and the list of children contain the top-level tags.

Public Methods

Children

`HttpdList &`**`HttpdXmlDomNode::Children`**` (void);`

This method returns a reference to the list of child nodes contained within this node.

Parent

`HttpdXmlDomNode *`**`HttpdXmlDomNode::Parent`**` (void);`

This method returns a pointer to the parent of this node. If this is the root node of the document then NULL is returned.

Attributes

`HttpdXmlAttributes *`**`HttpdXmlDomNode::Attributes`**` (void);`

This method returns a pointer the list of attribute pairs for this node.

Body

`HttpdStringSink &`**`HttpdXmlDomNode::Body`**` (void);`

This method returns a reference to the string sink that is used to store the tags body content.

BodySignificant

`bool `**`HttpdXmlDomNode::BodySignificant`**` (void);`

This method returns true if the body of this node contains any non-whitespace characters. Because XML uses nodes both as containers of data and as structural elements often it is useful to detect structural nodes. This method can be used to help determine if the body of a node is relevant.

CopyToHead

`int `**`HttpdXmlDomNode::CopyToHead`**` (HttpdXmlHost &`*`host`*`, HttpdXmlDomNode *`*`p_parent`*`);`

This method copies the node (and all of its children). The new subtree is inserted as the first child of *`p_parent`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). This operation is atomic: either the entire tree is copied or no portion of the copy remains.

CopyToTail

`int `**`HttpdXmlDomNode::CopyToTail`**` (HttpdXmlHost &`*`host`*`, HttpdXmlDomNode *`*`p_parent`*`);`

This method copies the node (and all of its children). The new subtree is inserted as the last child of *`p_parent`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). This operation is atomic: either the entire tree is copied or no portion of the copy remains.

Lookup

`const char *`**`HttpdXmlDomNode::Lookup`**` (const char *`*`p_path`*`);`

This method gets the content of the requested node relative to this node. See HttpdXmlDomBuilder::Lookup for a description of *`p_path`*.

LookupNode

`HttpdXmlDomNode *`**`HttpdXmlDomNode::LookupNode`**` (const char *`*`p_xml_path`*`);`

This method returns a pointer to the node specified by *`p_xml_path`* similar to the `Lookup` method. However the & specifier for attributes is not allowed in the path string.

If the node can not be found NULL is returned.

Set

`int `**`HttpdXmlDomNode::Set`**` (const char *`*`p_xml_path`*`, const char *`*`p_value`*`);`

This method sets the element specified by *`p_xml_path`* to *`p_value`*. Both node bodies and attributes can be set with this method.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`AddAttribute` (namespace version)

`int `**`HttpdXmlDomNode::AddAttribute`**` (HttpdXmlHost &`*`host`*`, const char *`*`p_name`*`, const char *`*`p_value`*`, const char *`*`p_namespace`*`);`

This method adds an attribute to the node. The *`host`* argument should be the host object used during the parse of the document containing this node. *`p_namespace`* must not be NULL.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

AddAttribute

`int `**`HttpdXmlDomNode::AddAttribute`**` (HttpdXmlHost &`*`host`*`, const char *`*`p_name`*`, const char *`*`p_value`*`);`

This method adds an attribute to the node. The *`host`* argument should be the host object used during the parse of the document containing this node.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

RemoveAttribute

`bool `**`HttpdXmlDomNode::RemoveAttribute`**` (HttpdXmlAttribute *`*`p_attr`*`);`

This method removes the specified attribute object from the node.

If the attribute was present then true is returned. Otherwise false is returned.

InsertLastChild

`int `**`HttpdXmlDomNode::InsertLastChild`**` (HttpdXmlHost &`*`host`*`, const char *`*`p_tag`*`, HttpdXmlDomNode *&`*`p_new`*`, const char *`*`p_namespace`*` = "");`

This method adds a new node as the last child of the current node. The *`host`* argument should be the host object used during the parse of the document.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If successful then *`p_new`* will point to the newly created node.

`InsertFirstChild`

`int` **`HttpdXmlDomNode::InsertFirstChild`** `(HttpdXmlHost &`*`host`*`, const char *`*`p_tag`*`, HttpdXmlDomNode *&`*`p_new`*`, const char *`*`p_namespace`*` = "");`

This method adds a new node as the first child of the current node. The *`host`* argument should be the host object used during the parse of the document.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If successful then *`p_new`* will point to the newly created node.

`InsertBefore`

`int` **`HttpdXmlDomNode::InsertBefore`** `(HttpdXmlHost &`*`host`*`, const char *`*`p_tag`*`, HttpdXmlDomNode *&`*`p_new`*`, const char *`*`p_namespace`*` = "");`

This method adds a new node as a sibling prior to the current node. The *`host`* argument should be the host object used during the parse of the document.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If successful then *`p_new`* will point to the newly created node.

`InsertAfter`

`int` **`HttpdXmlDomNode::InsertAfter`** `(HttpdXmlHost &`*`host`*`, const char *`*`p_tag`*`, HttpdXmlDomNode *&`*`p_new`*`, const char *`*`p_namespace`*` = "");`

This method adds a new node as a sibling of the current node. The *`host`* argument should be the host object used during the parse of the document.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If successful then *`p_new`* will point to the newly created node.

`HttpdXmlDomWriter` Reference

Introduction

The `HttpdXmlDomWriter` utility class can be used to write `HttpdXmlDomNode` trees out as XML. The intent of using this class is as a temporary object to walk a tree of nodes and outputting the XML representation to a HttpdWritable object.

When created the `HttpdXmlDomWriter` can be configured with a variety of options to affect the generated XML.

Public Methods

`HttpdXmlDomWriter`

`HttpdXmlDomWriter::HttpdXmlDomWriter` `(HttpdWritable *`*`p_target`*`, unsigned int `*`indent`*` = 2, unsigned int `*`base_indent`*` = 0, HttpdUint8 `*`flags`*` = 0, unsigned short `*`recursion_limit`*` = USHRT_MAX);`

This method constructs the writer object to generate XML to *`p_target`*. Nodes will be written out with an initial indent of *`base_indent`* spaces. Nested nodes will be indented by an additional *`indent`* spaces.

The *`flags`* parameter can be set to any combination of the following options:

Flag	**Meaning**
`XML_OPT_ANONYMOUS_CLOSE`	A shortcut for terminating leaf nodes `</>` will be used to reduce space.
`XML_OPT_TRIM_LEADING_WS`	Leading whitespace on the text content of nodes will be removed during writing.
`XML_OPT_TRIM_TRAILING_WS`	Trailing whitespace on the text content of nodes will be removed during writing.
`XML_OPT_ALWAYS_WRITE_BODY`	Always causes the text content of nodes to be written out. Ordinarilly the writer attempts to discern structural nodes from nodes containing content. Nodes that appear to be structural are not written out. Setting this option disables this check - increasing the size of the written XML.
`XML_OPT_NO_NEWLINES`	Causes newlines normally omitted for formatting to be omitted. This results in more compact, but less readable output. This option is most effective when setting the *`indent`* and *`base_indent`* to `0`.
`XML_OPT_USE_CDATA`	Causes `CDATA[]` encoding to be used if it would result in a smaller representation. This option is ignored if INC_XML_DOM_WRITE_CDATA is zero. Enabling this option consumes more CPU time during writing.
`XML_OPT_DEFAULT_NS_USED`	Indicates that the XML being written is a fragment within a larger document. The outer document may have assigned the default namespace to something besides the null default. In this case this option should be set so that the default namespace is assigned to null if it is needed. This option is ignored if INC_XML_NAMESPACES is non-zero.
`XML_OPT_IGNORE_SELECTORS`	This option causes the XML to be serialized without attempting to use the selectors from the source document for more readable XML. This option is useful if the XML being written is a fragment within a larger document and it is important to avoid namespace selector collisions with other content. This option is ignored if INC_XML_NAMESPACES is non-zero.

The *`recursion_limit`* is used to control stack space consumption. Writing an XML document from a DOM tree is a recursive process. This parameter limits the depth of the recursion. Attempts to write a document that requires more recursion than this limit will fail.

WriteMarkup

```
int HttpdXmlDomWriter::WriteMarkup (const HttpdXmlDomNode *p_node);
```

This method writes *`p_node`* and its children.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

WriteChildren

`int` **`HttpdXmlDomWriter::WriteChildren`** `(const HttpdXmlDomNode *`*`p_node`*`);`

This method writes the children of *`p_node`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

WriteDom

`int` **`HttpdXmlDomWriter::WriteDom`** `(const HttpdXmlDomBuilder *`*`p_builder`*`);`

This method writes the document held by *`p_builder`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Chapter 7. Processing JSON

“Streamy” Processing of JSON

Seminole includes a set of classes to assist in parsing, storing, and serializing JSON data. The parser also operates in a “streamy” fashion. This means that the document can be pumped piecemeal into the parser as it arrives. Additionally is a way to efficiently “patch in” external data into a JSON data structure. Like `HttpdXmlParser`, `HttpdJsonTokenizer` is derived from HttpdFifo. This allows `POST` requests to easily drive the parser with `HttpdReceiver` or `HttpdBoundaryReader` objects.

JSON analysis is implemented in layers. The first layer, `HttpdJsonTokenizer` is an abstract class that calls methods as tokens are recognized. On top of this `HttpdJsonParser` subclasses `HttpdJsonTokenizer` to maintain state and validate the token stream against the JSON grammar. Finally `HttpdJsonBuilder` subclasses `HttpdJsonParser` and builds a data structure as parsing progresses representing the JSON input.

Additionally the JSON toolkit contains a number of classes that represent the datatypes present in JSON. These objects can serialize themselves in JSON format to an `HttpdWritable`.

The JSON framework is defined in a header file called `sem_json.h`. In order to use any of these classes or methods, this header file must be included.

`HttpdJsonTokenizer` Reference

Introduction

The `HttpdJsonTokenizer` class is used for tokenizing JSON. This pure abstract class is derived from `HttpdFifo` and calls various methods when tokens are written to the FIFO. This class is typically not used directly. Rather it serves as a base for the `HttpdJsonParser` class.

Public Methods

`HttpdJsonTokenizer`

```
HttpdJsonTokenizer::HttpdJsonTokenizer (size_t initial_buffer_size = 0,
size_t max_buffer_size = infinity);
```

This method constructs the tokenizer. The *`initial_buffer_size`* and *`max_buffer_size`* arguments control the size of the `HttpdFifo` buffer.

`Finish`

```
int HttpdJsonTokenizer::Finish (void);
```

This method should be called when no more data is written to the tokenizer. It validates that all of the written data that has been digested.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

Protected Methods

Keyword

`virtual int` **`HttpdJsonTokenizer::Keyword`** `(int` *`kw`*`);`

This method is called when a keyword is encountered. The *`kw`* parameter identifies the keyword and takes on one of the following values:

- KW_FALSE for the `false` keyword.
- KW_TRUE for the `true` keyword.
- KW_NULL for the `null` keyword.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Identifier

`virtual int` **`HttpdJsonTokenizer::Identifier`** `(const char *`*`p_identifier`*`);`

This method is called for a non-quoted string that is not a keyword.

The buffer pointed to by *`p_identifier`* is only valid for the duration of the method call. If a dynamically allocated copy is required then use `HttpdJsonTokenizer::CopyString` rather than `HttpdUtilities::SaveString` to make a heap resident copy.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

QuotedString

`virtual int` **`HttpdJsonTokenizer::QuotedString`** `(const char *`*`p_string`*`);`

This method is called when a quoted string is recognized.

The buffer pointed to by *`p_string`* is only valid for the duration of the method call. If a dynamically allocated copy is required then use `HttpdJsonTokenizer::CopyString` rather than `HttpdUtilities::SaveString` to make a heap resident copy.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Token

`virtual int` **`HttpdJsonTokenizer::Token`** `(char` *`ch`*`);`

This pure virtual method is called for the following single character tokens: `{`, `}`, `:`, `,`, `[` and `]`.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Error

`virtual int` **`HttpdJsonTokenizer::Error`** `(int` *`error_type`*`, …);`

This method is called for malformed input. The *`error_type`* parameter determines how many string parameters follow.

Error Type	**Additional Parameters**
`JSON_ERR_EXPECTING`	What was expected.
`JSON_ERR_UNEXPECTED`	The unexpected item.
`JSON_ERR_EARLY_EOF`	None.
`JSON_ERR_BAD_STR_ESCAPE`	The invalid escape sequence.
`JSON_ERR_BAD_UNICODE`	The invalid hexadecimal sequence following a `\u`.

The default implementation returns `HttpdOpSys::ERR_BADPARAM` and ignores the additional parameters. The protected data member `mLineNumber` is the current line number within the document where the error occured.

`HttpdJsonParser` Reference

Introduction

The `HttpdJsonParser` class is used for processing JSON documents. It is subclassed (via `HttpdJsonTokenizer`) from `HttpdFifo` and shares the same public interface for receiving data. Only the additional methods are documented here.

Public Methods

HttpdJsonParser

`HttpdJsonParser::HttpdJsonParser` `(HttpdUint8` *`flags`* `= 0, size_t` *`initial_buffer_size`* `= 0, size_t` *`max_buffer_size`* `=` *`infinity`*`);`

This method constructs the parser. The *`initial_buffer_size`* and *`max_buffer_size`* arguments control the size of the `HttpdFifo` buffer.

If *`flags`* has the `HttpdJsonParser::FLAG_QUOTED_KEYS_ONLY` bit set then object keys must be quoted strings and not identifiers; as required by the JSON specification.

The object can not be used until the `Create` method is called first.

Create

`int` **`HttpdJsonParser::Create`** `(void);`

This method creates and initializes the parser.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Finish

`int` **`HttpdJsonParser::Finish`** `(void);`

This method should be called after the entire document has been written to the parser. It validates that all of the JSON has been digested and that all of the state machines are in their appropriate idle states.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

Protected Methods

TrueValue

```
virtual int HttpdJsonParser::TrueValue (void);
```

This method is called when a primitive value of `true` is parsed.

FalseValue

```
virtual int HttpdJsonParser::FalseValue (void);
```

This method is called when a primitive value of `false` is parsed.

NullValue

```
virtual int HttpdJsonParser::NullValue (void);
```

This method is called when a primitive value of `null` is parsed.

StringValue

```
virtual int HttpdJsonParser::StringValue (const char *p_string);
```

This method is called when a string value is parsed. If a heap resident copy of *`p_string`* is desired then the `HttpdJsonTokenizer::CopyString` method rather than the `HttpdUtilities::SaveString` should be used.

NumericValue

```
virtual int HttpdJsonParser::NumericValue (const char *p_num);
```

This method is called when a numeric value is parsed.

Push

```
virtual int HttpdJsonParser::Push (int building);
```

This method is called when a complex object is entered to push the current state on a stack to concentrate on the newly discovered container object. The *`building`* parameter can be either `JSON_BUILD_ARRAY` or `JSON_BUILD_OBJECT` depending on what is being built.

The `mpContext` member variable may be used to hold the current context to track the complex object being assembled. If this member is used to store a pointer to dynamically allocated storage then the overridden `Pop` method must free this storage before calling `Pop` in this (the base) class.

Pop

```
virtual int HttpdJsonParser::Pop (void);
```

This method is called when a complex object is completed to pop the previous build state from the stack.

`HttpdJsonBuilder` Reference

Introduction

The `HttpdJsonBuilder` class is used for building data structures from JSON documents. It is subclassed (via `HttpdJsonParser` and then `HttpdJsonTokenizer`) from `HttpdFifo` and shares the same public interface for receiving data. Only the additional methods are documented here.

Public Methods

HttpdJsonBuilder

HttpdJsonBuilder::HttpdJsonBuilder (HttpdUint8 *flags* = 0, size_t *initial_buffer_size* = 0, size_t *max_buffer_size* = *infinity*);

This method constructs the builder. The *initial_buffer_size* and *max_buffer_size* arguments control the size of the `HttpdFifo` buffer.

The *flags* argument supports all of the options in `HttpdJsonParser`.

The object can not be used until the `Create` method is called first.

Create

int **HttpdJsonBuilder::Create** (void);

This method creates and initializes the builder.

Upon success, `0` is returned; otherwise an error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Finish

int **HttpdJsonBuilder::Finish** (void);

This method should be called after the entire document has been written to the builder. The data structure built should not be accessed until this method is called.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). The returned value is obtained from the `Error` method which may be overridden for additional error reporting.

Datum

HttpdJsonDatum ***HttpdJsonBuilder::Datum** (void); const

This method returns a pointer the datum representing the parsed JSON. The datum is owned by the builder and it will be destroyed when the builder is destroyed.

TakeDatum

HttpdJsonDatum ***HttpdJsonBuilder::TakeDatum** (void);

This method takes ownership of the datum representing the parsed JSON. The datum must be destroyed by the caller when it is no longer needed.

Note

This method may only be called once and only after `Finish()` has been called.

`HttpdJsonDatum` Reference

Introduction

The `HttpdJsonDatum` class is the abstract base class that represents JSON data elements.

Note

This class should not be subclassed outside the JSON toolkit. For efficiency the base class must know about its derived types. Therefore the primary use of this type is as a pointer to JSON data.

Public Methods

`WriteQuotedString`

```
static int HttpdJsonDatum::WriteQuotedString (HttpdWritable *p_target,
const char *p_string);
```

This static method writes *`p_string`* to *`p_target`* surrounded with double quotes escaping any characters that the JSON standard requires.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Destroy`

```
static int HttpdJsonDatum::Destroy (const HttpdJsonDatum *p_datum);
```

For efficiency some subclasses of `HttpdJsonDatum` are allocated using specialized mechanisms. Instances of `HttpdJsonDatum` should only be destroyed with this static method. The `delete` operator should never be applied to this class or its subclasses.

`Type`

```
virtual int HttpdJsonDatum::Type (void);
```

This method returns an identifier of the type of this datum.

Constant	Actual Class	Description
`TYPE_UNDEFINED`	`HttpdJsonUndefined`	Unlike `null` the undefined object type represents a value that is not possibly encoded in JSON. This type is often returned from methods to indicate that no such value exists.
`TYPE_NULL`	`HttpdJsonNull`	This value represents a `null` in JSON.

Constant	Actual Class	Description
`TYPE_STRING`	`HttpdJsonString`	This value represents a string value.
`TYPE_TRUE`	`HttpdJsonTrue`	This value represents the value `true`.
`TYPE_FALSE`	`HttpdJsonFalse`	This value represents the value `false`.
`TYPE_LONG`	`HttpdJsonLong`	This value represents a non-fractional number within the range of the long type.
`TYPE_DOUBLE`	`HttpdJsonDouble`	This value represents a number within the range of the double type.
`TYPE_ARRAY`	`HttpdJsonArray`	This value represents an array of values.
`TYPE_OBJECT`	`HttpdJsonObject`	This value represents a map of string to values.
`TYPE_ABSTRACT`	`HttpdAbstractJson`	This value represents an artificially inserted JSON body.

Serialize

virtual int **HttpdJsonDatum::Serialize** (HttpdWritable **p_target*);

This method serializes this object in JSON notation to *p_target*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Get (by key)

virtual HttpdJsonDatum ***HttpdJsonDatum::Get** (const char **p_key*);

This method returns the object associated with *p_key*. If no value is present or this value is not a container then a pointer to the undefined object is returned.

Get (by index)

virtual HttpdJsonDatum ***HttpdJsonDatum::Get** (size_t *index*);

This method returns the value at *index*. If no value is present or this value is not a container then a pointer to the undefined object is returned.

Copy

HttpdJsonDatum ***HttpdJsonDatum::Copy** (void);

This method returns copies this value (and any values it contains, recursively) and returns the copy. If there is insufficient memory then `NULL` is returned. Upon success the returned value should be destroyed (via `Destroy()`) when it is no longer needed.

IsUndefined

bool **HttpdJsonDatum::IsUndefined** (void); const

This method returns `true` if this value is undefined.

IsNull

bool **HttpdJsonDatum::IsNull** (void); const

This method returns `true` if this value is JSON `null`.

IsTrue

`bool` **`HttpdJsonDatum::IsTrue`** `(void); const`

This method returns `true` if this value is JSON `true`.

IsFalse

`bool` **`HttpdJsonDatum::IsFalse`** `(void); const`

This method returns `true` if this value is JSON `false`.

IsString

`bool` **`HttpdJsonDatum::IsString`** `(void); const`

This method returns `true` if this value is a string.

IsArray

`bool` **`HttpdJsonDatum::IsArray`** `(void); const`

This method returns `true` if this value is an array.

IsObject

`bool` **`HttpdJsonDatum::IsObject`** `(void); const`

This method returns `true` if this value is a JSON object.

IsDouble

`bool` **`HttpdJsonDatum::IsDouble`** `(void); const`

This method returns `true` if this value is a double value and floating point JSON support is enabled.

IsLong

`bool` **`HttpdJsonDatum::IsLong`** `(void); const`

This method returns `true` if this value is a long value.

IsNumber

`bool` **`HttpdJsonDatum::IsNumber`** `(void); const`

This method returns `true` if this value is a number value of either long or double type.

GetLong

`bool` **`HttpdJsonDatum::GetLong`** `(long &`*`l`*`); const`

If the value can be stored in a long then this method stores the value in *l* and returns `true`. Otherwise `false` is returned.

GetDouble

`bool` **`HttpdJsonDatum::GetDouble`** `(double &d); const`

If the value can be stored in a double then this method stores the value in *d* and returns `true`. Otherwise `false` is returned.

Note

This method is only available if INC_JSON_FLOATING_POINT is enabled.

GetString

`const char *`**`HttpdJsonDatum::GetString`** `(void); const`

If the value is a string then the string value is returned. Otherwise `NULL` is returned.

HttpdJsonUndefined Reference

Introduction

Rather than returning `NULL` to represent a missing value the `HttpdJsonUndefined` helps avoid lots of checks for `NULL` because queries into the undefined object always result in a pointer to the undefined object.

For example to get the `"name"` of the first object in the array named `"people"` in a JSON object the following code may be used:

```
HttpdJsonDatum *p_name = p_obj->Get("people")->Get(0)->Get("name");
if (p_name->IsUndefined())
  …; // On error.
else
  … // On success.
```

Because the `Get` methods never return `NULL` and the `HttpdJsonUndefined` object always returns undefined for any queries no extraneous error checking is necessary until the final step.

Note

This value will never be the result of parsing JSON using `HttpdJsonBuilder`.

Public Methods

Undefined

`static HttpdJsonDatum *`**`HttpdJsonUndefined::Undefined`** `(void);`

This method returns a pointer to the undefined object. It never returns `NULL`.

`HttpdJsonNull` Reference

Introduction

A singleton instance of this class represents all `null` JSON values.

Public Methods

`Null`

```
static HttpdJsonDatum *HttpdJsonNull::Null (void);
```

This method returns a pointer to the `null` object. It never returns `NULL`.

`HttpdJsonTrue` Reference

Introduction

A singleton instance of this class represents all `true` JSON values.

Public Methods

`True`

```
static HttpdJsonDatum *HttpdJsonTrue::True (void);
```

This method returns a pointer to the `true` object. It never returns `NULL`.

`HttpdJsonFalse` Reference

Introduction

A singleton instance of this class represents all `false` JSON values.

Public Methods

`False`

```
static HttpdJsonDatum *HttpdJsonFalse::False (void);
```

This method returns a pointer to the `false` object. It never returns `NULL`.

`HttpdJsonString` Reference

Introduction

This object represents a JSON string.

Public Methods

`Create`

`static HttpdJsonString *`**`HttpdJsonString::Create`**` (const char *`*`p_string`*`);`

This method creates an object containing the specified string value. Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

An internal copy of *`p_string`* is made.

`Wrap`

`static HttpdJsonString *`**`HttpdJsonString::Wrap`**` (char *`*`p_string`*`);`

This method creates an object pointing to *`p_string`*. Therefore *`p_string`* must be allocated on the heap. Additionally this method takes ownership of the string. It will be freed when the datum is destroyed.

Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

Note

If this method fails and returns `NULL` then *`p_string`* is automatically freed.

`String`

`const char *`**`HttpdJsonString::String`**` (void);`

This method returns the string value of this datum. This method never returns `NULL`.

`Set`

`int `**`HttpdJsonString::Set`**` (const char *`*`p_string`*`);`

This method sets the value of this string object to *`p_string`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdJsonLong` Reference

Introduction

This object represents a number that can fit into a C++ long.

Public Methods

`Create`

`static HttpdJsonLong *`**`HttpdJsonLong::Create`**` (long `*`value`*`);`

This method creates an object containing the specified value. Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

Long

```
long HttpdJsonLong::Long (void);
```

This method returns the value of this datum.

Set

```
void HttpdJsonLong::Set (long value);
```

This method sets the value of this string object to *`value`*.

`HttpdJsonDouble` Reference

Introduction

This object represents a number that can fit into a C++ double. This class is only for parsing JSON if INC_JSON_FLOATING_POINT is enabled.

Public Methods

Create

```
static HttpdJsonDouble *HttpdJsonDouble::Create (double value);
```

This method creates an object containing the specified value. Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

Double

```
double HttpdJsonDouble::Double (void);
```

This method returns the value of this datum.

Set

```
void HttpdJsonDouble::Set (double value);
```

This method sets the value of this string object to *`value`*.

`HttpdJsonArray` Reference

Introduction

This object represents an array of JSON values.

Public Methods

Create

```
static HttpdJsonArray *HttpdJsonArray::Create (void);
```

This method creates an empty array (of length `0`). Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

Set

```
int HttpdJsonArray::Set (size_t pos, HttpdJsonDatum *p_obj);
```

This method sets the value of array position *`pos`* to *`p_obj`*. If *`pos`* is beyond the end of the array then the intermedia array positions are filled with `HttpdJsonUndefined`. These entries will not not be serialized.

This method always takes onwership of *`p_obj`*. In the event of an error *`p_obj`* is destroyed. Any object previously in the slot is destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Count

```
size_t HttpdJsonArray::Count (void);
```

This method returns the number of elements (including undefined slots) in the array.

Contents

```
HttpdJsonDatum **HttpdJsonArray::Contents (void);
```

This method returns a pointer to the array of contained values. The returned pointer is only valid until the array is modified or destroyed. The returned array should be considered to only contain `Count()` pointers.

This method can be used to efficiently iterate over the contents of the array (as opposed to calling `Get()` for each element). Additionally this array may be modified provided this is done with regards to proper object ownership and lifetime.

`HttpdJsonObject` Reference

Introduction

This object represents a container of JSON values indexed by string. Internally the mapping is maintained in HttpdJsonObject::Tuple which has the following structure:

Members of HttpdJsonObject::Tuple

Type: char *
Name: *`mpKey`*
Description: The name of the value.

Type: `HttpdJsonDatum *`
Name: *`mpValue`*
Description: The value associated with this name.

Public Methods

Create

```
static HttpdJsonObject *HttpdJsonObject::Create (void);
```

This method creates an object with no members. Upon success a pointer to the object is returned. When no longer needed the object should be destroyed via the `Destroy()` method. Upon failure `NULL` is returned.

Set

```
int HttpdJsonObject::Set (const char *p_key, HttpdJsonDatum *p_obj);
```

This method stores *`p_obj`* under the name *`p_key`*. If there is a previous value stored under that key it is destroyed.

This method always takes onwership of *`p_obj`*. In the event of an error *`p_obj`* is destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Insert

```
int HttpdJsonObject::Set (char *p_key, HttpdJsonDatum *p_obj);
```

This method stores *`p_obj`* under the name *`p_key`*. This method is not valid if a value already exists under that name. However this method is more efficient than `Set()` which does handle this case.

This method always takes onwership of both *`p_obj`* and *`p_key`*. In the event of an error they are destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Count

```
size_t HttpdJsonObject::Count (void);
```

This method returns the number of elements in the object.

Remove

```
HttpdJsonDatum *HttpdJsonObject::Remove (const char *p_key);
```

This method removes the value from the object. If the value is found then a pointer to the removed value (which must be freed if not used elsewhere) is returned. If the value is not found then `NULL` is returned.

GetTuple

```
HttpdJsonObject::Tuple *HttpdJsonObject::GetTuple (size_t index);
```

This method returns the containing object for the specified position. Typically this method is used to enumerate the contents of the object by iterating the index from `0` to one less than the return value of `Count()`.

`HttpdAbstractJson` Reference

Introduction

There may be certain circumstances where a large amount of data needs to be serialized in JSON format in certain positions in a data structure. However the wrapping of scalar values as `HttpdJsonDatum` objects can consume a large amount of memory. Instances of subclasses of the `HttpdAbstractJson` abstract base class can be inserted into a JSON data structure and generate serialized data on the fly.

Use of this class is a performance optimization that should be avoided unless necessary.

Public Methods

`Copy`

virtual HttpdJsonDatum ***HttpdAbstractJson::Copy** (void);

This method should return a copy of this object in whatever manner the subclass deems necessary. In the event of failure `NULL` should be returned.

`DeleteAfterDestroy`

virtual bool **HttpdAbstractJson::DeleteAfterDestroy** (void);

This method is called when this object is passed to `HttpdJsonDatum::Destroy`. If it returns `true` then this object is deleted. Otherwise it is assumed that no further action must be taken.

Chapter 8. WebDAV Extensions

WebDAV

The `HttpdFileHandler` implements the `HEAD` and `GET` methods on top of an `HttpdFileSystem`. The `HttpdWebDAVHandler` class extends `HttpdFileHandler` with *WebDAV* support.

WebDAV stands for “Web-based Distributed Authoring and Versioning.” It is a set of extensions to the HTTP protocol which allows users to collaboratively edit and manage files via HTTP.

`HttpdWebDAVHandler` is only available if Seminole is compiled with the prerequisite features:

INC_XML_NAMESPACES
INC_MODIFIABLE_FILESYSTEMS

The WebDAV *API* is available in the `sem_webdav.h` header file.

Most general purpose operating systems provide a way to mount a WebDAV-compliant HTTP server as a networked file system. This allows easy manipulation of content exposed via the `HttpdFileSystem` interface.

`HttpdWebDAVHandler` Reference

Introduction

The `HttpdWebDAVHandler` class extends the `HttpdFileHandler` class with the WebDAV protocol.

Public Methods

`HttpdWebDAVHandler`

`HttpdWebDAVHandler::HttpdWebDAVHandler` `(const HttpdWebDAVConfiguration *`*`p_config`*`, HttpdFileSystem *`*`p_filesys`*`, const char *`*`p_root`*` = HttpdUtilities::mRoot, const char *`*`p_prefix`*` = HttpdUtilities::mRoot, HttpdUint8 `*`flags`*` = 0);`

This method constructs the handler. With the exception of *`p_config`* the other parameters behave identically to their `HttpdFileHandler` counterparts. The WebDAV components are configured with the HttpdWebDAVConfiguration structure. At least one instance of that structure must exist and *`p_config`* must point to it.

The HttpdWebDAVConfiguration structure must have a lifetime equal to or greater than the `HttpdWebDAVHandler`.

`Create`

`int` **`HttpdWebDAVHandler::Create`** `(void);`

This method initializes the handler and must be called before requests may be applied to it. An error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned. If the return is unsuccessful then the handler may not be used.

LockSessions

`HttpdSessionManager &`**`HttpdWebDAVHandler::LockSessions`**` (void);`

WebDAV locks are managed internally by an instance of `HttpdSessionManager`. This method allows access to the session manager. If extra security is desired then reference returned by this method may be used to set scrubbing parameters before `Create` is called.

This method is only available if the INC_WEBDAV_LOCKING feature is enabled.

Protected Methods

GetLockCredentials

`bool `**`HttpdWebDAVHandler::GetLockCredentials`**` (RequestState &`*`state`*`, HttpdDAVLockCredentials &`*`creds`*`);`

This method called when a WebDAV lock is placed on a resource. It allows authorization data to be extracted from the request in *`state`* and encoded in a form that can be validated later. The *`creds`* argument can be used to hold this encoded data.

If true is returned then the credentials were successful encoded (and implicitly permission was granted to take the lock). If false is returned then the `HttpdWebDAVHandler` object will not perform the lock request or respond to the client. Therefore, if false is returned then a proper response must be sent to the client.

The HttpdDAVLockCredentials is an alias for HttpdParameter and can be used to store a scalar of most types.

This method is only available if the INC_WEBDAV_LOCKING feature is enabled. The default implementation of this method simply returns true.

DestroyLockCredentials

`void `**`HttpdWebDAVHandler::DestroyLockCredentials`**` (HttpdDAVLockCredentials &`*`creds`*`);`

This method is called each time `GetLockCredentials` returns true. This gives subclasses a chance to clean up any memory or resources that may have been allocated and stored in *`creds`*.

This method is only available if the INC_WEBDAV_LOCKING feature is enabled. The default implementation of this method simply returns.

LockActionAllowed

`int `**`HttpdWebDAVHandler::LockActionAllowed`**` (const RequestState &`*`state`*`, int `*`action`*`, LockRecord *`*`p_lock`*`);`

This method is called when a WebDAV lock is referenced by a client. It can be overridden to perform custom authorization checks on the credentials gathered at lock creation time by `GetLockCredentials`.

When called *`action`* is one of the following constants indicating the desired action:

LOCK_ACT_UNLOCK: Remove (destroy) a lock

LOCK_ACT_USE: Reference the lock when modifying a locked object.
LOCK_ACT_REFRESH: Refresh a lock timer so that it does not expire.

The *`p_lock`* parameter is a pointer to the lock object. This object has a data member, `mCredentials` that is the HttpdDAVLockCredentials that was set by `GetLockCredentials`. Subclasses can check this member and determine if *`action`* is allowed.

If the action is allowed then HTTPD_RESP_OK should be returned. If access is not allowed then the returned HTTP status code is sent to the requestor.

This method is only available if the INC_WEBDAV_LOCKING feature is enabled. The default implementation of this method simply returns HTTPD_RESP_OK.

`HttpdWebDAVConfiguration` Reference

Introduction

The `HttpdWebDAVConfiguration` struct is used to configure instances of `HttpdWebDAVHandler`. Because this configuration structure is a passive entity there are no methods. Instead the members are accessed directly as needed.

Public Data

`mCapabilities`

```
… mCapabilities;
```

This member controls the capabilities clients have when accessing the *WebDAV* resource. It is a combination of the following bit flags:

HTTPD_WEBDAV_CAN_CREATE: Create new resources
HTTPD_WEBDAV_CAN_DELETE: Delete existing resources
HTTPD_WEBDAV_CAN_MKCOL: Create collection (directory) resources
HTTPD_WEBDAV_CAN_CHANGE: Change existing file resources
HTTPD_WEBDAV_ALLOW_INFINITE_LOCK: Allow locks to be taken with an infinite timeout
HTTPD_WEBDAV_READ_WRITE is shorthand for: HTTPD_WEBDAV_CAN_CREATE, HTTPD_WEBDAV_CAN_DELETE, HTTPD_WEBDAV_CAN_MKCOL, HTTPD_WEBDAV_CAN_CHANGE

`mMaxInfiniteDepth`

```
unsigned int mMaxInfiniteDepth;
```

The *WebDAV* protocol allows clients to specify the how recursive filesystem hierarchies are operated upon. The depth is specified in the protocol as either `0`, `1`, or `infinity`. Because these file operations are carried out by recursive routines a depth of infinity is impractical — especially on systems with little stack space.

This member is the maximum depth that the `HttpdWebDAVHandler` is willing to recurse when the client specifies a depth of `infinity`.

mPutTimeout

```
unsigned int mPutTimeout;
```

This parameter controls the timeout, in seconds, for reading the entity body during a `PUT` request.

mMaxLocks

```
size_t mMaxLocks;
```

This parameter controls the maximum number of lock objects that may be taken for this handler. This member is only present if INC_WEBDAV_LOCKING is enabled.

mMaxLockLifetime

```
long mMaxLockLifetime;
```

This parameter controls the maximum duration a lock may exist in seconds. If `mCapabilities` contains HTTPD_WEBDAV_ALLOW_INFINITE_LOCK then this value also governs the duration used for infinite lock timeouts as well. This member is only present if INC_WEBDAV_LOCKING is enabled.

Chapter 9. Error Logging and Reporting

Introduction

Many embedded devices are distant from the people that administrate them. This makes serial consoles as the only method of error reporting impractical.

Seminole provides an optional component for logging messages and then displaying those logged messages on demand. The `HttpdConsoleLog` (and the associated `HttpdConsoleHandler` class) provide a virtual serial console that is accessible via HTTP.

All of the definitions for the console mechanism are in the `sem_console.h`. This file automatically includes `seminole.h` if it has not been included already.

The `HttpdConsoleHandler` class is optional, if desired, a more complex mechanism involving templates can also be constructed.

`HttpdConsoleLog` Reference

Introduction

The interface of the `HttpdConsoleLog` class is generic enough to also allow the display of the console data through other mechanisms (such as a serial port). It uses the HttpdWritable interface as a destination for the console data.

The console data is kept in a circular, fixed-size buffer. This means that even in the face of total memory exhaustion the console can still be used to record events that can be accessed later when memory pressure is reduced.

Thread Safety

This class provides a thread-safe *API*. Multiple threads may call methods on a single instance of this class without issue.

Public Methods

`Create`

```
int HttpdConsoleLog::Create (size_t sz);
```

Initialize the console log. Before `Log` or `Dump` can be called, the object must be initialized with this method. The parameter *`sz`* is the number of bytes that this console should use.

On success a `0` should be returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Log`

```
void HttpdConsoleLog::Log (const char *p_str);
```

Add an entry to the log. The string *`p_str`* is added with no additional formatting to the log. Older messages are deleted as necessary.

`Dump`

`int` **`HttpdConsoleLog::Dump`** `(HttpdWritable *`*`p_target`*`, DumpMode` *`mode`* `= DUMP_ALL);`

This method dumps the contents of the log to the stream pointed to by *`p_target`*.

Because the `HttpdConsoleLog` is a circular buffer of variably-sized messages it is possible that the oldest message may be partially overwritten by the tail end of the newest message. The *`mode`* parameter selects if the partial message should be omitted. If *`mode`* is DUMP_ALL then even partially overwritten messages will be displayed. Otherwise, if *`mode`* is DUMP_CLEAN only full messages will get written.

On success a `0` should be returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Public Data

`HttpdConsoleLog` contains no publically accessible data members.

`HttpdConsoleHandler` Reference

Introduction

The `HttpdConsoleHandler` class is derived from the generic handler (`HttpdHandler`) which can be inserted into an `Httpd` object.

This handler simply sends out the contents of the console with a *MIME* type of `text/plain`. There is a virtual method, `Authorized` that can be overridden for access control.

Public Methods

`HttpdConsoleHandler`

`HttpdConsoleHandler::HttpdConsoleHandler` `(const char *`*`p_prefix`*`, HttpdConsoleLog *`*`p_log`*`, HttpdConsoleLog::DumpMode` *`mode`* `= DUMP_ALL);`

The constructor associates the handler with a URL prefix of *`p_prefix`*. The *`p_log`* parameter must point to a `HttpdConsoleLog` object that is initialized before the handler object is inserted in the server.

The optional *`mode`* parameter controls if partial log entries should be shown (DUMP_ALL) or not (DUMP_CLEAN).

Protected Methods

`Authorized`

`bool` **`HttpdConsoleHandler::Authorized`** `(HttpdRequest *`*`p_request`*`);`

This method determines if the request should be processed. The default implementation of this method simply returns true. But subclasses may wish to override this method to provide authentication.

If this method returns false then no further action is performed by `HttpdConsoleHandler`.

Public Data

`HttpdConsoleLog` contains no publically accessible data members.

Chapter 10. The Application Framework

Introduction

Seminole provides a powerful framework that handles most of the grunt work of producing intuitive web interfaces. This framework is based on the Model-View-Controller (MVC) paradigm. The framework is highly customizable and relies upon almost all of the core Seminole API's.

Overview

Developing applications using HTML and CGI is a complex task. The HTTP protocol is stateless; browsers require regeneration of an entire page on each form submission; form input objects support only the most rudimentary data types. All of these problems require careful management of state and very complex event flow. The Seminole application framework handles all of the complex machinery for a fully-functional web application.

The application framework is similar in some ways to traditional graphical interfaces: a tree of widgets (displayable objects) receives events from a dispatching mechanism and uses a rendering mechanism to update their visible state. In a traditional graphical interface, the widget tree and event handlers (and related “invisible” objects) are contained in a desktop. Some GUI's allow more than one desktop to exist at the same time; all isolated from one another.

In the application framework model the desktop is called a “session”. Depending on the method of state tracking used there can be one single shared session or multiple independent ones. Each session contains a tree of widgets that represent portions of an HTML document. Some of the widgets can be merely for structuring purposes; others can be input controls or data displays. The most important characteristic of widgets is that they maintain their state on the server side. This allows what the browser is showing to be redisplayed easily without passing large amounts of state between the browser and the server.

Rendering of widgets is done using the template engine. Starting with the root object, each widget is responsible for displaying itself as well as any children it may contain. Most often each widget is rendered using a template dedicated to that particular widget. Rather than there being a single large template for a page, content generated by the application framework is generated using the output of many small templates sewn together.

Web applications are event driven. The browser must send a request to the server before state updates can be seen; although the state of widgets can be updated at any time. When a request comes in from a browser the request is analyzed. The parameters on the incoming request are analyzed to find the event that triggered the transaction. Once the event is found, it is sent through the dispatcher. After the event is processed the root widget is painted to send updated content back to the browser.

The dispatcher acts as a registry for objects that are interested in events. Handlers register for the event stream of a session using a priority number. When a session is first created, the dispatcher has a handler installed that finds the widget that the event is targeted for and delivers the event to that widget. Other handlers can be registered as needed.

For example, dialogs listen on the event stream for any HTML field values that may be sent back from the client. This allows a partially edited dialog to update its fields with any changes the user made even if the event was not related to the dialog box. This behavior keeps the widgets as up-to-date as possible. Other handlers can register for events at an even lower priority to perform cleanup duties after the event has been dispatched to its target.

Widgets are identified by two different names. A short and simple string name (called a “local identifier”) is for widgets to be identified by their relatives. The most common use of the local identifier is for templates to reference the painting of child widgets. Widgets also possess a global identifier that is a number. While the local identifier only has to be unique with respect to siblings, the global identifier is unique amongst all of the widgets in a particular session.

The global identifier is designed to be compact as well as efficiently mapped to a particular widget handle. The global identifier is used in generated HTML to attach events to their target widgets. Unlike the local identifier, the global identifier is generated automatically by the widget manager when a widget is constructed.

Application developers can write their own widgets as well as utilizing a library of pre-defined widgets for developing interfaces. Often times an entire application can be built by writing a small amount of “glue code” on top of the widgets included with Seminole. Usually even the glue code for applications can be generated from a small specification file using the specgen tool.

`HttpdStringProvider` Reference

Introduction

To support interfaces with different languages it is important that user-visible strings are not scattered throughout application code. The `HttpdStringProvider` provides an interface to a catalog of “localized” strings. The strings are indexed by a numeric identifier. The origins of the identifier are dependent on the particular mechanism used to catalog the strings.

The type HttpdStringId is the scalar type to be used for identifying strings in a catalog. Implementations of the `HttpdStringProvider` interface should ensure that all valid values of this particular type (unsigned int) are handled.

Public Methods

Read (static buffer version)

```
int HttpdStringProvider::Read (HttpdStringId id, char *p_buf, size_t
buflen);
```

This method reads the string identified by *`id`* into the buffer provided pointed to by *`p_buf`*. If the string is longer than *`buflen`* an error of `HttpdOpSys::ERR_LIMITRCHD` is returned.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Read (dynamic buffer version)

```
int HttpdStringProvider::Read (HttpdStringId id, const char *&p_buf);
```

This method reads the string identified by *`id`* into the buffer that is managed by the implementation of the string provider. The address of the string is placed into the *`p_buf`* parameter.

When the string is no longer needed the address in the *`p_buf`* should be passed to the `HttpdStringProvider::Free` method.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”) and the value of *`p_buf`* is undefined.

Note

The address placed into *`p_buf`* upon success isn't necessarily a dynamically allocated buffer. The contents of the buffer should never be modified.

`Free`

```
void HttpdStringProvider::Free (const char *p_buf);
```

This method releases any memory associated with a string read using the dynamic version of `HttpdStringProvider::Read` (the one which takes only two parameters).

`HttpdStringBundle` Reference

Introduction

The class `HttpdStringBundle` implements the string provider interface storing strings in a catalog file generated by the msgcmp tool.

Note

Only additional methods are described here. This class implements the abstract methods in the `HttpdStringProvider` class.

Public Methods

`Open`

```
int HttpdStringBundle::Open (HttpdFile *p_file);
```

This method associates the file pointed to by *`p_file`* with the string bundle.

On success a `0` should be returned; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Note

This function must be called and succeed before any of the `HttpdStringProvider` methods are called.

`HttpdStringTable` Reference

Introduction

The `HttpdStringTable` class implements the string provider interface with a lower code-size footprint than the `HttpdStringBundle` class. Rather than use external files, the strings are dispensed from a statically initialized array.

For systems where code is executed directly from flash memory and there is free flash memory this implementation of `HttpdStringProvider` is also much faster. Unlike `HttpdStringBundle` it is also harder to translate the strings or localize particular builds because a recompile is necessary (rather than just replacing a file).

Note

Only additional methods are described here. This class implements the abstract methods in the `HttpdStringProvider`.

Public Methods

`HttpdStringTable`

`int` **`HttpdStringTable::HttpdStringTable`** `(const char **`*`pp_table`*`, size_t` *`count`*`);`

This method initializes the string table object. The *`pp_table`* parameter points to an array of pointers to the strings. The size of the table is limited to *`count`* strings.

When fetching strings the identifier is the index into the array. It is up to the programmer to maintain appropriate symbolic constants for each string.

`HttpdWidgetConfig` Reference

Introduction

The `HttpdWidgetConfig` acts as an interface to a collection of objects that are used to visually represent a web application. This provides a single mechanism to change the entire look and feel of an application.

Resources are most often templates but can be any data stored in files. Resources are identified by string names (which may or may not map to file names). Implementors of the `HttpdWidgetConfig` interface can be chained so that the newest addition to the chain has the first chance of resolving the resource identifier to a valid file.

This nesting allows some widgets to be given a different look and feel easily. The interface specified by the `HttpdWidgetConfig` always returns loaded `HttpdFileInfo` objects. To increase performance some implementations of the `HttpdWidgetConfig` can cache these objects to save expensive filesystem searches.

Public Methods

`Resource`

`HttpdFileInfo *` **`HttpdWidgetConfig::Resource`** `(const char *`*`p_resource_name`*`, HttpdWidgetConfig *&`*`p_config`*`);`

This method finds the resource identified by the name *`p_resource_name`*. The object that this method is applied to is searched first, if the resource is not there, each parent in the chain of `HttpdWidgetConfig` objects is searched until a resource is found.

If no resource can be found by that name, NULL is returned. If a resource is found, *`p_config`* is set to point to the configuration object that found the resource.

`Release`

`void` **`HttpdWidgetConfig::Release`** `(HttpdFileInfo *`*`p_resource`*`);`

Once a resource is found using the `Resource` method, it should be released by calling this method on the object that found the resource (which is available from the *`p_config`* parameter of the `Resource` method).

The default implementation of this method simply returns. Implementations of the `HttpdWidgetConfig` interface can override this method if cleanup on the returned `HttpdFileInfo` object is required.

Note

This method should only be called for objects returned in the *`p_config`* parameter of the `HttpdWidgetConfig::Resource` method.

Strings

`int` **`HttpdWidgetConfig::Strings`** `(const char *`*`p_resource`*`, HttpdStringBundle &`*`bundle`*`);`

This method initializes the string bundle *`bundle`* from the file identified by the resource name *`p_resource`*. If successful, the file object associated with the string bundle must be closed explicitly before the string bundle object is destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Protected Methods

HttpdWidgetConfig

`HttpdWidgetConfig::HttpdWidgetConfig` `(HttpdWidgetConfig *`*`p_parent`*`);`

This constructor initializes the base class and sets the parent widget configuration to *`p_parent`*. If there is no parent to this configuration, then *`p_parent`* should be set to NULL.

FindResource

`HttpdFileInfo *` **`HttpdWidgetConfig::FindResource`** `(const char *`*`p_resource_name`*`);`

This pure virtual method must be implemented by subclasses. When invoked, the current object should search for a resource *`p_resource_name`* in the objects list of resources. If the resource is found, a pointer to the `HttpdFileInfo` object should be returned.

If the named resource could not be found, this method should return NULL. By returning NULL the `HttpdWidgetConfig::Resource` method will continue searching in other objects.

HttpdResourceMap Reference

Introduction

`HttpdResourceMap` implements the `HttpdWidgetConfig` interface using a sorted table that maps resource identifiers to file names. This class also caches all of the `HttpdFileInfo` objects upon initialization, making access to resources (when using some filesystems) much faster.

Public Types

```
struct ResourceMap
{
 const char *mpResourceName;
 const char *mpFileName;
};
```

Public Methods

`HttpdResourceMap`

`HttpdResourceMap::HttpdResourceMap` `(HttpdWidgetConfig *`*`p_parent`*`);`

This method initializes the `HttpdResourceMap` object and assigns *`p_parent`* as the objects parent. The object can not be used for resolving resources until the `HttpdResourceMap::Load` method is called and completes successfully.

`Load`

`int` **`HttpdResourceMap::Load`** `(const ResourceMap *`*`p_map`*`, size_t` *`sz`*`, HttpdFileSystem *`*`p_fsys`*`);`

This method prepares the resources in the table specified by *`p_map`*. The *`sz`* parameter specifies the number of entries in the *`p_map`* table. All of the pathnames in the *`p_map`* table should reside in the filesystem specified by *`p_fsys`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdAppTemplateEnvironment` Reference

Introduction

`HttpdAppTemplateEnvironment` is the top-level symbol table used when painting widgets. It serves two fundamental purposes. First it provides a standard set of directives that widget templates can use. Second, it serves as an anchor for other symbol tables to find the widget and `HttpdAppPainter` objects.

The `HttpdAppTemplateEnvironment` is created by instances of `HttpdAppTemplateProcessor`. During painting the symbol table objects of widgets can obtain a pointer to the `HttpdAppTemplateEnvironment` from the top symbol table pointer:

```
HttpdAppTemplateEnvironment *p_env =
   (HttpdAppTemplateEnvironment *)p_command->Processor()->Top();
```

Template Directives

The `HttpdAppTemplateEnvironment` class provides many symbols for widget templates.

Table 10.1. Evaluation Directives

`tag`	This evaluates to a unique identifier for this widget that can be used for CGI parameters.
`event`	This evaluates to a unique identifier for this widget that when sent from the browser results in an event being dispatched to this widget. Generally this is used to name submit buttons.
`url`	This is the URL of the current application.
`localid`	This evaluates to the local identifier of the widget.
`session/`*`name`*	This calls the session objects `Attribute` method with a parameter of *`name`*. This mechanism acts as an escape for the session object to be used for any widget template if so desired.

Table 10.2. Conditional Directives

`is-hidden`	This evaluates to true if the `HTTPD_WIDGET_HIDDEN` flag is set for this widget.
`is-disabled`	This evaluates to true if the `HTTPD_WIDGET_DISABLED` flag is set for this widget.

Public Methods

Widget

`HttpdWidget *` **`HttpdAppTemplateEnvironment::Widget`** `(void);`

This method returns a pointer to the widget currently being painted. The return value can never be NULL.

Painter

`HttpdAppPainter &` **`HttpdAppTemplateEnvironment::Painter`** `(void);`

This method returns a reference to the current painter object. The painter object is responsible for painting all widgets during a particular request from the browser.

`HttpdAppTemplateProcessor` Reference

Introduction

`HttpdAppTemplateProcessor` is a subclass of `HttpdTemplateProcessor` that is used for painting widgets. Each widget gets its own instance of `HttpdAppTemplateProcessor` during the particular paint cycle. This is true assuming the widget has not overridden `HttpdWidget::Paint` to paint the widget using an alternative mechanism.

This class also has several static helper methods for painting operations. These methods should only be called when template commands are being processed by an instance of

`HttpdAppTemplateProcessor`. Although, it is not necessary to pass the pointer to the template processor around, it can be obtained easily from any of the command objects.

Unlike its base class, a `HttpdAppTemplateProcessor` automatically installs an instance of `HttpdAppTemplateEnvironment` as the first entry in the template symbol table.

Public Methods

HttpdAppTemplateProcessor

HttpdAppTemplateProcessor::HttpdAppTemplateProcessor (HttpdAppPainter &*painter*);

The constructor takes a reference to the current painting object and initializes the processor for the current painting cycle. Once constructed, the `StartProcessing` method should be called to initiate the painting cycle.

StartProcessing

int **HttpdAppTemplateProcessor::StartProcessing** (const char **p_resource*, HttpdWidget **p_widget*);

Process a template specified by the resource name *p_resource* for the widget *p_widget*. Symbols should be associated with the template processor before this method is called.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

WriteResourceString

int **HttpdAppTemplateProcessor::WriteResourceString** (HttpdTemplateCommand **p_command*, HttpdStringId *string*);

This static method writes the localized string identified by *string* to the output associated with *p_command*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

GetPainter

HttpdAppPainter * **HttpdAppTemplateProcessor::GetPainter** (HttpdTemplateCommand **p_command*);

This static method returns a pointer to the painter object that is managing the current paint cycle. `GetPainter` should not be called if the current template command is not associated with a `HttpdAppTemplateProcessor` object. This method will never return NULL.

GetWidget

HttpdWidget * **HttpdAppTemplateProcessor::GetWidget** (HttpdTemplateCommand **p_command*);

This static method returns a pointer to the widget that is being painted. `GetWidget` should not be called if the current template command is not associated with a `HttpdAppTemplateProcessor` object. This method will never return NULL.

`HttpdAppStringConstants` Reference

Introduction

The `HttpdAppStringConstants` structure represents a table of strings (which may or may not be localized) indexed by name.

A common use for this class is to handle HTTPD_DLG_TEMPLATE_EVAL events in dialog widgets. This allows widgets to substitute different strings depending on their runtime state.

Although the tables to describe the strings can be built manually, using specgen to generate the tables is the preferred method.

Public Methods

`WriteConstant`

`int` **`HttpdAppStringConstants::WriteConstant`** `(HttpdTemplateCommand *`*`p_command`*`, const char *`*`p_label`*`);`

This method writes the string identified by *`p_label`* in place of the template command identified by *`p_command`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If the string identified by *`p_label`* is not in the table, `ERR_NOTFOUND` is returned.

Public Data

`mpIndex`

This is a pointer to an array of StringIndex records. This array maps the symbolic string names to the correct string identifier or constant string offset.

Note

The table must be sorted so that the labels are in ascending order; a binary search is used to find the target record.

`mCount`

This is the number of records pointed to by `mpIndex`.

`mppStrings`

This is a table of non-localized strings. The first byte of each label string in `mpIndex` is byte that determines if the associated HttpdStringId is a localized string identifier or a non-localized string. If the byte is zero, then a non-localized string is assumed. In that case the string identifier is an index into this table.

`HttpdWidget` Reference

Introduction

`HttpdWidget` is the base class for all widgets in the application framework. Although the `HttpdWidget` is not abstract it is designed to be sub-classed. Widgets are arranged in a hierarchy and, with the exception of the root widget, all widgets have a parent widget. Widgets are always stored on the heap and should only be stored in memory allocated via HttpdOpSys::Malloc. To keep the widget tree consistent widgets should never be destroyed using `delete`; instead call the `Destroy` method to destroy a widget.

By default an instance of `HttpdWidget` can not respond to a painting request correctly. Subclasses must either override `HttpdWidget::PaintingResource` or `HttpdWidget::Paint` to handle painting requests. If `PaintingResource` is overridden then the widget will handle painting requests using the template mechanism. Otherwise, the `Paint` method can be overridden to handle painting using any method the implementor desires.

Public Methods

`HttpdWidget`

`HttpdWidget::HttpdWidget` `(const char *`*`p_local_id`*`, HttpdWidgetContainer *`*`p_parent`*`, int &`*`rc`*`);`

Construct a widget object. The *`p_local_id`* parameter can be NULL if this widget should not have a local identifier. To create a top-level widget *`p_parent`* should be set to the address of the root widget, obtained by calling `HttpdAppSession::Root`.

If widget construction fails this constructor sets *`rc`* to an error value (see Table 4.1, “OS Abstraction Layer Error Codes”). Subclasses should check for success (*`rc`* equals zero) to avoid further construction. In addition, constructors of subclasses can return their own error codes in *`rc`*.

Note

It is important that subclasses handle failed construction gracefully. Code that creates a widget should check if the value in *`rc`* is non-zero. If so, the `Destroy` method should be invoked on the partially constructed widget. Therefore, if construction fails the widgets destructor is still invoked and should handle the partial construction case.

`Destroy`

`void` **`HttpdWidget::Destroy`** `(void);`

This method handles graceful destruction of a widget. Widgets should never be destroyed any other way. This ensures that a widget can properly clean-up after its self before its virtual destructor gets invoked. Once a widgets virtual destructor is invoked, no more virtual methods can be called on the widget. The `Destroy` method handles cleaning up the widget and eventually releasing its memory.

Subclasses of `HttpdWidget` should override `Destroy` to handle cleanup. Overridden versions should always call the `Destroy` method of the superclass as the very last operation.

`LocalId`

```
const char * HttpdWidget::LocalId (void);
```

Returns a pointer to the local identifier of the widget. If the widget does not have a local identifier then NULL is returned.

`Config` (Getter)

```
HttpdWidgetConfig * HttpdWidget::Config (void);
```

This method returns a pointer to the resource manager associated with the widget. The resource manager is responsible for setting the look-and-feel policy for the widget. By default, the `HttpdWidgetConfig` pointer is inherited from the parent widget.

This method can never return NULL.

`Config` (Setter)

```
void HttpdWidget::Config (HttpdConfig *p_config);
```

This method sets the current resource manager of the widget to *`p_config`*. It is up to the caller to ensure that the lifetime of the resource manager specified by *`p_config`* exceeds the lifetime of the widget.

`Flags` (Getter)

```
HttpdWidgetFlags HttpdWidget::Flags (void);
```

Each widget has a small scalar value that keeps a variety of flag bits. The meaning of some bits are common to all widgets. Others are free for subclasses to use. This method returns the current value of the flags for the widget.

Table 10.3. Widget Flags

Flag Name	Description
`HTTPD_WIDGET_CONTAINER`	This flag is set if the widget is the subclass `HttpdWidgetContainer`. Only container widgets should set this flag.
`HTTPD_WIDGET_HIDDEN`	If this flag is set the widget should not output any content during a painting cycle.
`HTTPD_WIDGET_DISABLED`	A widget in the disabled state should not respond to external events.
`HTTPD_WIDGET_STATIC_STATE`	This flag only applies to child widgets of a `HttpdWidgetDialog` widget. If set, controls will not receive updates during the manipulation of the dialog.
`HTTPD_WIDGET_DEFUNCT`	This flag can be set to have the event dispatcher automatically destroy a widget when the current event dispatching cycle is over. This is useful when a widget may still need to exist further on during the event handling chain but must be cleaned up before the painting cycle.

Flag Name	Description
`HTTPD_WIDGET_USER_FLAG`	This flag (and all of the remaining space) are available for subclasses to use freely.

`Flags` (Setter)

```
void HttpdWidget::Flags (HttpdWidgetFlags flags);
```

This sets the widget flags to the value of *`flags`*. The previous state of all flags is erased and replaced with the flags set in *`flags`*.

`GlobalId`

```
HttpdWidgetId HttpdWidget::GlobalId (void);
```

This method obtains the global identifier assigned to the widget during construction.

`Parent`

```
HttpdWidgetContainer * HttpdWidget::Parent (void);
```

This method returns a pointer to the parent widget of this widget.

`Session`

```
HttpdAppSession * HttpdWidget::Session (void);
```

This method returns a pointer to the session that owns this widget.

`Event`

```
int HttpdWidget::Event (HttpdAppEvent &event);
```

Process an event for this widget. Events dispatched to this widget are handled by this virtual method. Subclasses should override this method if they are expecting to handle events.

Implementations of this method should return `0` upon success or an error value (see Table 4.1, “OS Abstraction Layer Error Codes”).

`ActionVa`

```
int HttpdWidget::ActionVa (unsigned int req, va_list va);
```

This virtual method functions as a “catch-all” point for various miscellaneous services a widget can provide. Rather than defining additional virtual methods and type-casting `HttpdWidget` pointers, the operation can be posted to `ActionVa` (or by using the `HttpdWidget::Action` wrapper) and send to the widget. If the requested action, specified by *`req`* can not be performed by this widget, `HttpdOpSys::ERR_WRONGTYPE` is returned. Other operations should return an appropriate status code.

Subclasses of `HttpdWidget` can override this method and handle specific requests. Requests that are not understood should be passed to the `ActionVa` method of the superclass.

Action

int **HttpdWidget::Action** (unsigned int *req*, …);

This method is a wrapper for invoking the `ActionVa` method of `HttpdWidget`. The variable argument list is packaged into a va_list before `ActionVa` is called and cleaned up after it returns.

Paint

int **HttpdWidget::Paint** (HttpdAppPainter &*paint*);

The default implementation of this method is to paint the widget using templates. If a different approach to painting a widget is to be employed subclasses can override this method.

Key

void **HttpdWidget::Key** (char **p_key*, const char *prefix* = 't');

This method computes a unique prefix for this particular widget. The prefix is based upon the global identifier and as such is unique across all widgets in a particular session. This prefix is useful for naming CGI parameters that are specific to this particular widget.

The *p_key* parameter must point to a buffer of at least `HTTPD_WIDGET_KEY_LEN` characters. The *prefix* character determines the type of prefix generated. Certain characters have special meaning. The character `e` identifies the string as an event. The default character, `t` is commonly used to identify data parameters such as field values.

If a widget needs multiple prefixes an additional identifier should be concatenated to the result in *p_key* rather than altering *prefix*.

Key

void **HttpdWidget::Key** (char **p_key*, const char **p_suffix*, const char *prefix* = 't');

This method generates a unique identifier for the particular widget. The two-parameter version of `Key` is used to create a prefix which is then prepended to *p_suffix*. The result is placed in the buffer pointed to by *p_key* which must be `HTTPD_WIDGET_KEY_LEN` characters longer than the length of the string pointed to by *p_suffix*.

Protected Methods

PaintingResource

const char * **HttpdWidget::PaintingResource** (void);

If subclasses do not override the `HttpdWidget::Paint` method they should override this method to return the resource name of the template file that should be used when painting this widget.

Implementations of this method should never return NULL from this method. Only a pointer to the name of a valid file resource should be returned.

ExecuteTemplate

int **HttpdWidget::ExecuteTemplate** (HttpdAppTemplateProcessor &*proc*);

Subclasses that use template-based painting can override this method to add additional template symbols to the template processor. This method should declare any additional symbol tables and link them to the template processor using instances of `HttpdTemplateScope`.

After all the appropriate symbols are bound to *`proc`* the `ExecuteTemplate` should call `ExecuteTemplate` in its super class.

The return value from the call to the superclass `ExecuteTemplate` should be returned to the caller. If painting is successful, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

`HttpdWidgetContainer` Reference

Introduction

The `HttpdWidgetContainer` is a subclass of `HttpdWidget` and behaves as a widget that can also contain child widgets. An instance of `HttpdWidgetContainer` can also contain children derived from `HttpdWidgetContainer` resulting in a tree of widgets.

All of the protected and public methods described in the documentation for `HttpdWidget` can be overridden in subclasses of `HttpdWidgetContainer` as well with the expected behavior. The only catch is to ensure that when `HttpdWidgetContainer` is used as the class name when calling superclass methods such as `Destroy` or `ExecuteTemplate`.

Template Directives

Templates for `HttpdWidgetContainer` and its subclasses can use additional directives for managing child widgets.

Table 10.4. Evaluation Directives

`child/`*`xxxx`*	This paints a the child widget identified by the local identifier *`xxxx`*.
`child`	When inside a `children` loop, this is directive paints the currently iterated child.

Table 10.5. Conditional Directives

`has-children`	This evaluates to true if the widget has any children.

Table 10.6. Loop Directives

`children`	This iterates the body for each of the child widgets. The `child` evaluation directive paints the currently iterated child.

Public Methods

`HttpdWidgetContainer`

`HttpdWidgetContainer::HttpdWidgetContainer` `(const char *`*`p_local_id`*`, HttpdWidgetContainer *`*`p_parent`*`, int &`*`rc`*`);`

Construct a the container widget. The parameters are identical to the parameters of the `HttpdWidget` constructor.

DestroyAllChildren

`void` **`HttpdWidgetContainer::DestroyAllChildren`** `(void);`

This method destroys all of the child widgets. The parent widget is not destroyed. This method is implicitly called when the parent is destroyed.

FindByLocalId

`HttpdWidget *` **`HttpdWidget::FindByLocalId`** `(const char *`*`p_id`*`);`

Find the child widget identified by the local identifier *`p_id`*. If the child widget is found then a pointer to it is returned. If the child widget could not be found then NULL is returned.

Children

`HttpdList &` **`HttpdWidgetContainer::Children`** `(void);`

This method returns a reference to the `HttpdList` that holds references to the child widgets. The list can be iterated using an instance of `HttpdListIterator` where the owner pointer is a direct pointer to the widget.

Protected Methods

RemovingChild

`void` **`HttpdWidgetContainer::RemovingChild`** `(HttpdWidget *`*`p_widget`*`);`

This virtual method is called when a child widget of this widget is destroyed. This can be either because the container widget itself is being destroyed or any one of its children are being destroyed. In the former case `RemovingChild` is called for each widget being removed.

Subclasses of `HttpdWidgetContainer` can override this method to perform additional processing when a child is destroyed.

`HttpdAppEvent` Reference

Introduction

The `HttpdAppEvent` structure wraps up an event that is dispatched to `HttpdWidget::Event` methods. There are no public methods in this structure. It functions as a wrapper for the current state during event dispatching to avoid passing around lots of parameters.

Public Data Members

mpPath

```
const char *mpPath
```

The `mpPath` member is the result from the dispatcher calling `HttpdHandler::IsMyPath`. It is never NULL but can be the empty string.

mpEvent

```
const char *mpEvent
```

The `mpEvent` member is the event name from the dispatched event. The event names come from key values generated by the `HttpdWidget::Key` methods. Like `mpPath` this member can not be NULL.

mpRequest

```
HttpdRequest *mpRequest
```

`mpEvent` is a pointer to the current HTTP request object. This member variable can never be NULL.

mpTarget

```
HttpdWidget *mpTarget
```

This is the target widget for this event. It is possible for this member to be NULL if the widget that was sent the event was deleted on the server side before the request from the client side was processed.

mpHandler

```
HttpdHandler *mpHandler
```

This is a pointer to the handler object processing the request. The `mpHandler` member can never be NULL or else events couldn't even be processed.

mpSession

```
HttpdAppSession *mpSession
```

This is the session object associated with this event. Events are never processed unless a valid session exists for them; therefore this member can never be NULL.

mParameters

```
HttpdCgiHash mParameters
```

This is a collection of all of the parameters provided as part of the request. This includes both parameters that are part of the URI query string as well as any data associated with the `POST` method.

Note

As a general rule widgets should only access or generate parameters with names that were created using one of the `HttpdWidget::Key` methods. This provides each widget with its own name space and prevents widgets from interfering with one another.

`mPerformPaint`

```
bool mPerformPaint
```

This data member is initialized to `true` by the handler before it is dispatched. It should be set to `false` if a widget determines that a paint cycle should not be requested after event processing.

In general setting this flag to `false` should only be used for extreme failures or for cases where a widget is performing very specialized painting. “Normal” applications in general should not modify this member.

`HttpdAppPainter` Reference

Introduction

The `HttpdAppPainter` structure is created during a painting cycle to hold information that is shared by all widgets during a painting cycle. A reference to this structure is passed to `HttpdWidget::Paint` methods.

Public Data Members

`mpEvent`

```
HttpdAppEvent *mpEvent
```

The `mpEvent` member points to the current event that initiated a painting cycle.

`mpOutput`

```
HttpdDynamicOutput *mpOutput
```

This is an instance of `HttpdDynamicOutput` that is used to send content out to the client. By the time the `HttpdWidget::Paint` method is called the header phase is complete and only the `Body` method should be called on `mpOutput`.

`HttpdAppEventHandler` Reference

Introduction

When an request becomes an event inside `HttpdAppHandler` it is dispatched through a set of handlers that are all given a chance to handle the message. Each of these handlers is a subclass of `HttpdAppEventHandler`. A per-session instance of `HttpdAppDispatcher` maintains a list of `HttpdAppEventHandler` objects.

Public Methods

`HttpdAppEventHandler`

`**HttpdAppEventHandler::HttpdAppEventHandler** (HttpdAppDispatchPriority *pri*);`

The constructor initializes the event handler object and sets its priority to *`pri`*. The type HttpdAppDispatchPriority is a scalar value that is used to define a priority value relative to other event handlers.

Absolute values should never be specified for *`pri`*. Instead, offsets relative to a known priority should be used. Event dispatching to widgets functions at `HTTPD_APP_DEFAULT_PRI` priority. The final cleanup of events is handled at `HTTPD_APP_CLEANUP_PRI`. Normal event handlers should be positioned anywhere from `HTTPD_APP_DIALOG_PRI` to just below `HTTPD_APP_CLEANUP_PRI`.

The numerically higher the priority number the lower the priority of the event handler. The lower the priority of the event handler the later (in time) the handler gets its turn to handle the event.

Note

`HttpdAppEventHandler` objects should always be stored in storage obtained from HttpdOpSys::Malloc. Storage for the object is released by the `HttpdAppEventDispatcher` object when the handler is no longer needed. Calling the `Release` method queues an event handler for destruction when it is no longer in use.

`HandlerNode`

`HttpdListNode * **HttpdAppEventHandler::HandlerNode** (void);`

Instances of `HttpdAppEventHandler` are tracked in a `HttpdList` object. This method returns a pointer to the internal `HttpdListNode` that links this event handler into the list. The owner pointer of the node should always point to the `HttpdAppEventHandler` object.

`Release`

`void **HttpdAppEventHandler::Release** (void);`

To ensure the event handler is never removed before it may be needed the `delete` operator should not be used to destroy the object. The `HttpdAppEventDispatcher` handles destruction of the event handler object when it is safe.

Calling this method marks the event handler for pending deletion.

`HandleEvent`

`int` **`HttpdAppEventHandler::HandleEvent`** `(HttpdAppEvent &`*`event`*`, bool &`*`cont`*`);`

This pure virtual method must be overridden by subclasses to perform the specialized action during an event. The event is passed in as the parameter *`event`*. If no further event processing should be performed the *`cont`* should be set to `false`.

Unless this is the lowest priority (highest numerically) event handler or the *`cont`* parameter is set to `false` the return value of this method is ignored.

If this instance is the lowest priority handler or if *`cont`* is set to `false` then the return value is returned from the `HttpdAppDispatcher::HandleEvent` method.

`HttpdAppEventDispatcher` Reference

Introduction

The `HttpdAppEventDispatcher` class contains a list of `HttpdAppEventHandler` objects. When an event comes in from a browser, the `HttpdAppEvent` is given to each `HttpdAppEventHandler` object for processing.

By default the `HttpdAppEventDispatcher` object has an internal handler that dispatches an event to the `HttpdWidget::Event` method (called the “default event handler”). Other handlers can be installed to hook the event stream. The most important use of this feature is the mechanism that `HttpdWidgetDialog` uses to keep its control widgets in sync with the updates from the browser.

The `HttpdAppEventDispatcher` object also contains a low-priority “cleanup event handler” which executes after the default event handler to perform housekeeping tasks for the event dispatcher itself.

Public Methods

`List`

`HttpdList &` **`HttpdAppEventDispatcher::List`** `(void);`

Instances of `HttpdAppEventHandler` are tracked in a `HttpdList` object. This method returns a pointer to the internal `HttpdList` that tracks the event handlers.

`Insert`

`void` **`HttpdAppEventDispatcher::Insert`** `(HttpdAppEventHandler *`*`p_handler`*`);`

`Insert` inserts the event handler identified by *`p_handler`* into the dispatcher.

Note

Event handlers should be inserted using this method and not by inserting directly into the list returned by the `List` method.

`Default`

`HttpdAppEventHandler *` **`HttpdAppEventDispatcher::Default`** `(void);`

This method returns a pointer to the default event handler.

`HandleEvent`

`int` **`HttpdAppEventHandler::HandleEvent`** `(HttpdAppEvent &`*`event`*`);`

This method dispatches *`event`* through the handlers. The return value is the return value of the `HandleEvent` method of the last `HttpdAppEventHandler` object that processed the event.

`HttpdAppSession` Reference

Introduction

A session object represents all of the data identifying the state of a particular user interface. An application consists of one or more session objects. In turn, each session object contains a unique tree of widgets and event dispatcher. Each user of a web application gets their own unique instance of `HttpdAppSession`.

Application session objects are not related to `HttpdSessionObject` objects. Application session objects exist without reguard to their attachment to a particular client browser. This policy is set by subclasses of the application handler object `HttpdAppHandler`.

However, `HttpdSessionObject` objects can be used to manage multiple `HttpdAppSession` objects if desired. For simple devices where only one user will be accessing the device at a time a single session can be used to reduce code size.

Public Methods

`HttpdAppSession`

`HttpdAppSession::HttpdAppSession` `(HttpdWidgetConfig *`*`p_config`*`, HttpdStringProvider *`*`p_strings`*`);`

The *`p_config`* parameter points to an instance of the HttpdWidgetConfig class. This is the default widget configuration that is inherited for all widgets contained in this session.

The session object also holds a pointer to a string provider passed in as *`p_strings`*. This string provider is consulted by the widget painting code to obtain replaceable strings when needed.

Both the configuration object and the string provider work together to ensure application code does not contain any user-visible strings. This behavior makes it easy to allow multiple sessions for users of different languages to coexist simultaneously.

Note

The constructor does not perform a complete initialization. The `Create` method should be called after object construction to perform additional initialization.

`Root`

`HttpdWidgetRoot *` **`HttpdAppSession::Root`** `(void);`

This method returns a pointer to the root widget of the session. The root widget is an instance of a class called `HttpdWidgetRoot` and is a subclass of `HttpdWidgetContainer`. The returned pointer can never be NULL.

Dispatcher

`HttpdAppDispatcher & **HttpdAppSession::Dispatcher** (void);`

Each session has a unique instance of `HttpdAppDispatcher` to route events through interested handlers. This method returns a reference to the dispatcher object.

Mutex

`HttpdMutex & **HttpdAppSession::Mutex** (void);`

Event processing for a session is synchronized using a mutex object. In normal operation the `HttpdAppHandler` object will lock the mutex during the time an event is being dispatched to a session. This prevents simultaneous requests from arriving (possibly on different worker threads) and causing data structure corruption within the session or widget tree.

Note

Application code that is not running under the context of an incoming request for this session should lock this mutex while performing any operations with the session or its associated widgets.

Strings

`HttpdStringProvider * **HttpdAppSession::Strings** (void);`

This method returns the pointer to the string provider object that was passed in at object construction time. Unlike the widget configuration only the session manager maintains a pointer to the string provider.

When needed by widget painting code the string provider should obtained using this method.

Create

`int **HttpdAppSession::Create** (void);`

This method performs further initialization on the session object. After construction this method should be called before the session is allowed to process an event. If this method returns failure the object should be destroyed and the request should be failed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Attribute

`int **HttpdAppSession::Attribute** (HttpdWritable *p_out, const char *p_attr, HttpdTemplateCommand *p_command);`

This virtual method is called when the template tag `session/xxxx` is evaluated. The string after the forward slash is placed in *`p_attr`* and the command being executed is placed in *`p_command`* and this method is called.

Subclasses of that wish to provide global data to widget templates should override this method and write output to *`p_out`*. If the string in *`p_attr`* is not an attribute the subclass is interested in control should be passed to the `Attribute` method of the super-class.

Upon success, this method should return `0`; otherwise a system dependent error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdAppHandler` Reference

Introduction

The `HttpdAppHandler` class is a subclass of `HttpdHandler` that manages the dispatching of events to sessions. `HttpdAppHandler` is an abstract class and requires that subclasses provide a session management policy.

Protected Methods

GetSession

`HttpdAppSession * `**`HttpdAppHandler::GetSession`**` (HttpdAppEvent &`*`ev`*`);`

This abstract method should be implemented by subclasses. For the incoming event, *`ev`* the associated session object should be found and returned.

If NULL is returned then the handler assumes that `GetSession` detected an error condition and sent an appropriate response. It is important that in the event of returning NULL the `GetSession` always issues a call to `Respond` on the `mpRequest` member of *`ev`*.

ReleaseSession

`void `**`HttpdAppHandler::ReleaseSession`**` (HttpdAppSession *`*`p_session`*`);`

This abstract method is called after the handler has dispatched an event to *`p_session`*. Subclasses should perform any cleanup work on the session during this method.

`HttpdAppHandler` will never call this method with a *`p_session`* value of NULL.

ContentType

`void `**`HttpdAppHandler::ContentType`**` (HttpdAppPainter &`*`painter`*`);`

This method is called to generate the `Content-Type` *MIME* header for a normal painting cycle response. Subclasses can override this method to send out additional headers or send out a different `Content-Type` header.

This method is called during the header phase of the response. Headers should be submitted using the `Header` method of the `mpOutput` member of *`painter`*.

`HttpdSingleSessionApplication` Reference

Introduction

`HttpdSingleSessionApplication` is a subclass of `HttpdAppHandler` that implements a simple policy for session management. Every request for the application shares a single session. This class is most appropriate for very low-end hardware with little memory and is only administrated by one person at a time.

Because of its simplicity this application handler also presents the simplest interface to the programmer. The session object is created or statically declared in the application-specific code and a given to the `HttpdSingleSessionApplication` object during its construction.

Public Methods

`HttpdSingleSessionApplication`

`HttpdSingleSessionApplication::HttpdSingleSessionApplication` `(const char *`*`p_prefix`*`, HttpdAppSession *`*`p_session`*`);`

Construct a single session application handler. The handler assigned *`p_prefix`* as the URL prefix of the application. The memory pointed to by *`p_prefix`* should have a lifetime greater than or equal to the lifetime of the `HttpdSingleSessionApplication` object.

Writing Single-Session Application Specifications

When using the specgen tool the `app` package can be used to automatically generate the initialization machinery of an application. This machinery is in the form of a generated function that initializes the application handler object.

For example, assuming the following specification fragment:

```
application myApp : single
{
  menu       mnuMain;            ❶
  prefix     "/app";
  resources  resEnglish;         ❷
  string     resource "US-en";
};
```

❶❷ It is assumed these objects are defined elsewhere.

the application can be instantiated with the following code fragment in the startup of the system:

```
Httpd            *p_webserver = …
HttpdFileSystem  *p_fs        = …
HttpdAppHandler  *p_handler;

int rc = myApp(p_handler, p_fs);  ❶
if (rc != 0)
{
  printf("Error starting application: %d\n", rc);
  return;
}

p_webserver->Install(p_handler);  ❷
```

❶ The `myApp` routine is generated as a result of the `myApp` specification. The generated routine always takes two arguments. The first is a reference to the pointer that is to receive the handler address and the second is the `HttpdFileSystem` that is used to initialize the resources.

Note

The generated function should only be called once during the startup of the system.

The `menu` statement is optional and if not specified then no desktop widget will be created in the session. The `string` statement can also take a different form to specify an instance of `HttpdStringTable` that is declared with the `stringtable` specification:

```
string provider  strProvider;
```

HttpdSessionApplication Reference

Introduction

A `HttpdSessionApplication` uses the `HttpdSessionManager` to support multi-session applications. In this configuration multiple users can use an application at the same time without interfering with one another. Multi-session applications are also localizable; meaning that different sessions can use different resources and string providers simultaneously.

`HttpdSessionApplication` is an abstract class and does not institute a policy for how session identification is passed. Seminole provides two subclasses that implement a passing policy. The `HttpdFormSessionApplication` class uses hidden form variables to pass the session identifier while the `HttpdCookieSessionApplication` class uses cookies.

Each approach has its advantages and disadvantages. Passing the session identifier in hidden form values does not require cookies (which some users find distasteful) although the session can be easily lost if the user navigates outside the application. Cookies are more robust and are the preferred method of keeping state with HTTP. Although using cookies means that user can only log in once to a particular application with a particular browser.

Session objects in multi-session applications must be an instance of `HttpdSessionApplication::Session` or one of its subclasses. This class combines the `HttpdAppSession` with the `HttpdSessionObject` class.

Because of the added complexity of session construction and deletion, `HttpdSessionApplication` objects are configured using an instance of a stand-alone structure, `HttpdSessionApplication`::Config. The Config structure includes a pointer to a “logon procedure.” This function pointer is called when a new session is to be created. This allows the application program to perform security checks or other operations at login time.

Thankfully this complexity is normally hidden when using the `app` package with the specgen tool.

Using `HttpdSessionApplication` handlers requires that several static HTML pages for handling user logon. The URI's for these pages are stored in the Config structure.

Public Methods

HttpdSessionApplication

`HttpdSessionApplication::HttpdSessionApplication` `(const Config *`*`p_config`*`);`

This constructs a `HttpdSessionApplication` object. The *`p_config`* parameter points to a configuration structure which must have an equal or greater lifetime than the `HttpdSessionApplication` object.

Note

This method only performs a partial initialization. In addition the `HttpdSessionApplication::Create` method must be called before the object can be used as a handler.

Create

`int` **`HttpdSessionApplication::Create`** `(size_t` *`max_sessions`*`);`

This method completes the initialization of the `HttpdSessionApplication` object. The *`max_sessions`* parameter determines the maximum number of session objects that can be tracked by the application.

On failure an error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned; otherwise `0` is returned.

Insert

`int` **`HttpdSessionApplication::Insert`** `(HttpdAppEvent &`*`event`*`, Session *`*`p_session`*`);`

This static method inserts the session object identified by *`p_session`* in the application. Once inserted the session object is managed by the application automatically. The *`event`* parameter is the event object that resulted in the sessions creation. This method is typically called from a logon procedure after successful creation of the session.

On failure an error code from Table 4.1, "OS Abstraction Layer Error Codes" is returned; otherwise `0` is returned.

The Config Structure

The configuration structure is the set of parameters that initialize multi-session applications. The declaration is as follows:

```
struct Config
{
  const char      *mpLogonPageUrl;
  const char      *mpLogoffPageUrl;
  const char      *mpLogonFailedUrl;
  const char      *mpLogonExpiredUrl;
  Session       *(*mpLogonProc)(HttpdAppEvent &event, bool &redirect);

#if defined(HTTPD_INC_BACKGROUND_SESSION_PURGE)
  int              mMaxSessionAge;
  size_t           mScrubbingBatchSize;
  unsigned long    mCycleTime;
#endif
};
```

Configuration Fields

`mpLogonPageUrl`	This is the URL or absolute path of the HTML document that should be presented to a user who is not logged in.
`mpLogoffPageUrl`	The user is redirected to this URL or absolute path when they request to log out of an application.
`mpLogoffPageUrl`	The user is redirected to this URL or absolute path when they request to log out of an application.
`mpLogonFailedUrl`	The user is redirected to this URL or absolute path when the request to login is denied by the logon procedure.
`mpLogonFailedUrl`	The user is redirected to this URL or absolute path when the request to login is denied by the logon procedure.
`mpLogonExpiredUrl`	The user is redirected to this URL or absolute path when the user was logged in but their session had since expired.
`mpLogonProc`	This is the address of the logon procedure for the application.
`mMaxSessionAge`	If background session scrubbing is enabled then this is parameter is used as the argument when the `HttpdSessionManager::MaxSessionAge` method is called.
`mScrubbingBatchSize`	If background session scrubbing is enabled then this is parameter is used as the argument when the `HttpdSessionManager::ScrubbingBatchSize` method is called.
`mCycleTime`	If background session scrubbing is enabled then this is parameter is used as the argument when the `HttpdSessionManager::CycleTime` method is called.

If desired all of the various logon URI's members can be pointed to the same HTML document if no details feedback about logon should be given.

The Logon Procedure

The logon procedure is responsible for examining any parameters that may have been submitted with from the various logon forms (pointed to by the Config structure). If the parameters, such as user-name and password, are correct then a session object and its corresponding desktop widget should be created.

Once created the logon procedure should call `HttpdSessionApplication::Insert` to insert the newly created session into the applications' session manager. If successful a pointer to the newly created session object should be returned.

If any kind of fatal error is encountered then the *`redirect`* should be set to `false`, a fatal response should be sent, and NULL should be returned:

```
if (fatal_error)
{
  redirect = false;
  event.mpRequest->Respond(HTTPD_RESP_SRV_ERROR);
  return (NULL);
}
```

For the case of an incorrect logon (which is not a fatal error) the *`redirect`* should be left alone and NULL should be returned. In this scenario no response should be sent by the logon procedure. The logic in `HttpdSessionApplication` will redirect the user in this case to the URI in the `mpLogonFailedUrl` field of the Config structure.

Writing Multi-Session Application Specifications

The specgen tool with the `app` package is the preferred way to develop a multi-session application. There are two basic approaches to defining a multi-session application. The first approach, non-localized relies on a single resource map and a single string provider. The second and more complicated approach allows multiple resource maps and string providers to be used with a set being chosen at logon time.

The non-localized approach is as follows:

```
application myApp : session
{
  menu           mnuMain;
  prefix         "/app";
  resources      EnglishResources;
  string         resource "US-en";
  type           cookie; ❶
  max_users      256;
  new_session    CreateAppSession; ❷
  logon <-
  {
    HttpdCgiParameter *p_param = event.mParameters.Find("username");
    if (p_param == NULL)
      return (NULL);

    const char *p_username = p_param->mPair.mpValue;

    p_param = event.mParameters.Find("password");
    if (p_param == NULL)
      return (NULL);
    const char *p_password = p_param->mPair.mpValue;

    if ((strcmp(p_password, "password") != 0)
    ||  (strcmp(p_username, "user") != 0))
      return (NULL);

    return (CreateAppSession(event, redirect)); ❸
  };

  logon_url         "/login/login.html";
  logoff_url        "/login/login.html";
  logon_failed_url  "/login/login_failed.html";
  logon_expired_url "/login/login_expired.html";

  scrubbing ❹
  {
    max_age     86400;  # Seconds
```

```
        batch_size  8;      # Sessions
        cycle_time  720;    # Seconds
      };
   };
```

❶ This statement determines the way in which the session identifier is passed. It can be `cookie` to use cookies or `form` to use hidden form fields.

❷ If specified, the `new_session` directive requests that a function be created to create the session. This is strictly a convenience as this could be done manually. Its main purpose is to simplify the logon code fragment in the `logon` directive.

❸ To create and insert the session object the logon code can simply call the helper routine created with `new_session`.

❹ This optional statement sets the scrubbing parameters of the session. If not specified the values shown here are used as defaults.

Note

The `string`, `menu`, and `resources` directives behave like their single-session counterparts.

As with single-session applications this results in a function called `myApp` that performs the initialization of the application. The code for starting the application using this function is identical to the single-session version.

The localized version shares most of the directives of the non-localized version except the `string` and `resources` directives are replaced with a list of the various locales the application supports:

```
# Replace 'string' and 'resources' for multiple locales:
locales
{
#   Locale            Resources           String bundle resource name

    "US-english" :    EnglishResources,   "US-en";
    "GB-english" :    EnglishResources,   "GB-en"; ❶
    "German"     :    GermanResources,    "DE-de";
};
```

❶ Resources do not have to be distinct (and neither do string bundles). Here we can see both British and US locales share `EnglishResources` but use different string bundles.

When the `locales` keyword is used the function generated by the `new_session` directive will look at the CGI parameters in the event for a value named `locale`. The locale with the selected name will be used.

Menus

Introduction

A menu is a collection of buttons that invoke an event to a particular widget. Menus are managed with three different objects. An `HttpdMenuItem` describes a particular menu choice. An `HttpdMenu` instance is usually owned by a widget that wishes to present a menu. When painting, an instance of `HttpdMenuSymbols` is used to add template symbols for the particular menu.

Menu definitions can be build automatically using the specgen tool and the `menus` package. They can also be constructed by hand if necessary.

`HttpdMenu` Reference

Public Methods

`HttpdMenu`

`HttpdMenu::HttpdMenu` `(const HttpdMenuItem *`*`p_items`*`, size_t `*`count`*`);`

The constructor initializes a menu object to contain a the items described by *`p_items`*. The *`count`* parameter specifies how many `HttpdMenuItem` elements *`p_items`* points to.

Note

After construction the `HttpdMenu` object must be initialized further with the `Create` method. The call to `Create` is often done in the owning widgets constructor.

`Create`

`int `**`HttpdMenu::Create`**` (void);`

This method performs final initialization of the menu object. By default all of the options are marked as enabled.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Dispatch`

`int `**`HttpdMenu::Dispatch`**` (HttpdAppEvent &`*`event`*`);`

Widgets that own menus should pass their events to this method. If the event is the result of a menu action this method will call the appropriate call-back for the menu item selected.

If *`event`* is a menu event the return value is the return value from the menu action call-back. This is typically `0` on success or system dependent error value (see Table 4.1, “OS Abstraction Layer Error Codes”). If the event is not for the widgets menu then `HTTPD_TEMPLATE_NOT_HANDLED` is returned.

Note

Unlike widgets, menus have no unique identifier. Instead menus depend on the unique identifiers of the containing widget to be uniquely identified on the client. Therefore only one menu can be managed by a widget at a time without extra logic.

`Enabled`

`HttpdBitSet & `**`HttpdMenu::Enabled`**` (void);`

Menu items can be enabled or disabled by manipulating the elements in the returned `HttpdBitSet`. If the index is present in the set then the item is considered enabled. By default all menu items are enabled.

`Count`

`unsigned int `**`HttpdMenu::Count`**` (void);`

This method returns the number of items in the menu.

FindItem

```
unsigned int HttpdMenu::FindItem (HttpdStringId item);
```

This method finds a menu item based upon label. If found the zero-based index of the item is returned. If the item could not be found then the return value is equal to the return value of the `Count` method.

`HttpdMenuItem` Reference

Public Data Members

mItem

```
HttpdStringId mItem
```

This field is the string identifier used to paint the label for this entry.

mpAction

```
int (*mpAction)
 (
   size_t index,
   HttpdAppEvent &event
 )
```

If this field is not NULL then the function it points to is called when the menu item is invoked. The *`event`* parameter is the event record for the menu selection. The *`index`* parameter is the index of the `HttpdMenuItem` record in the array that defines the menu.

If this field is NULL then this entry is considered a spacer or category for organizing selections.

`HttpdMenuSymbols` Reference

`HttpdMenuSymbols` is provided to paint a menu during template-based widget painting. Menus can be painted in a variety of ways. Frequently button bars (such as the buttons at the bottom of the standard dialog) are menus in disguise.

Template Directives

The `HttpdMenuSymbols` symbol table adds a single looping directive for painting a menu, `menu-items`. Within this loop several additional directives are available pertaining to the current menu item.

Table 10.7. Directives available during `menu-items`

Directive	Type	Description
`enabled`	Conditional	This condition is true if the current menu item has not been marked as disabled.

Directive	Type	Description
`heading`	Conditional	This condition identifies the current menu item as being a heading entry with no associated action (`mpAction` is NULL).
`label`	Evaluation	This directive evaluates to the string label assigned to the current entry.
`link`	Evaluation	This directive evaluates to the tag that invokes this particular menu event. This tag should be present as a CGI parameter during the submission of the request to the server.

Public Methods

`HttpdMenuSymbols`

`HttpdMenuSymbols::HttpdMenuSymbols` `(HttpdMenu *`*`p_menu`*`);`

The symbol table is associated with the menu definition of *`p_menu`* as well as implicitly with the current widget being painted. This symbol table should not be used unless an instance of `HttpdAppTemplateProcessor` is performing the template execution.

After construction the `HttpdMenuSymbols` object should be installed in the current symbol scope with an instance of `HttpdTemplateScope`.

Writing Menu Specifications

Although the table of `HttpdMenuItem` can be built by hand the `menus` package for the specgen tool processes a elegant syntax for defining a menu.

The `menus` package adds a new directive called `menu` for defining a menu. The `menu` directive is followed by an identifier that gives a symbolic name for the menu definition. This symbolic name is used to declare the array of `HttpdMenuItem` structures and can be used to initialize `HttpdMenu` objects.

For example, assuming a menu definition with a symbolic name of `main_menu` a menu object called `menu_object` can be declared as follows:

```
HttpdMenu menu_object(main_menu, HTTPD_NUMELEM(main_menu));
```

Note

Notice that the HTTPD_NUMELEM macro is used to determine the number of items in the `main_menu` array. This is guaranteed to work because **specgen** always declares the array with the number of elements (even in `extern` declarations).

In addition to declaring an array of structures, the symbolic name of the menu may be useful to other **specgen** packages.

Following the symbolic name is the actual message definition block. There are two directives. The `option` directive defines a selectable option complete with associated code. The `heading` directive defines a menu item with no associated call-back (`mpAction` is NULL).

For example, assuming the following definition for a menu named `main_menu`:

```
menu main_menu
{
  heading MSG_SYSTEM;
  option MSG_REBOOT <- reboot();
  option MSG_SAVECONFIG: SaveConfiguration;

  heading MSG_STATUS;
  option MSG_CPULOAD <- do_cpu_load(event);
  option MSG_ADDRESSING: DisplayAddressing;
};
```

The `MSG_SAVECONFIG` and `MSG_ADDRESSING` options are referencing external routines that are prototyped to match the type of the `mpAction` member of `HttpdMenuItem`. The `MSG_REBOOT` and `MSG_CPULOAD` options are wrapped in anonymous functions prototyped with *`index`* and *`event`* parameters.

It is also possible to attach a menu option directly to a dialog box (either modal or non-modal). This still requires a code fragment because a dialog box needs an object to manipulate although the code fragment is much simpler.

```
menu main_menu
{
  heading MSG_CONFIGURATION;
  option MSG_T1BOARD  dialog(dlgT1Board, data) <-
  { data = &T1Parameters; };
  option MSG_LANBOARD dialog(dlgT1Board, ethdata) <-
  {
    ethdata = GetLANParameters();
    if (ethdata == NULL)
      return (HttpdOpSys::ERR_OUTOFMEM);
  };
};
```

The `dialog` statement associates a particular dialog box with the named variable. The attached code should assign a valid pointer for the dialog structure to the named variable or return an error code.

Alternatively, the `dialog` keyword can be replaced with the `modal_dialog` keyword to invoke a modal dialog box.

`HttpdWidgetDesktop` Reference

Introduction

The `HttpdWidgetDesktop` widget is used to manage a typical application view. Although it is not a requirement that a desktop widget is used to manage the application some widget types do require it to be present. Under normal circumstances a desktop widget should always be created. Only under rare circumstances (such as extreme code size limitations) should the desktop be avoided.

The desktop widget is a container widget and is always a child of the root widget. The most prominent features of the desktop widget are the menu bar and the status area. The menu bar is an instance of `HttpdMenu` and is used to provide a navigational menu for the overall structure of the application. The status area is a used to display informational messages; often as an indication of success or failure when performing an action.

An additional feature of the desktop widget is that it works in conjunction with the `HttpdAppModal` class to force the user to perform a particular action.

Template Directives

Table 10.8. Evaluation Directives

`top`	This directive paints the current or “top” widget. This directive should not be invoked unless the `has-top` conditional is true.
`status`	Displays the current status message. This directive should not be invoked unless the `has-status` conditional is true.
`clear-status`	Clears the status message if it is set.

Table 10.9. Conditional Directives

`menu-hidden`	This conditional determines if the menu should be hidden.
`has-top`	This conditional is true if there is a current or “top” widget.
`has-status`	This conditional is true if a status message is pending.

Public Methods

HttpdWidgetDesktop

`HttpdWidgetDesktop::HttpdWidgetDesktop` `(const char *`*`p_local_id`*`, HttpdWidgetContainer *`*`p_parent`*`, const HttpdMenuItem *`*`p_items`*`, size_t` *`count`*`, int &`*`rc`*`);`

Construct a desktop widget. The *`p_local_id`*, *`p_parent`*, and *`rc`* arguments function identically to the corresponding arguments in the `HttpdWidget` constructor.

The *`p_items`* and *`count`* argument define the desktop menu.

MenuHidden

`unsigned int &` **`HttpdWidgetDesktop::MenuHidden`** `(void);`

This method returns a reference to an internal counter that determines if the application menu is hidden. The returned reference should never be directly assigned to. Instead the returned reference should be either incremented and decremented. It is important that the number of increments match the number of decrements.

The manipulation of the menu hidden counter is automatically handled by the `HttpdAppModal` class.

Menu

`HttpdMenu &` **`HttpdWidgetDesktop::Menu`** `(void);`

This method returns a reference to the menu object of the desktop widget. The most common use of this is to enable or disable specific items in the desktop menu depending on system state.

Top

`HttpdWidget *&` **`HttpdWidgetDesktop::Top`** `(void);`

The default behavior of the desktop is to consider one widget as the current widget the user is interacting with. This method returns a reference to the variable that identifies the current widget.

If the current widget is set to NULL then no widget is considered current.

Status

`void` **`HttpdWidgetDesktop::Status`** `(HttpdStringId` *`message`*`);`

Set a status message to be displayed on the desktop on the next painting cycle. If a previous message was set to be displayed it is replaced with *`message`*.

Desktop

`HttpdWidgetDesktop *` **`HttpdWidgetDesktop::Desktop`** `(HttpdWidget *`*`p_widget`*`);`

This static method helps locate the desktop widget by given any valid widget (*`p_widget`*). If the desktop can not be found, NULL is returned. This method should only be called when employing the desktop widget.

CreateDesktop

`int` **`HttpdWidgetDesktop::CreateDesktop`** `(HttpdAppSession *`*`p_session`*`, const HttpdMenuItem *`*`p_menu`*`, size_t` *`menu_count`*`);`

This static method creates a standard desktop widget in the session specified by *`p_session`*. The *`p_menu`* and *`menu_count`* arguments define the application menu. The current widget is set to NULL.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdAppModal` Reference

Introduction

When using the `HttpdWidgetDesktop` widget it is often desirable to temporarily suspend the normal desktop navigation until a certain condition is met. Most commonly the suspension of desktop navigation is needed for the lifetime of a particular widget.

The `HttpdAppModal` object can be embedded in a widget to ensure that navigation is disabled for the life of that widget. This class provides no methods beyond a constructor and destructor. Simply initializing it in the constructor of a widget is sufficient to make a widget modal.

Public Methods

`HttpdAppModal`

`HttpdAppModal::HttpdAppModal` `(HttpdWidget *`*`p_widget`*`, int &`*`rc`*`);`

This constructor can be called in the initializer list of a widget with `this` passed in as the value for *`p_widget`* and *`rc`* being the same *`rc`* reference passed to the constructor of a widget.

Dialogs

Introduction

A dialog is a container widget that holds a collection of "controls." Controls are widgets that allow the user to manipulate information. A dialog is defined by a set of data structures called a dialog template. The dialog template is rather complex but the specgen tool builds the template structure automatically from a specification.

The information that a dialog box represents is defined by an associated structure where each control widget manipulates a particular field. Dialog widgets automatically manage the transfer of data between the structure and the control widgets.

In order for the dialog widget to manage the control widgets it relies on a call-back routine called a manager. Each field definition has a pointer to a manager procedure. The manager procedure is used to construct the widget and transfer values from the structure and validation. The manager procedure is flexible; taking a opcode value that identifies the requested action and a parameter list as a va_list.

Although any widget can be managed inside a dialog box most control widgets are subclasses of `HttpdWidgetField`. The `HttpdWidgetField` widget maintains an error state and provides a manager procedure that handles a few basic events.

A dialog template is defined by an instance of the `HttpdDialogTemplate` structure. This structure points to an array of `HttpdDialogField` that defines each field. Each `HttpdDialogField` structure can point to a "configuration structure" that is specific to the widget or manager of the field.

Data Types

`HttpdDialogTemplate` Public Data Members

`mpName`

```
        const char *mpName
```

This field identifies the local identifier of the dialog widget.

`mpLayout`

```
const char *mpLayout
```

This field identifies the resource used as a template to paint the dialog box.

mpFields

```
const HttpdDialogField *mpFields
```

This field points to an array of field descriptors that describe the parameters of the dialog.

mFieldCount

```
size_t mFieldCount
```

This field defines the number of elements in the `mpFields` array.

mpMenuItems

```
const HttpdMenuItem *mpMenuItems
```

Each dialog box has an associated menu. This field is a pointer to the menu item table.

mMenuCount

```
size_t mMenuCount
```

This field is the number of elements in the `mpMenuItems` array.

mpInit

```
int (*mpInit)
 (
   HttpdWidgetDialog *p_dialog
 )
```

This call-back is called when the dialog box has finished its basic initialization and created all of its control widgets. Returning a non-zero value from this callback will prevent the dialog from being constructed with a successful return code.

mpValidate

```
int (*mpValidate)
 (
   HttpdWidgetDialog *p_dialog
```

```
         )
```

This call-back is a final validation function for the dialog box. Although each field can be validated independently the job of this validator routine is to ensure that all of the field values make sense as a whole. This routine should return `HTTPD_TEMPLATE_FALSE_CASE` if the validation constraints are not met. Otherwise `0` should be returned. If no overall validation is required this field can be set to `HttpdWidgetDialog::NullProc`.

mpOnComplete

```
int (*mpOnComplete)
 (
   HttpdWidgetDialog *p_dialog
 )
```

If this field is not NULL the function it points to is called after the dialog structure is updated during the `HttpdWidgetDialog::Complete` method.

mpOnCancel

```
int (*mpOnCancel)
 (
   HttpdWidgetDialog *p_dialog
 )
```

If this field is not NULL the function it points to is called during the `HttpdWidgetDialog::Cancel` method.

mCompletedMsg

```
HttpdStringId mCompletedMsg
```

If the `HTTPD_DLG_SET_COMPLETION_MSG` flag is set in the `mFlags` field this message is set in the status field of the desktop widget when the `HttpdWidgetDialog::Complete` method is called.

mCancelledMsg

```
HttpdStringId mCancelledMsg
```

If the `HTTPD_DLG_SET_CANCELLED_MSG` flag is set in the `mFlags` field this message is set in the status field of the desktop widget when the `HttpdWidgetDialog::Cancel` method is called.

mFlags

```
unsigned char mFlags
```

This field contains a set of flags that effect the behavior of dialog box event processing. Several options can be set in this field.

Table 10.10. Dialog Template Flags

`HTTPD_DLG_SET_COMPLETION_MSG`	The string identified by `mCompletedMsg` should be set in the desktop status on dialog completion.
`HTTPD_DLG_SET_CANCELLED_MSG`	The string identified by `mCancelledMsg` should be set in the desktop status on dialog cancellation.
`HTTPD_DLG_NO_CONSTRAINT_IF_ERROR`	Do not call the function pointed to by `mpValidate` if any of the fields did not pass validation. Normally the validation function is called even if some fields are not valid.

`HttpdDialogField` Public Data Members

`mpName`

```
const char *mpName
```

This data member identifies the name of the field widget.

`mpTemplate`

```
const char *mpTemplate
```

This data member identifies the resource name used for painting the widget.

`mOffset`

```
size_t mOffset
```

This data member is the offset in the dialog structure of where the field data resides.

`mpManager`

```
HttpdFieldManager mpManager
```

This data member points to a call-back routine to manage basic operation of the field. Rather than rely on subclassing for each specialized field which is tedious and error prone (as well as a bloaty approach) the manager procedure acts as a simple adaptor to an underlying instance of a widget.

The typedef HttpdFieldManager is defined as a pointer to the manager with the following prototype:

```
typedef int (*HttpdFieldManager)
(
  const HttpdDialogField *p_field,
  HttpdWidget *p_widget,
  int action,
  va_list va
);
```

The manager procedure handles many different chores with the *action* identifying the request. Arguments are encapsulated in the *va* argument list.

Table 10.11. Field Manager Procedure Events

`HTTPD_DLG_CREATE_WIDGET`	This event is sent when the control widget is to be created. The *va* list contains two additional parameters. The first is a pointer to the dialog widget and has a type of HttpdWidgetContainer *. The second parameter is a pointer that holds the address of the newly created widget and has a type of HttpdWidget **. If success is returned the second parameter must be set to point to the newly created widget. In addition, the newly created widget must have the dialog container widget as its parent.
`HTTPD_DLG_INIT_WIDGET`	This event is sent when after all control widgets have been set to their initial values but before the template `mpInit` routine is called.
`HTTPD_DLG_SET_WIDGET`	This event instructs the control widget to update its state from the field stored in the dialog structure. The *va* list contains a single void * parameter that is the address of the field in the dialog structure.
`HTTPD_DLG_GET_WIDGET`	This event instructs the control widget to update the field in the dialog structure with the controls current value. The *va* list contains a single void * parameter that is the address of the field in the dialog structure.
`HTTPD_DLG_VALIDATE`	This event is sent during an update cycle of the dialog. If the control widget is performing validation it should use this event as an indication to examine its current value for invalid data. If the data is found not to be valid the state of the widget should be updated appropriately.
`HTTPD_DLG_IS_VALID`	This event is sent by the dialog widget to determine if the current data of the control is valid during the last `HTTPD_DLG_VALIDATE` operation. If the data is valid then `HTTPD_TEMPLATE_TRUE_CASE` should be returned. Otherwise if the last validation found erroneous data then `HTTPD_TEMPLATE_FALSE_CASE` should be returned.
`HTTPD_DLG_CLEAR_ERROR`	This event should clear any error determined by the previous `HTTPD_DLG_VALIDATE` operation.

	Subsequent `HTTPD_DLG_IS_VALID` requests should `HTTPD_TEMPLATE_TRUE_CASE` until the next validation.
`HTTPD_DLG_TEMPLATE_EVAL`	This operation is called when the template of a `HttpdWidgetField` (or one of its subclasses) encounters a `manager/` directive. The *`va`* argument list contains two additional parameters. The first, of type const char * is the string following the slash in the directive. The second parameter is a pointer to the `HttpdEvalCommand` object.
`HTTPD_DLG_USER`	This constant can be used as the base identifier for application-specific manager requests. The application framework never sends requests with this identifier or any values larger than `HTTPD_DLG_USER`.

mLabel

```
HttpdStringId mLabel
```

This data member identifies the string label assigned to the field.

mpConfig

```
const void * mpConfig
```

This data member is a pointer to a field-specific configuration structure. It is available for use by the manager procedure or the control widget. If no configuration structure is needed then this field should be set to NULL.

`HttpdWidgetDialog` Reference

`HttpdWidgetDialog` is a subclass of `HttpdWidgetContainer` and manages field widgets using the dialog template. A dialog widget has a menu and possesses the template directives for painting menus.

Public Methods

HttpdWidgetDialog

`HttpdWidgetDialog::HttpdWidgetDialog` `(HttpdWidgetContainer *`*`p_parent`*`, const HttpdDialogTemplate *`*`p_template`*`, void *`*`p_data`*`, int &`*`rc`*`);`

The constructor initializes a dialog widget. *`p_parent`* is the parent of the dialog widget; this is typically the desktop widget. The address of the dialog template must be passed in *`p_template`*. The *`p_data`* argument must be a pointer to an instance of the dialog structure. The lifetime of the dialog structure must be at least as long as the lifetime of the dialog widget. Failures during widget (or control widget) creation are identified in the *`rc`* argument.

Template

`const HttpdDialogTemplate *` **`HttpdWidgetDialog::Template`** `(void);`

This method returns a pointer to the template structure that defines this dialog widget. The returned pointer is never NULL.

Data

`void *& **HttpdWidgetDialog::Data** (void);`

This method returns a reference to the pointer to the dialog structure.

Modified

`bool **HttpdWidgetDialog::Modified** (void);`

This method queries the control widgets and determines if they contain modified values. Control widgets that can not report their modified status are ignored when tabulating the results.

The dialog template conditional `any-modified` evaluates to the same value as this method.

ControlCount

`size_t **HttpdWidgetDialog::ControlCount** (void);`

This method returns the number of controls in the dialog.

Control

`HttpdWidget * **HttpdWidgetDialog::Control** (size_t *index*);`

This method returns a pointer to the widget that implements the control for the field identified by *`index`*.

Field

`const HttpdDialogField * **HttpdWidgetDialog::Field** (size_t *index*);`

This method returns a pointer to the field descriptor for the field identified by *`index`*.

ValidateFields

`int **HttpdWidgetDialog::ValidateFields** (void);`

This method calls the field-specific validators for all fields. The `mpValidate` call-back in the dialog template is not called. If an error is encountered during the validation no further processing is done and the error code is returned. Upon success a value of `0` is returned.

AreFieldsValid

`int **HttpdWidgetDialog::AreFieldsValid** (bool &*valid*);`

This method queries the manager procedure of each field to determine if any fields have a pending error. Upon success, *`valid`* is set to reflect the state of the fields and `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

ValidateAll

`int **HttpdWidgetDialog::ValidateAll** (bool &*valid*);`

This method first applies the per-field validators to ensure that each field is acceptable. Afterwards, the template validator function is called to ensure that the relationships between the fields are not violating any constraints. Upon success, the *`valid`* is set to `true` if all data is valid or `false` if some constraints have been violated. Otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

MoveValues

`int` **`HttpdWidgetDialog::MoveValues`** `(int` *`action`*`, void *`*`p_data`*`);`

This method moves values between the dialog structure pointed to by *`p_data`* and the dialog controls. If the *`action`* argument is `HTTPD_DLG_GET_WIDGET` then the values from the dialog controls are copied to the dialog structure. If the *`action`* argument is `HTTPD_DLG_SET_EVENT` then the values from the dialog structure are propagated to the controls.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Cancel

`int` **`HttpdWidgetDialog::Cancel`** `(void);`

This method cancels any pending changes in the dialog and marks the dialog widget as defunct: after processing the current event the dialog widget is destroyed. No changes are made to the dialog structure.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Complete

`int` **`HttpdWidgetDialog::Complete`** `(void);`

This method attempts to apply any pending changes in the dialog to the dialog structure if the data passes the validation constraints. If the data is valid then the widget is marked as defunct and the dialog structure is updated. If the data is not valid then the dialog remains active.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

ManageField

`int` **`HttpdWidgetDialog::ManageField`** `(const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int` *`action`*`, …);`

This static method calls the manager procedure for the field specified by *`p_field`*. The *`p_widget`* argument should be a pointer to the control widget.

ManageField

`int` **`HttpdWidgetDialog::ManageField`** `(HttpdFieldManager` *`p_manager`*`, const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int` *`action`*`, …);`

This static method calls the specified manager procedure with the provided arguments. This method is normally used to call a different manager method from a manager that only hanldes a few specific events.

Create

`int` **`HttpdWidgetDialog::Create`** `(HttpdAppEvent &`*`event`*`, const HttpdDialogTemplate *`*`p_template`*`, void *`*`p_data`*`);`

This static method is a convenience routine for creating dialog boxes in response to events. Typically these events are menu events although any valid event structure will do.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, "OS Abstraction Layer Error Codes").

`InitOptionalField`

`void` **`HttpdWidgetDialog::InitOptionalField`** `(HttpdWidget *`*`p_widget`*`, bool` *`present`*`);`

This static method should be used when a field is optional and should only be displayed when certain conditions are met. A field manager routine should call this method during the `HTTPD_DLG_INIT_WIDGET` event. The *`present`* should be true if the field is required during the initial display of the dialog.

`ShowOptionalField`

`bool` **`HttpdWidgetDialog::ShowOptionalField`** `(HttpdWidget *`*`p_widget`*`, bool` *`present`*`, int &`*`rc`*`);`

This static method should be used when a field is optional and should only be displayed when certain conditions are met. A field manager routine should call this method during the `HTTPD_DLG_VALIDATE` event. The *`present`* should be true if the field is required for the current state of the dialog.

If this method returns true then the event handler should not proceed with any further validation and the value in *`rc`* should be returned from the field manager. If the return value is false then validation should proceed as normal.

`HttpdWidgetField` Reference

The `HttpdWidgetField` class is the base class for control widgets that support the concept of "validation." Validation means that when a particular field takes on an unacceptable value the dialog box will not update the dialog structure.

Template Directives

Table 10.12. `HttpdWidgetField` Template Directives

Directive	Type	Description
`data-tag`	Evaluation	This directive evaluates to a unique tag name for the data value of this field.
`error`	Evaluation	This directive evaluates to the current error message if any is present.
`label`	Evaluation	Each field has an associated label string, defined in the dialog template, to identify it. This directive evaluates to that label string.
`manager/`*`xxx`*	Evaluation	This directive calls the fields manager procedure with a

Directive	Type	Description
		`HTTPD_DLG_TEMPLATE_EVAL` request.
`has-error`	Conditional	If the field is in an error state this conditional directive evaluates to true.
`is-modified`	Conditional	If the field has been modified since its creation then this directive evaluates to true.

Public Methods

`HttpdWidgetField`

`HttpdWidgetField::HttpdWidgetField (HttpdWidgetContainer *p_parent, const HttpdDialogField *p_field, int &rc);`

This constructor initializes the field widget. The *`p_parent`* parameter is a pointer to the dialog widget. A pointer to the field descriptor should be passed in as *`p_field`*. The *`rc`* is the error code and has the same semantics as the *`rc`* argument in `HttpdWidget`'s constructor.

`SetError` (string version)

`void HttpdWidgetField::SetError (char *p_error);`

This method places the field widget into an “error state” with an error message of *`p_error`*. The *`p_error`* parameter must point to a string in storage obtained from HttpdOpSys::Malloc. Once given to this method the string is owned by the widget and should not be freed by the calling code.

`SetError` (localized version)

`void HttpdWidgetField::SetError (HttpdStringId error_message);`

This method places the field widget into an “error state” with an error message of *`error_message`*.

`ClearError`

`void HttpdWidgetField::ClearError (void);`

This method removes the widget from an error state.

`HasError`

`bool HttpdWidgetField::HasError (void);`

This method returns true if this widget is in an error state. Otherwise, false is returned.

`Manager`

`int HttpdWidgetField::Manager (const HttpdDialogField *p_field, HttpdWidget *p_widget, int action, va_list va);`

This static method is a basic implementation of a field manager procedure. It handles the `HTTPD_DLG_TEMPLATE_EVAL`, `HTTPD_DLG_IS_VALID`, and `HTTPD_DLG_CLEAR_ERROR` requests. Subclasses of `HttpdWidgetField` should call this static method as the default case in any manager procedures they define.

`HttpdWidgetScalar` Reference

A `HttpdWidgetScalar` is a control widget that can take on a string value. The definition of string value in this context is purposefully broad. Numeric values are also considered strings and can be handled by a `HttpdWidgetScalar`. The `HttpdWidgetScalar` class is a subclass of `HttpdWidgetField` and therefore can be validated.

Template Directives

Templates for `HttpdWidgetScalar` can also take advantage of the directives provided by its base class, `HttpdWidgetField`.

Table 10.13. `HttpdWidgetScalar` Template Directives

Directive	Type	Description
`value`	Evaluation	This directive evaluates to the current value of the widget. The value is HTML-escaped.
`have-data`	Conditional	This directive is true if the widget has a value.

Public Methods

`HttpdWidgetScalar`

`HttpdWidgetScalar::HttpdWidgetScalar (HttpdWidgetContainer *p_parent, const HttpdDialogField *p_field, int &rc);`

This constructor initializes the scalar widget. The *`p_parent`* parameter is a pointer to the dialog widget. A pointer to the field descriptor should be passed in as *`p_field`*. The *`rc`* is the error code and has the same semantics as the *`rc`* argument in `HttpdWidget`'s constructor.

GetValue

`const char * HttpdWidgetScalar::GetValue (void);`

This method returns a pointer to the value of the widget. If the widget has no value then NULL is returned.

SetValue

`int HttpdWidgetScalar::SetValue (const char *p_value);`

This method sets the current value of the widget to *`p_value`*. NULL can be passed in for *`p_value`* to indicate that the widget has no value.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Manager`

`int HttpdWidgetScalar::Manager (const HttpdDialogField *p_field, HttpdWidget *p_widget, int action, va_list va);`

This static method is a skeletal manager procedure for scalar fields. It handles the `HTTPD_DLG_CREATE_WIDGET` request in addition to all of the requests handled by the `HttpdWidgetField::Manager` procedure.

`HttpdWidgetOption` Reference

A `HttpdWidgetOption` is a control widget that selects between a finite set of options. This widget is typically rendered in HTML as a pulldown menu or a series of radio buttons. The `HttpdWidgetOption` class is a subclass of `HttpdWidgetField` and therefore can be validated.

This widget requires a configuration structure pointed to by the `mpConfig` member of the field descriptor. The configuration structure, `HttpdWidgetOption::Options` is defined as follows:

```
struct Options
{
  const HttpdStringId   *mpLabels;
  size_t                 mCount;
};
```

The `mpLabels` data member of the configuration structure should point to an array of string identifiers enumerating each of the possible choices for the option. The `mCount` data member is the number of elements in the `mpLabels` array.

Template Directives

Templates for `HttpdWidgetOption` can also take advantage of the directives provided by its base class, `HttpdWidgetField`.

Table 10.14. `HttpdWidgetOption` Template Directives

Directive	Type	Description
`option/`*`nnn`*	Evaluation	This directive evaluates to the label of a particular identifier. The component following the slash (*`nnn`*) is an integral constant that identifies the options index.
`option-id`	Evaluation	This directive evaluates to the current option index when looping over all of the options. This directive should only be evaluated when inside an `options` loop.
`option-label`	Evaluation	This directive evaluates to the current option label when looping over all of the options. This directive should only be evaluated when inside an `options` loop.
`is-current-selection`	Conditional	This directive is true if the current option is the selected item when looping over all of the options. This directive should

Directive	Type	Description
		only be evaluated when inside an `options` loop.
`have-current-selection`	Conditional	This directive is true if the widget has a currently selected object.
`options`	Loop	This directive loops over all of the possible options.

Public Methods

`HttpdWidgetOption`

`**HttpdWidgetOption::HttpdWidgetOption** (HttpdWidgetContainer *`*`p_parent`*`, const HttpdDialogField *`*`p_field`*`, const Option *`*`p_options`*`, int &`*`rc`*`);`

This constructor initializes the option widget. The *`p_parent`* parameter is a pointer to the dialog widget. A pointer to the field descriptor should be passed in as *`p_field`*. The options list (which is normally stored in the field descriptor) is passed in via *`p_options`*. The *`rc`* is the error code and has the same semantics as the *`rc`* argument in `HttpdWidget`'s constructor.

GetCurSelection

`int` **`HttpdWidgetOption::GetCurSelection`** `(void);`

This method returns a the index of the current selection. If the value is currently selected then -1 is returned.

SetCurSelection

`void` **`HttpdWidgetOption::SetCurSelection`** `(int` *`index`*`);`

This method sets the current selection of the widget to the option identified by the *`index`* parameter. If *`index`* is set to `-1` no item is considered selected.

`Manager`

`int` **`HttpdWidgetOption::Manager`** `(const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int` *`action`*`, va_list` *`va`*`);`

This static method is a skeletal manager procedure for option selection fields. It handles the `HTTPD_DLG_CREATE_WIDGET` request in addition to all of the requests handled by the `HttpdWidgetField::Manager` procedure.

`HttpdWidgetBoolean` Reference

A `HttpdWidgetBoolean` is a control widget that is either on or off. This widget is typically rendered in HTML as a checkbox. The `HttpdWidgetBoolean` class is a subclass of `HttpdWidgetField` and therefore can be validated.

Template Directives

Templates for `HttpdWidgetBoolean` can also take advantage of the directives provided by its base class, `HttpdWidgetField`.

Table 10.15. `HttpdWidgetBoolean` Template Directives

Directive	Type	Description
`presence-key`	Evaluation	This directive evaluates to a widget tag for identifying if this widget value was even present in the returned set of CGI values. The need for this extra field is due to the fact that when a checkbox HTML object is no selected (but present) no value is returned. Templates should include a `hidden` input field with a non-empty string using this key name.
`is-selected`	Conditional	This directive is true if the state of the widget is true.

Public Methods

`HttpdWidgetBoolean`

`HttpdWidgetBoolean::HttpdWidgetBoolean (HttpdWidgetContainer *p_parent, const HttpdDialogField *p_field, int &rc);`

This constructor initializes the boolean widget. The *`p_parent`* parameter is a pointer to the dialog widget. A pointer to the field descriptor should be passed in as *`p_field`*. The *`rc`* is the error code and has the same semantics as the *`rc`* argument in `HttpdWidget`'s constructor.

GetCurState

`bool HttpdWidgetBoolean::GetCurState (void);`

This method returns a the current state of the widget.

SetCurState

`void HttpdWidgetBoolean::SetCurState (bool state);`

This method sets the current state of the widget to the *`state`*.

`Manager`

`int HttpdWidgetBoolean::Manager (const HttpdDialogField *p_field, HttpdWidget *p_widget, int action, va_list va);`

This static method is a skeletal manager procedure for boolean fields. It handles the `HTTPD_DLG_CREATE_WIDGET` request in addition to all of the requests handled by the `HttpdWidgetField::Manager` procedure.

`HttpdWidgetMulti` Reference

A `HttpdWidgetMulti` is similar to a `HttpdWidgetScalar` widget but is designed for multi-part strings. An example of a multi-part string is an IPv4 address in dotted decimal notation. The `HttpdWidgetMulti` widget consists of one or more named scalar values. Typically these widgets are used with specialized templates to setup a rigid layout for the field. The `HttpdWidgetScalar` class is a subclass of `HttpdWidgetField` and therefore can be validated.

This widget requires a configuration structure pointed to by the `mpConfig` member of the field descriptor. The configuration structure, `HttpdWidgetMulti::Options` is defined as follows:

```
struct Options
{
  const char   *const *mpFields;
  size_t               mCount;
};
```

The `mpFields` data member of the configuration structure should point to an array of strings naming each of the components of the multi-field. The `mCount` data member is the number of elements in the `mpFields` array.

Template Directives

Templates for `HttpdWidgetMulti` can also take advantage of the directives provided by its base class, `HttpdWidgetField`.

Table 10.16. `HttpdWidgetMulti` Template Directives

Directive	Type	Description
`value/`*`index`*	Evaluation	This directive evaluates to the value of a particular field of the widget. The *`index`* string can either be a field index preceded by an at sign (@) or the name of a field. The value is HTML-escaped.
`field-tag/`*`index`*	Evaluation	This directive evaluates to a unique name for this field that should be used for naming the form elements. As with the `value` directive, the *`index`* string can either be a field index preceded by an at sign (@) or the name of a field.
`have-value/`*`index`*	Conditional	This directive evaluates true if the field has a value.
`value-equals/`*`index`*	Conditional	This directive determines if the value of the specified field is equal to the attribute of the `value` attribute. Alternatively, the value to compare against can be specified as a URI-escaped string in the attribute `escaped`.
`have-field/`*`index`*	Conditional	This directive determines if the specified field is valid.

Public Methods

`HttpdWidgetMulti`

`HttpdWidgetMulti::HttpdWidgetMulti (HttpdWidgetContainer *p_parent, const HttpdDialogField *p_field, const Options *p_options, int &rc);`

This constructor initializes the multi widget. The `p_parent` parameter is a pointer to the dialog widget. A pointer to the field descriptor should be passed in as `p_field`. The options, normally stored in the field descriptor, should be passed in as `p_options`. The `rc` is the error code and has the same semantics as the `rc` argument in `HttpdWidget`'s constructor.

Index

`bool HttpdWidgetMulti::Index (const char *p_label, size_t &index);`

This method obtains the index of the field identified by the label `p_label`. If such a field exists then `index` is set to the index of that field and true is returned. If the field specified by `p_label` is not a member of the multi widget then false is returned.

GetValue

`const char * HttpdWidgetMulti::GetValue (size_t index);`

This method returns a pointer to the value of the widget field specified by `index`. If the widget has no value then NULL is returned.

SetValue

`int HttpdWidgetMulti::SetValue (size_t index, const char *p_value);`

This method sets the current value of the field identified by `index` to `p_value`. NULL can be passed in for `p_value` to indicate that the field has no value.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

`Manager`

`int HttpdWidgetMulti::Manager (const HttpdDialogField *p_field, HttpdWidget *p_widget, int action, va_list va);`

This static method is a skeletal manager procedure for multi fields. It handles the `HTTPD_DLG_CREATE_WIDGET` request in addition to all of the requests handled by the `HttpdWidgetField::Manager` procedure.

`HttpdFieldManagers` Reference

The `HttpdFieldManagers` structure contains static methods and other definitions that provide standard behavior for various dialog input fields when using standard data types.

Public Methods

`StoreUnsigned`

`void HttpdFieldManagers::StoreUnsigned (IntegerType type, void *p_dest, unsigned long value);`

This static method is used to update an unsigned integral value pointed to by *`p_dest`* with *`value`*. The *`type`* argument determines the type of the object pointed to by *`p_dest`*. The IntegerType type is an enumeration defined in the scope of `HttpdFieldManagers`. It can take on the values of `CharType`, `ShortType`, `IntType`, or `LongType`.

`StoreSigned`

`void` **`HttpdFieldManagers::StoreSigned`** `(IntegerType` *`type`*`, void *`*`p_dest`*`, long` *`value`*`);`

This static method is similar to `StoreUnsigned` except that the target values are assumed to be signed.

`FetchUnsigned`

`unsigned long` **`HttpdFieldManagers::FetchUnsigned`** `(IntegerType` *`type`*`, const void *`*`p_source`*`);`

This static method performs the reverse operation of `StoreUnsigned`. The unsigned integral value pointed to by *`p_source`* that is of type *`type`* is returned.

`FetchSigned`

`long` **`HttpdFieldManagers::FetchSigned`** `(IntegerType` *`type`*`, const void *`*`p_source`*`);`

This static method performs the reverse operation of `StoreSigned`. The signed integral value pointed to by *`p_source`* that is of type *`type`* is returned.

`EnumManager`

`int` **`HttpdFieldManagers::EnumManager`** `(const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int` *`action`*`, va_list` *`va`*`);`

This static method is a field manager procedure for enumerations that are explicitly of type int and enumerated started at `0` and increasing monotonically.

Important

Only widgets that are instances of `HttpdWidgetOption` should use this manager procedure.

`BoolManager`

`int` **`HttpdFieldManagers::BoolManager`** `(const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int` *`action`*`, va_list` *`va`*`);`

This static method is a field manager procedure for values that are explicitly of type bool.

Important

Only widgets that are instances of `HttpdWidgetBoolean` should use this manager procedure.

Public Structures

UnsignedInteger

The `UnsignedInteger` structure provides a static method, `Manager` that can be used to manage unsigned integral input fields based on the `HttpdWidgetScalar` input widget.

When using this manager procedure the `mpConfig` member of the field descriptor should point to an instance of the `UnsignedInteger` class.

Data member `mMinimum`

```
unsigned long mMinimum;
```

This member defines the minimum value this field can take on.

Data member `mMaximum`

```
unsigned long mMaximum;
```

This member defines the maximum value this field can take on.

Data member `mBelowMinimum`

```
HttpdStringId mBelowMinimum;
```

If the value of the field is below the minimum value the string identified by `mBelowMinimum` is used to indicate the error.

Data member `mAboveMaximum`

```
HttpdStringId mAboveMaximum;
```

If the value of the field is above the maximum value the string identified by `mAboveMaximum` is used to indicate the error.

Data member `mInvalid`

```
HttpdStringId mInvalid;
```

If the value of the field is not a valid number then the string identified by `mInvalid` is used to indicate the error.

Data member `mType`

```
IntegerType mType;
```

This member defines the size of the field. It can take on the values `CharType`, `ShortType`, `IntType`, or `LongType`.

Data member `mBase`

```
enum { Hex, Dec } mBase;
```

This member defines the base used for the string representation of the value.

SignedInteger

Like `UnsignedInteger`, the `SignedInteger` structure provides a static method, `Manager` that can be used to manage signed integral input fields based on the `HttpdWidgetScalar` input widget.

When using this manager procedure the `mpConfig` member of the field descriptor should point to an instance of the `SignedInteger` class.

Data member `mMinimum`

```
long mMinimum;
```

This member defines the minimum value this field can take on.

Data member `mMaximum`

```
long mMaximum;
```

This member defines the maximum value this field can take on.

Data member `mBelowMinimum`

```
HttpdStringId mBelowMinimum;
```

If the value of the field is below the minimum value the string identified by `mBelowMinimum` is used to indicate the error.

Data member `mAboveMaximum`

```
HttpdStringId mAboveMaximum;
```

If the value of the field is above the maximum value the string identified by `mAboveMaximum` is used to indicate the error.

Data member `mInvalid`

```
HttpdStringId mInvalid;
```

If the value of the field is not a valid number then the string identified by `mInvalid` is used to indicate the error.

Data member `mType`

```
        IntegerType mType;
```

This member defines the size of the field. It can take on the values `CharType`, `ShortType`, `IntType`, or `LongType`.

StaticStringBuffer

The `StaticStringBuffer` structure provides a static method, `Manager` that can be used to manage fixed-size zero-terminated string input fields based on the `HttpdWidgetScalar` input widget.

When using this manager procedure the `mpConfig` member of the field descriptor should point to an instance of the `StaticStringBuffer` class.

Data member `mBufferSize`

```
        size_t mBufferSize;
```

This member defines the size of the buffer that holds the data, including the zero-terminator byte.

Data member `mTooLong`

```
        HttpdStringId mTooLong;
```

If the string value entered by the user can not fit in the buffer this localized string is used to indicate the error.

TimeDateStamp

The `TimeDateStamp` structure provides a static method, `Manager` that can be used to manage input fields based for the `HttpdTimeStamp` class.

When using this manager procedure the `mpConfig` member of the field descriptor should point to an instance of the `TimeDateStamp` class.

Data member `mInvalid`

```
        HttpdStringId mInvalid;
```

This member defines the string that should be displayed when the field is in an error state.

Data member `mUseAmPm`

```
        bool mUseAmPm;
```

If true then an extra selection for AM or PM is provided for the time component.

Ipv4Address

The `Ipv4Address` structure provides a static method, `Manager` that can be used to manage input fields that are string-based IPv4 addresses.

When using this manager procedure the `mpConfig` member of the field descriptor should point to an instance of the `Ipv4Address` class.

Data member `mInvalid`

```
HttpdStringId mInvalid;
```

This member defines the string that should be displayed when the field is in an error state.

Data member `mpValidate`

```
bool (*mpValidate)(const HttpdUint8 *p_octets);
```

A pointer to a routine that validates the four octets of the address. Two built-in validator routines, `Host` and `Netmask` can be used to validate host addresses and netmasks, respectively.

Dialog Specifications

The specgen tool can be used to generate the `HttpdDialogTemplate` structure and associated field descriptors. The `dialogs` package defines a single directive, `dialog` that defines a dialog template. Within the `dialog` directive other directives define the structure of the dialog:

Table 10.17. Components of a `dialog` body

`struct`	If specified this names the dialog structure associated with this dialog. If this directive is omitted then the struct takes on the name *`Dialog`*Data, where *`Dialog`* is the name of the dialog.
`template`	This assigns the template resource to the dialog.
`menu`	Associates a menu with a dialog.
`fields`	Defines the set of fields in the dialog.
`complete`	Defines a completion handler for the dialog.
`cancel`	Defines a cancellation handler for the dialog.
`validator`	Defines a validator procedure for the dialog.
`validate`	Define the behavior of the validation phase for the dialog.
`status`	Define desktop status messages displayed during dialog completion or cancellation.

Using the vague description above lets dive right into a basic example of a dialog to control a motor:

```
dialog motor_control
{
  # Values are stored in a structure called MotorParameters.
  struct MotorParameters;
```

```
# Use the std_dialog resource for painting.
template "std_dialog";

# Use a menu defined earlier, called motor_menu for the dialogs
# menu bar.
menu motor_menu;

# Upon completion, execute this code.
complete <-
{
  if (UpdateMotorController())
    Commit();
};

# Upon cancellation, call the Motor::LockoutChanges routine.
cancel Motor::LockoutChanges;

# Display these messages when cancelled/completed.
status
{
  completion   MSG_STATUS_CHANGES_SET;
  cancellation MSG_STATUS_OPERATION_DISABLED;
};

# Validation.
validator <-
{
  DoValidationStuff();
  return OtherValidationStuff();
};

# Only execute the validator if none of the fields are in an
# an error state, as opposed to 'always'.
validate if_no_error;

# The fields.
fields
{
  speed : unsigned
  {
    label    MSG_TEMPERATURE;       # Label for this field.
    type     short;                 # The data type is short.
    minimum  16   : MSG_TOO_SLOW;   # too slow, display this message.
    maximum  1200 : MSG_TOO_FAST;   # too fast, display this one.
    invalid  MSG_INVALID_NUMBER;    # invalid, display this.
    template "scalar";              # Use this template.
  };

  spin : boolean
  {
    label    MSG_MOTOR_ON;          # Label for this field.
    template "boolean";             # Use this template.
  };
```

```
        };
      };
```

Collections

Introduction

Collection widgets present lists of information and can allow manipulation of those lists. Collection widgets do not directly contain items or any kind of list; that is up to the application. Instead, collection widgets make use of an “adaptor” class which provides an abstract interface to the list. Data adaptors are subclasses of `HttpdCollectionData` which implement its abstract methods.

The `HttpdCollectionData` interface represent list items with a generic pointer. To physically present those objects to the user an additional abstract interface is provided to perform the rendering task. Rendering implementations are subclasses of `HttpdCollectionObjectRenderer` and implement template directives.

Seminole includes some pre-built adaptor classes for various data structures. In addition, a manager procedure is provided so that a collection widget can be a member of a dialog box.

Like dialog boxes, collection widgets have menus that can be used to perform associated actions. The collection widget provides methods for determining if an object is selected and if so, which object to menu handlers that take action on a particular object.

`HttpdCollectionData` Reference

Public Methods

`Current`

void * **HttpdCollectionData::Current** (void);

This method returns the currently indexed item of the collection. `HttpdCollectionData` objects maintain a cursor that points to a particular element. This method can return NULL to identify that no item is currently selected.

`First`

int **HttpdCollectionData::First** (void);

This method positions the cursor to the first element of the collection. If an error is encountered during the moving of the cursor then an error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”). For success (even if there are no items in the collection) `0` should be returned.

`Next`

int **HttpdCollectionData::Next** (unsigned int *count*);

This method positions the cursor to the element in the collection *count* items following the current element. If an error is encountered during the moving of the cursor then an error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”). For success (even if there are no more items in the collection) `0` should be returned.

Prev

int **HttpdCollectionData::Prev** (unsigned int *count*);

This method positions the cursor to the element in the collection *count* items preceding the current element. If an error is encountered during the moving of the cursor then an error value should be returned (see Table 4.1, “OS Abstraction Layer Error Codes”). For success (even if there are no more previous in the collection) 0 should be returned.

IsFirst

bool **HttpdCollectionData::IsFirst** (void);

This method should return true is the currently selected element is the first element of the collection. If any other element is selected by the cursor, false should be returned.

Event

int **HttpdCollectionData::Event** (HttpdAppEvent &*ev*);

This method is called when an event from the user (via a button that is part of the collection widget) is destined for a particular item of the collection. The cursor of the collection data is positioned to the selected item before this method is called.

This method should return an error code (see Table 4.1, “OS Abstraction Layer Error Codes”) or a template return code.

Unlike all of the other methods in this class, Event is not abstract and does not have to be implemented by subclasses. The default behavior is to ignore the event.

HttpdCollectionObjectRenderer Reference

The HttpdCollectionObjectRenderer class is a subclass of HttpdSymbolTable that also defines an additional abstract method to specify a particular collection object.

During the rendering of a collection widget each item in the data adaptor is iterated for the current view. The pointer to the current object (obtained from HttpdCollectionData::Current) is given to the SetObject method of this class.

Templates can then contain directives for displaying a collection item that are delivered to subclasses of HttpdCollectionObjectRenderer.

Public Methods

SetObject

int **HttpdCollectionObjectRenderer::SetObject** (void **p_object*);

This method is called to prepare the renderer to render the object identified by *p_object*. The object should remain in effect until the next call to SetObject. It is guaranteed that *p_object* will never be NULL.

This method should return an error code (see Table 4.1, “OS Abstraction Layer Error Codes”) on failure or 0 on success.

`HttpdCollectionWidget` Reference

Template Directives

The `HttpdCollectionWidget` paints itself using templates. The templates can make use of directives provided by `HttpdCollectionWidget` as well as additional directives provided by the renderer object. The standard menu template directives also apply.

Collection Widget Template Evaluation Directives

`cur-list-idx`	This directive evaluates to the zero-based offset of the currently painted item in the widget. It should only be used inside a `records` loop directive.
`data-key`	This directive evaluates to the data key for the widget.

Collection Widget Template Conditional Directives

`more-ahead`	This directive is true if there is an additional page of items to be viewed. Typically this should result in a "Next" button being presented to the user.
`more-behind`	This directive is true if there are additional items before the items displayed on the current page. Typically this should result in a "Prev" button being presented to the user.
`is-first`	This directive is true if currently displayed item is the first in the collection. This directive should only be used inside a `records` loop.
`is-odd-row`	This directive is true if the current row is odd; the first row is considered even. This directive is useful to alternate the colors of each record to assist in readability.

The collection widget also provides a looping directive, `records` that is used to display the current view of items. The `records` should only be evaluated once per collection widget during a single painting cycle.

Public Methods

`HttpdCollectionWidget`

`HttpdCollectionWidget::HttpdCollectionWidget` `(const char *`*`p_local_id`*`, HttpdWidgetContainer *`*`p_parent`*`, const char *`*`p_template`*`, void *`*`p_data`*`, const Options *`*`p_options`*`, int &`*`rc`*`);`

Construct a collection widget. The *`p_local_id`*, *`p_parent`*, and *`rc`* arguments function identically to the corresponding arguments in the `HttpdWidget` constructor.

The *`p_template`* parameter is the resource identifier of the template that should be used to paint this widget. The remaining parameters are passed as a pointer to an Options structure. The most important field of this structure is the setup function (`mpSetup`). This routine is called to fabricate the renderer and collection objects backing the widget. The setup method is passed the *`p_data`* argument of the widget constructor. This pointer is typically used to identify the data that is being displayed.

The Options structure is defined as follows:

```
struct Options
{
   int    (*mpSetup)(void                           *p_data,
                     HttpdCollectionObjectRenderer *&p_render,
                     HttpdCollectionData           *&p_adaptor);

   unsigned int          mPageSize;
   const HttpdMenuItem  *mpMenuItems;
   size_t                mMenuCount;
   HttpdWidgetFlags      mFlags;
   unsigned short        mRefreshInterval;
};
```

All members of the structure should be initialized before being passed to the `HttpdCollectionWidget` constructor. The `mPageSize` member determines the number of items that should be displayed on the widget at any given time. The `mpMenuItems` and `mMenuCount` members define the associated menu. The `mFlags` member contains additional widget flags. The `HttpdCollectionWidget` widget has two additional flags:

- `FREE_RENDERER` - The renderer object was allocated dynamically and should be deleted when the widget is destroyed.
- `FREE_DATA` - The data object was allocated dynamically and should be deleted when the widget is destroyed.

The `mRefreshInterval` member, if greater than zero, forces the client to repaint the widgets for the specified number of seconds. Changes in the data will be displayed during a repaint.

Menu

`HttpdMenu &` **`HttpdCollectionWidget::Menu`** `(void);`

Returns a referene to the menu object associated with the widget.

HaveSelection

`bool` **`HttpdCollectionWidget::HaveSelection`** `(void);`

This function returns `true` if an object is selected. Only code executing as part of an event handler (such as menu handlers) should call this routine.

Data

`HttpdCollectionData *` **`HttpdCollectionWidget::Data`** `(void);`

This function returns a pointer to the data abstraction backing the collection widget.

Renderer

`HttpdCollectionObjectRenderer *` **`HttpdCollectionWidget::Renderer`** `(void);`

This function returns a pointer to the rendering object backing the collection widget.

Manager

`int` **`HttpdCollectionWidget::Manager`** `(const HttpdDialogField *`*`p_field`*`, HttpdWidget *`*`p_widget`*`, int `*`action`*`, va_list `*`va`*`);`

This static method provides basic manager functions to embed a collection widget in a dialog box. The configuration structure is expected to be a HttpdCollectionWidget::Options structure; the same structure that is passed to the `HttpdCollectionWidget` constructor.

`HttpdCollectionListAdaptor` Reference

The `HttpdCollectionListAdaptor` class adapts a list represented by the `HttpdList` class to provide the `HttpdCollectionData` interface.

Public Methods

`HttpdCollectionListAdaptor`

`HttpdCollectionListAdaptor::HttpdCollectionListAdaptor` `(HttpdList &`*`list`*`);`

Associates the adaptor with the list specified by *`list`*. The owner pointer of the `HttpdListNode` object is used as the object pointer that is passed to the rendering object.

`HttpdCollectionArrayAdaptor` Reference

The `HttpdCollectionArrayAdaptor` class adapts a list represented as a normal array (either static or dynamic) to provide the `HttpdCollectionData` interface.

Public Methods

`HttpdCollectionArrayAdaptor`

`HttpdCollectionArrayAdaptor::HttpdCollectionArrayAdaptor` `(void *`*`p_array`*`, size_t `*`count`*`, size_t `*`slotsz`*`);`

Associates the adaptor with the array pointed to by *`p_array`*. The *`count`* parameter specifies how many elements are in the array while *`slotsz`* specifies the size of each element.

`HttpdWidgetBackBlocker` Reference

Introduction

The `HttpdWidgetBackBlocker` widget is a subtle widget that can be used to prevent the use of the "Back" button in many browsers (interfering with the state of the user interface). This widget is normally transparent and can be made a child of the desktop where it can be painted with the `child/` directive anywhere inside the content area.

When the back button is used the client is sent a redirect to a specific URL. A sensible option is to redirect the user to the URL for the application handler. This will result in an update for the current user-interface state.

Public Methods

`HttpdWidgetBackBlocker`

`HttpdWidgetBackBlocker::HttpdWidgetBackBlocker` `(const char *`*`p_local_id`*`, HttpdWidgetContainer *`*`p_parent`*`, int `*`status`*`, const char *`*`p_redirect_to`*`, int &`*`rc`*`);`

Construct a back-blocking widget. The *`p_local_id`*, *`p_parent`*, and *`rc`* arguments function identically to the corresponding arguments in the `HttpdWidget` constructor.

If the “Back” button constraint is violated a redirect is sent as the response using the *`status`* and *`p_redirect_to`* arguments. Under normal circumstances a *`status`* of `HTTPD_RESP_MOVED_TEMP` should be used in conjunction with a *`p_redirect_to`* obtained from calling `HttpdAppHandler::Prefix`.

Chapter 11. Imaging Library

What is the Imaging Library?

Introduction

Seminole handlers are not restricted to generating HTML or textual data; any binary data can be generated. The Seminole imaging library takes advantage of this feature to display data graphically rather than textually.

When using the imaging library, application code can use graphics primitives to draw on a canvas object. The canvas object can then generate a graphics file on demand in response to a request.

The current implementation supports generation of GIF87a graphics files. Custom formats can be implemented by subclassing the abstract `HttpdCanvas` class.

Using the Imaging Library

In order to use the imaging library application code must include the `sem_image.h` header file. There are two ways to use the imaging library. The first approach is to create the canvas, paint the image, and render the canvas in the context of a handlers `Handle` method.

An alternative approach is to draw the image at the applications convenience and only perform the rendering step in the handler. If this approach is used then the access to the canvas object should be synchronized with a mutex (`HttpdMutex`).

Setting up a request handler to draw dynamic images is easy. A subclass of `HttpdHandler` is installed in the server object. The handler then performs the drawing and rendering steps if the request is for this handler:

```
bool MyHandler::Handle(HttpdRequest *p_request)
{
  if (IsMyPath(p_request))
  {
    HttpdGif87aRenderer canvas;
    HttpdColor          red, green blue;

    // Create a 425x125 pixel canvas with 8-bit depth.
    if (canvas.Create(425, 125, 8) != 0)
      goto failure;

    // Allocate colors.
    if (canvas.Color(255, 0, 0, 0, red) != 0)
      goto failure;
    if (canvas.Color(0, 255, 0, 0, green) != 0)
      goto failure;
    if (canvas.Color(0, 0, 255, 0, blue) != 0)
      goto failure;

    // Draw on the canvas.
```

```
      …

      // Render the output.
      canvas.Render(p_request);
      return (true);
    }
    else // Not interested.
      return (false);

  failure:
    p_request->Respond(HTTPD_RESP_SRV_ERROR);
    return (true); // Handled, but not well.
  }
```

The HttpdColor and HttpdCoord) types are abstract types defined by `sem_image.h` for specifying colors and coordinates to the generic drawing routines. There is nothing GIF specific about these types until they are used by the `HttpdGif87aRenderer` canvas.

`HttpdRect` Reference

Introduction

The `HttpdRect` struct represents a rectangular area on the canvas using four points: top, left, bottom, and right. The struct also provides methods for performing various operations with rectangles.

Because rectangles are such a fundamental concept there is little need for accessor methods for each data member. Instead the members can be accessed directly as needed.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Data

`mTop`

```
HttpdCoord mTop;
```

The top (lowest y coordinate) of the rectangle; inclusive.

`mLeft`

```
HttpdCoord mLeft;
```

The left (lowest x coordinate) of the rectangle; inclusive.

mBottom

```
HttpdCoord mBottom;
```

The bottom (highest y coordinate) of the rectangle; inclusive.

mRight

```
HttpdCoord mRight;
```

The right (highest x coordinate) of the rectangle; inclusive.

Public Methods

Width

`HttpdCoord` **`HttpdRect::Width`** `(void);`

This method returns the width of the rectangle.

Height

`HttpdCoord` **`HttpdRect::Height`** `(void);`

This method returns the height of the rectangle.

Intersection

`void` **`HttpdRect::Intersection`** `(const HttpdRect &`*`r`*`);`

This method intersects the rectangle object with the rectangle defined by *r*. The result of the intersection is the new dimension of the rectangle object.

Union

`void` **`HttpdRect::Union`** `(const HttpdRect &`*`r`*`);`

This method adjusts the rectangle object so that it encompases both the original area and the area defined by *r*.

Encloses

`bool` **`HttpdRect::Encloses`** `(const HttpdRect &`*`r`*`);`

This method tests if the rectangle defined by *r* is completely enclosed by the rectangle object.

Overlaps

`bool` **`HttpdRect::Overlaps`** `(const HttpdRect &`*`r`*`);`

This method tests if the rectangle defined by *r* overlaps the area covered by the rectangle object.

Offset

`void` **`HttpdRect::Offset`** `(HttpdCoord` *`x_move`*`, HttpdCoord` *`y_move`*`);`

This method moves the origin of the rectangle by the specified offsets.

Inflate

`void` **`HttpdRect::Inflate`** `(HttpdCoord` *`x_grow`*`, HttpdCoord` *`y_grow`*`);`

The rectangle is grown on all four sides. Therefore the total expansion on the X-axis is twice *x_grow* and the total expansion on the Y-axis is twice *y_grow*

Deflate

`void` **`HttpdRect::Deflate`** `(HttpdCoord` *`x_shrink`*`, HttpdCoord` *`y_shrink`*`);`

The rectangle is shrunk on all four sides. Therefore the total reduction on the X-axis is twice *x_shrink* and the total reduction on the Y-axis is twice *y_shrink*

Subtract

`unsigned int` **`HttpdRect::Subtract`** `(const HttpdRect &`*`r`*`, HttpdRect *`*`p_subrects`*`);`

Compute the list of rectangles covering the area of this rectangle without the rectangle *r*. The *p_subrects* parameter must point to an array of at least `HTTPD_RECT_MAX_AREA_FRAGMENTS` elements. This method returns the number of rectangles used in the *p_subrects* array.

HttpdCanvas Reference

Introduction

`HttpdCanvas` represents an abstract drawing surface. Subclasses of `HttpdCanvas` implement its interface for a particular type of drawing surface.

There are two kinds of drawing routines a canvas provides. Pixel-oriented routines are fast but work only with pixels. Brush-oriented routines are generally more powerful (although slower) and draw using an abstract drawing tool, called a brush.

Brushes can provided by subclasses of `HttpdCanvas` or as stand-alone enhancements to canvas-provided brushes. At any point in time the canvas has an active brush which is used by all brush-based drawing operations.

Public Methods

Color

`int` **`HttpdCanvas::Color`** `(unsigned char` *`red`*`, unsigned char` *`green`*`, unsigned char` *`blue`*`, unsigned char` *`thresh`*`, HttpdColor &`*`color`*`);`

This method obtains a HttpdColor value for the specified values of *`red`*, *`green`*, and *`blue`*. The *`thresh`* value determines the how accurate the resulting color must be. The higher the threshold value the less exact the match is.

If the color could be allocated the *`color`* argument is set to the appropriate color value and 0 is returned. Otherwise an error code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

The purpose of the threshold value is to allow a canvas with limited color resources to share color entries with previously used colors. If `0` is specified for the threshold then an exact match is requested.

Brush

```
HttpdBrush * HttpdCanvas::Brush (HttpdBrush *p_brush);
```

This method sets the current brush of the canvas to the brush pointed to by *`p_brush`*. A pointer to the previously active brush is returned.

Pen

```
HttpdColor HttpdCanvas::Pen (HttpdColor pen);
```

This method sets the current pen color of the canvas to the color specified by *`pen`*. The previous pen color is returned.

The pen color is used by default brush (see `DefaultBrush`) for drawing basic pixels.

Size

```
void HttpdCanvas::Size (HttpdRect &r);
```

This method sets *`r`* to the rectangle that defines the drawing area.

DefaultBrush

```
HttpdBrush * HttpdCanvas::DefaultBrush (void);
```

This method returns a pointer to the “default brush.” This brush paints single pixels (the smallest drawing unit possible) using the current pen color.

Note

This function never returns NULL as the default brush should always exist for the life of the canvas and be created during the construction of the canvas.

Box

```
void HttpdCanvas::Box (const HttpdRect &r);
```

This method draws a one-pixel border around the perimeter of the rectangle specified by *`r`* in the current pen color.

FilledRect

```
void HttpdCanvas::FilledRect (const HttpdRect &r);
```

This method fills the pixels in the rectangle specified by *`r`* with the current pen color.

HPixelLine

`void` **`HttpdCanvas::HPixelLine`** `(HttpdCoord` *`y`*`, HttpdCoord` *`start_x`*`, HttpdCoord` *`stop_x`*`);`

This method draws a 1-pixel tall horizontal line from *`start_x`* to *`stop_x`* (inclusive) at *`y`* pixels from the origin. The line is drawn in the current pen color.

VPixelLine

`void` **`HttpdCanvas::VPixelLine`** `(HttpdCoord` *`x`*`, HttpdCoord` *`start_y`*`, HttpdCoord` *`stop_y`*`);`

This method draws a 1-pixel wide vertical line from *`start_y`* to *`stop_y`* (inclusive) at *`x`* pixels from the origin. The line is drawn in the current pen color.

Line

`void` **`HttpdCanvas::Line`** `(const HttpdRect &`*`rect`*`);`

This method draws a line from the top left corner of the rectangle *`r`* to the bottom right using the current brush.

Circle

`void` **`HttpdCanvas::Circle`** `(HttpdCoord` *`x_center`*`, HttpdCoord` *`y_center`*`, HttpdCoord` *`radius`*`);`

This method draws a circle with the specified center and radius using the current brush.

RoundRect

`void` **`HttpdCanvas::RoundRect`** `(const HttpdRect &`*`rect`*`, HttpdCoord` *`roundness`* `= 4);`

This method draws a rectangle of coordinates *`rect`* with rounded corners. The *`roundness`* parameter specifies how many pixles the diagonal lines on each corner take up. Keep in mind that if *`roundness`* is too large the rounded corners will no longer look round.

Grid

`void` **`HttpdCanvas::Grid`** `(const HttpdRect &`*`r`*`, HttpdCoord` *`x_spaces`*`, HttpdCoord` *`y_spaces`*`);`

This method draws a 1-pixel wide grid using the current pen color covering the specified rectangle. The *`x_spaces`* and *`y_spaces`* parameters determine the number of graduations for each axis.

LineGraph

`void` **`HttpdCanvas::LineGraph`** `(const HttpdRect &`*`r`*`, const long *`*`p_values`*`, size_t` *`count`*`, long` *`minval`*`, long` *`maxval`*`, size_t` *`offset`* `= 0);`

This method graphs the data pointed to by *`p_values`* in the rectangle defined by *`r`*. The data must be signed long integers. The *`count`* parameter specifies how many elements *`p_values`* points to and *`minval`* and *`maxval`* specify the miniumum and maximum ranges to be graphed.

The *`offset`* parameter specifies the where the graph should start within the *`p_values`* array. Reguardless of the value of this parameter, *`count`* data points are always plotted. If *`offset`* is non-zero then the graph simply wraps around to the beginnig of the array until *`count`* values are plotted.

The *`offset`* parameter makes it easy to draw a graph on a window of constantly changing data.

`HttpdSquareBrush` Reference

Introduction

The `HttpdSquareBrush` class implements the abstract `HttpdBrush` interface. It applies an existing brush in a square (or rectangular) pattern for each drawing operation. This brush is generally used to make shapes appear "thicker."

Public Methods

HttpdSquareBrush

`HttpdSquareBrush::HttpdSquareBrush` `(HttpdBrush *`*`p_brush`*`, HttpdCoord` *`x_cnt`*`, HttpdCoord` *`y_cnt`*`);`

The square brush object is configured to draw using *`p_brush`* for *`x_cnt`* repetitions along the x-axis and *`y_cnt`* repetitions on the y-axis.

`HttpdFont` Reference

Introduction

The `HttpdFont` class is used for drawing text on a canvas. Each font object has configurable spacing and scaling. Once created a `HttpdFont` object is read-only and thread safe and may be used from multiple threads (requests) without synchronization.

Public Methods

HttpdFont

`HttpdFont::HttpdFont` `(HttpdCoord` *`scale`* `= 1, HttpdCoord` *`spacing`* `= 1);`

This method constructs of font of the specified scale with the specified character spacing.

CharWidth

`HttpdCoord` **`HttpdFont::CharWidth`** `(char` *`ch`*`); const`

This method computes the width of the specified character (in pixels).

StringWidth

`HttpdCoord` **`HttpdFont::StringWidth`** `(const char *`*`p_string`*`); const`

This method computes the width of the string *`p_string`* (in pixels).

Draw

HttpdCoord **HttpdFont::Draw** (HttpdCanvas **p_canvas*, HttpdCoord *x*, HttpdCoord *y*, const char **p_string*); const

This method draws the string *p_string* at the coordinates *x*, *y* of the canvas *p_canvas* using the current pen color of the canvas.

The total horizontal width of the drawn string (in pixels) is returned.

HttpdGif87aRenderer Reference

Introduction

The HttpdGif87aRenderer class implements the HttpdCanvas interface. The contents of this canvas can be rendered as a GIF87a graphics files in response to an HTTP request.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Thread Safety

This class is completely reentrant. Multiple threads may share this class provided each instance is accessed only by one thread at a time. If instances of this class are to be used by multiple threads then the caller must provide mutual exclusion.

Public Methods

Create

int **HttpdGif87aRenderer::Create** (HttpdCoord *width*, HttpdCoord *height*, unsigned int *depth* = 8);

Before the object can be used the Create method must be called. This method prepares the canvas for drawing with the specified dimentions and color-depth (in bits).

This method may be called on an already-initialized HttpdGif87aRenderer object to re-create the object with a new size and/or depth. Keep in mind that re-creating an already existing canvas will obliterate the image on the previous canvas.

Upon success, 0 is returned. Upon error, a code from Table 4.1, “OS Abstraction Layer Error Codes” is returned.

Render

void **HttpdGif87aRenderer::Render** (HttpdRequest **p_request*);

Given a request object, *p_request*, the contents of the canvas are sent back as the response.

No *MIME* header generation or any other response processing should be performed upon *p_request* before this method is called.

After this method returns no further actions of any kind should be performed on the request object.

Chapter 12. Web Sockets

Introduction

The WebSocket protocol is an extension to HTTP that provides a bidirectional communications path between an HTTP client and Seminole. WebSockets avoid the overhead of repeated socket connections and allow for long-term data transfer that HTTP does not.

An important thing to keep in mind is that many browser implementations of WebSockets do not implement HTTP authentication even though the WebSockets protocol supports it. Therefore it may be necessary to implement authentication using WebSockets messages or via HttpdSessionManager.

Each WebSocket connection also consumes a worker thread. It may also be necessary for applications to ration the number of socket connections open at any one time to prevent other HTTP requests from starvation. In fact the number of WebSocket connections can be artificially restricted by the session limit of `HttpdSessionManager`.

`HttpdWebSocket` Reference

Introduction

The `HttpdWebSocket` class implements an active WebSockets (RFC 6455) connection. In addition this class provides static methods for the detection and establishment of WebSocket connections.

Messages are represented with the following structure:

```
struct Message
{
  HttpdUint8   mMessage;
  void        *mpBuffer;
  size_t       mSize;
};
```

The `mMessage` is the type associated with the frame. The WebSockets protocol defines two frame types:

HTTPD_WS_TEXT_FRAME (UTF-8)
HTTPD_WS_BINARY_FRAME

Public Methods

`IsRequest`

`bool **HttpdWebSocket::IsRequest** (HttpdRequest *p_request);`

This method may be called in the `Handle` method of an `HttpdHandler` subclass to determine if *`p_request`* is a request to establish a web socket connection.

If a web socket connection should be established then true is returned. Otherwise false is returned.

Connect

`bool` **`HttpdWebSocket::Connect`** `(HttpdRequest *`*`p_request`*`, HttpdWebSocket &`*`socket`*`);`

If *`p_request`* is a web socket request (`IsRequest` returned true) this method attempts to connect to the far end. If successful *`socket`* may then be used to perform I/O. If unsuccessful then no further processing should be done on *`p_request`* and true should be returned from `Handle`.

This method returns true upon success or false upon failure.

Setup

`bool` **`HttpdWebSocket::Setup`** `(HttpdRequest *`*`p_request`*`, HttpdWebSocket &`*`socket`*`);`

If *`p_request`* is a web socket request and the connection can be established then true is returned and *`socket`* must eventually be closed.

In the event of failure false is returned and the `HttpdHandler::Handle` method must perform no further processing and return true.

Close

`void` **`HttpdWebSocket::Close`** `(void);`

This method closes a websocket connection.

SetMaxRxSize

`void` **`HttpdWebSocket::SetMaxRxSize`** `(size_t` *`max_msg_size`*`);`

To prevent clients from consuming excessive amounts of memory the maximum message size received by a socket may be set using this method. If a message is received that is larger than the maximum size then the connection is severed.

The default maximum message size is the largest value that size_t can represent.

Send

`int` **`HttpdWebSocket::Send`** `(const Message &`*`msg`*`);`

This method sends the message to the peer.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). Additionally `HttpdWebSocket`::HTTPD_WS_CONNECTION_CLOSED is returned if the socket is closed for whatever reason.

Received

`int` **`HttpdWebSocket::Received`** `(Message &`*`msg`*`, unsigned int` *`timeout`*`);`

Wait for a message to be transmitted by the far end or for *`timeout`* seconds to elapse with no received message. If a message is successfully received then the fields of *`msg`* are filled in and `0` is returned.

After this method returns the message buffer may be used (and even written to) until `Finish` is called with the message. Until `Finish` is called no further calls to `Received` should be made.

However it is possible to call `Send` prior to calling `Finish`. In fact it is posisble to call `Send` with the message structure (and buffer) that was recieved. Because the message buffer can be written to the responses to the client can be a modified version of the request message.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). Additionally `HttpdWebSocket`::HTTPD_WS_CONNECTION_CLOSED is returned if the socket is closed for whatever reason. If no message is received within the timeout period then `HttpdOpSys`::ERR_NOTREADY is returned.

Normally code should not attempt to retry receiving a message if an error is returned. An exception to this rule is if ERR_NOTREADY is returned.

Code that uses `HttpdWebSocket` objects should keep in mind that even in a transmit-only sitation this method should be called periodically to ensure that the connection is properly maintained (i.e. ping frames are acknowledged) and disconnects are detected.

Received (multiple wait version)

`int` **`HttpdWebSocket::Received`** `(Message &`*`msg`*`, unsigned int `*`timeout`*`, HttpdSocketWaitHandle `*`wait_for`*`);`

This method is similar to the version defined above that does not include the *`wait_for`* parameter. Additionally this version is only available if the portability layer supports multiple-wait socket reception (`HAVE_SOCK_WAIT` is defined as `1`).

This version waits either for a message to be received from the peer, for the specified timeout to occur, or for the platform specific signaling mechanism referenced by *`wait_for`* to be signaled. The latter results in a return of `HttpdOpSys::ERR_NOTREADY` to be returned.

Finish

`virtual void` **`HttpdWebSocket::Finish`** `(const Message &`*`msg`*`);`

Complete message reception and release any resources held by *`msg`*. This method should be called on messages obtained by calling `Received`.

Protected Methods

UnhandledFrame

`virtual bool` **`HttpdWebSocket::UnhandledFrame`** `(Message &`*`msg`*`, HttpdUint8 `*`opcode`*`, bool `*`fin`*`, int &`*`rc`*`);`

This method is called when a frame is received with an unrecognized `OPCODE` field. When called the *`opcode`* parameter is the value of the 4-bit opcode field in the received frame. The *`fin`* parameter is set to true if the `FIN` bit is set in the frame. Finally the *`msg`* structure contains a valid size and buffer pointer for the received frame.

If this method returns false then the frame is dropped and frame reception begins again. If true is returned then *`rc`* is returned to the caller of `Received`.

It is expected that if the frame is to be handled gracefully the `Finish` method is called on *`msg`*.

This implementation simply causes the `Received` method to return `HttpdOpSys::ERR_BADFORMAT` in the event of an unrecognized opcode.

Fragment

`virtual bool` **`HttpdWebSocket::Fragment`** `(Message &`*`msg`*`, HttpdUint8` *`opcode`*`, bool` *`fin`*`, int &`*`rc`*`);`

This method handles fragment reassembly. If a data frame with the `FIN` bit clear or a continuation frame is received from the far end it is routed to this method. This method then stores the fragment in a reassembly buffer. The *`opcode`* parameter is the value of the 4-bit opcode field in the received frame. The *`fin`* parameter is set to true if the `FIN` bit is set in the frame. Finally the *`msg`* structure contains a valid size and buffer pointer for the received frame.

If this method returns false then the frame is dropped and frame reception begins again. If true is returned then *`rc`* is returned to the caller of `Received`.

It is expected that this method also removes the message bytes from the protected `mFifo` member after processing. For example:

```
mFifo.Consume(msg.mSize);
```

In the event that the client wants to send large messages that can not be buffered in the memory of the target this method may be overridden to process fragments on the fly.

If this method is subclassed to support special fragment processing it is important to remember that frames with an opcode of HTTPD_WS_TEXT_FRAME must contain UTF-8 encoded data. It is possible that an encoded character may straddle two frames and this needs to be accounted for.

Chapter 13. Endpoint Discovery

Introduction

Endpoint Location

Providing a web-based interface for a device is only useful if it can be easily found. With a multitude of embedded devices all offering a web interface it can be difficult to find out what URL to go to.

Seminole provides two components to help solve this dilemma. One component, the discovery server, sits alongside the webserver and uses *multicast* UDP to help users locate the webserver. The other component, the discovery client, is either installed or downloaded (from a well-known site, such as a corporate website) where the web browser runs.

When the discovery client is invoked it will attempt to find all of the reachable discovery servers on a given network that match a set of criteria. In addition to the URL other small bits of information can be transmitted from the the server to the client. This allows the discovery mechanism to also act as an overall status display for all the nodes in the network.

The Discovery Server

The discovery server (`HttpdDiscoveryServer`) executes on its own thread and does not impact the processing of HTTP requests in any way. The discovery server does associate with an instance of `Httpd` to derive the target URL.

A small structure must be provided to the server that describes the class of device, any name-value parameters to report back and the network parameters. A default set of network parameters is provided that should be used in most cases.

Several methods of `HttpdDiscoveryServer` may be overridden in a subclass for added functionality. The most important of these, `BuildResponse` can be used to send real-time status data about this endpoint for the client to display. A good example of this would be if any faults are present, security alerts, or even environmental conditions (i.e. temperature, power supply levels, etc.).

The Discovery Client

The included discovery client is written in the Java programming language so that it can be run on a wide variety of client systems (we hope). It runs as an applet within a web browser. This allows it to open a connection to the located in the user's preferred browser.

The client should require only a few configuration parameters in an HTML document for configuration and some way of delivering the client to where it is needed. One old fashioned approach to delivering the client is to package the files on a CD-ROM. Another more modern approach is to delivery the client via a well-known web server. It is also possible to serve the discovery client right from an embedded device using Seminole although this implies that the URL for at least one device is well known.

The Java Discovery Client

Compiling

Compiling the Java discovery client requires an operating JDK in your current path. As with all other Seminole components the build system is capable of building the client automatically although it is not built as part of the default build procedure. Instead the `discovery_client` target must be used. For example:

```
$ ./buildit ports/PORTFILE discovery_client
```

If all goes well the discovery client (well the JAR file) is named `built/PORTFILE/lib/Discovery.jar` This file can be combined with some instructional HTML that launches it and deployed to wherever it is needed.

Due to the Java security model it is necessary that if the client is delivered over an untrusted source (HTTP for example) that it be signed. Signing the JAR file requires that a public key be generated and given a name. Once the key is created it can be signed by setting the `JAVA_SIGN_KEY` Perl variable in your build file to the alias given to your key.

There are several ways to create the key but the easiest is using the **keytool** command. There are two steps to making this key. The first command is used to generate the key. The second signs the key with itself. It is also possible to sign the key via a trusted third party.

It is also important to specify the validity (in days) of the key. To avoid repeatedly having to re-sign the applet it is a good idea to make the key last a long time. Of course this has security implications and the security conscious should weigh their options.

To get started generating a key named `MyCompany` use the following commands:

```
$ keytool -genkey -validity 365 -alias MyCompany
$ keytool -selfcert -validity 365 -alias MyCompany
```

Once the keys are setup the discovery client can be cleaned by “building” the `discovery_clean` target:

```
$ ./buildit ports/PORTFILE discovery_clean
```

After cleaning the client can then be re-built at described above.

Instructional HTML

In order to properly execute the JAR an HTML document must be created to launch the applet. This is also a terrific place to include things like product setup instructions, troubleshooting guides, and technical support contacts. The minimal content required to run the client is an `APPLET` tag. For example:

```
<html>
 <head><title>Active Frobinator 2000's</title>
 <body>
  <applet
     code="gladesoft.seminole.discovery.DiscoveryApplet"
     archive="Discovery.jar"
     height=420 width=340>
    <param name="discovery.silence-timeout-url" value="none_found.html">
  </applet>
 </body>
</html>
```

The parameter that is set within the `APPLET` tag, `discovery.silence-timeout-url` is only one of many that can be set to control the appearance and behavior of the client. Almost all of the parameters have reasonable defaults if not specified.

Note

All parameters used by the discovery client are prefixed with `discovery.`. Any further references to the parameters imply this prefix.

Some parameters are not configurable at runtime and must be changed by recompiling the applet. Most of these hard-coded constants as well as the defaults are in the `DiscoveryConfiguration` class. The most important parameter is `TIME_INTERVAL`. This is the number of milliseconds an operation cycle of the client takes. The default value is 100ms. Other parameters are based on these units.

Parameters that specify fonts do so in a standard way. They consist of three fields that are comma separated. The first field is the name of the font (as seen by the JRE). The second parameter is one or more of the following attributes combined with + characters:

- `bold` - Renders the font in bold.
- `italic` - Renders the font in italics.
- `null` - Ignored. Useful to signify no attributes.

The final field is the point size of the font. So a font specification may look like this:

```
SansSerif,bold,14
```

Colors also are specified in a standard way. A color is composed of red, gree, and blue values (ranging from 0 to 1) separated by commas. An optional alpha component (also in the range from 0 to 1) may follow the green value.

Table 13.1. Discovery Client Parameters

Option	Meaning	Default Value
`rx-port`	This is the port to listen on for beacons from the server. It should match the configuration of the `HttpdDiscoveryServer` instance.	1175
`tx-port`	This is the port to send discovery requests to the server. It should match the configuration of the `HttpdDiscoveryServer` instance.	1176
`broadcast`	This is the addresses on which discovery requests are sent. It should be the multicast group address that the `HttpdDisocveryServer` is configured to listen on. Multiple	`238.17.40.9,ff05::1:1174`

Option	Meaning	Default Value
	addresses may be specified by separating them with commas.	
`classes`	This is the list of device classes to allow through. Every instance of `HttpdDiscoveryServer` has a list of "classes" that describe the device. If this parameter is present then it is a comma separated list of device classes to display. Only if there is an intersection in the two lists is the endpoint displayed.	None. Not specifying this parameter implies no filtering.
`allowed-schemes`	This parameter is a comma separated list of URL "schemes" without the `://` portion that are displayable. Discovery servers can provide the scheme used to access them. However this may be an untrustworthy source. As such this parameter can be used to filter out unwanted protocols.	`http,https`
`bind`	This parameter allows the socket to be bound to a particular interface address.	There is no default. When this parameter is not specified the socket is bound to the wildcard interface address.
`max-age`	This parameter controls the number of time units that an endpoint will remain in the list without receiving a beacon from the discovery server. Setting this value too low means that in heavy packet loss situations endpoints may disappear prematurely. Setting this value too high means that if an endpoint is removed from the network it will remain in the list way too long.	20 time interval units
`send-pacing`	This parameter controls how often probe packets are sent. It should be smaller than `max-age`. However setting this parameter too high will result in longer delays for a new node to display. Setting this parameter lower increases network traffic.	10 time interval units
`view-in-place`	This setting determines if the selected endpoint should be displayed in the same browser window (for values of `on`, `true`, or `1`) or in a separate window	`true`

Option	Meaning	Default Value
	(for values of `off`, `false`, or `0`). Some browsers have security implications with regards to opening new windows. For maximum reliability and ease of use this option should probably be set to `true`. However if multiple devices are typical then you may wish to consider setting this value to `false`.	
`max-silence`	If no endpoint is seen by this many time units the client will navigate to the URL specified by the `silence-timeout-url` parameter. This is useful to redirect to a “debugging” page to help the user figure out why no endpoints were seen.	10 time interval units
`silence-timeout-url`	This is the URL to navigate to if no endpoint is found in `max-silence` time units.	If this parameter is not specified then this feature is disabled.
`load-fail-url`	The loading of the applet can fail for a variety of reasons (for example lack of permission or an incompatible runtime environment). In the event this happens the client will attempt to navigate to this URL where instructions to remedy the situation or discover the endpoint manually may be found.	If this parameter is not specified then this feature is disabled.
`url-font`	This value sets the font a discovered URL is displayed with.	`Monospaced,bold,16`
`detail-label-font`	This value sets the font a for an attribute label.	`SansSerif,bold,12`
`detail-value-font`	This value sets the font a for an attribute value.	`SansSerif,null,12`
`bg-color`	This value sets the background color for a discovered endpoint.	White
`highlight-bg-color`	This value sets the background color for a discovered endpoint that has been selected by the user.	`0.47,0.94,0.47` (a light green)
`url-color`	This value sets the background color for the displayed URL.	Blue
`attr-label-color`	This value sets the color that the attribute label text is drawn in.	Black

Option	**Meaning**	**Default Value**
`attr-color`	This value sets the color that the attribute value text is drawn in.	Gray
`attr-label-color-hl`	This value sets the color that highlighted attribute label text is drawn in.	Orange
`attr-color-hl`	This value sets the color that highlighted attribute value text is drawn in.	Orange
`border-color`	This value sets the color that the border around a discovered endpoint is drawn in.	Black
`attr-order`	This value sets the order in which attributes are displayed. See below for further details.	`descr`
`ep-sort`	This value is an optional specifier for how to sort the displayed endpoints. See below for further details.	`time`
`enable-icons`	If enabled then an optional icon will be displayed for each discovered endpoint (if the server provides one). Otherwise icons from the server are ignored.	`true`
`max-icon-width`	Specifies the maximum icon width allowed before an icon is scaled. The value may be specified as a percentage of the available area if suffixed by a `%` or an absolute size in pixels.	`10%`
`max-icon-height`	Specifies the maixmum icon height allows before an icon is scaled. The value may be specified as a percentage of the available area if suffixed by a `%` or an absolute size in pixels.	`30%`
`min-audio-time`	Specifies the minimum amount of time (in milliseconds) between audio notifications. In order to prevent repeated audio notifications this value can be set to ignore audio notifications if they are close together (timewise). Setting this parameter to `0` will disable audio cues.	`30%`
`string-file`	If specified this value references a properties file to use for localized strings. Setting this avoids using the Java `ResourceBundle`	Not set

Option	Meaning	Default Value
	class which often requests many non-existant files. For example setting this to `/res.properties` will select the default properties file at the root of the JAR.	
`string-url`	If specified this value is a URL relative to the document base of a properties file to use for localized strings.	Not set

By default the client is built in “restricted” mode. This disables the effect of the `broadcast`, `bind`, `rx-port`, and `tx-port` parameters. Normally these should always be left at the defaults for security reasons. However, for debugging or special deployments the constant `RESTRICTED`, defined in `DiscoveryConfiguration.java` may be set to `false` to enable the functionality of these parameters.

Attributes

Each endpoint can display a small set of associated data items along with its URL. Each of these data items has an internal name that the client and server use to identify an attribute. The client also associates a “display name” along with each attribute that is shown to the user.

The `attr-order` parameter specifies the internal name of all possible attributes as well as the order in which to display them. It is okay to specify a display name that is not received from the server in the `attr-order` parameter, it will simply be ignored. At a minimum servers should send an attribute called `descr` that is a brief description of the endpoint.

Unlike other labels, the displayable string for the `descr` attribute is automatically defined to display “Description”. Other attribute display text can be set by defining specially named parameters. For example:

```
<param name="discovery.label-attr-vers" value="Version">
```

The attribute above configures the client to display the string `Version` as the label for a parameter of `vers`. Any attribute name mentioned in `attr-order` should have a translation entry as in the above example.

Formatting Attributes

There are also a number of parameters that can affect the formatting of the attribute values. Some of these parameters perform string manipulations while others work on attributes that are textual representations of numbers. The formatting is done using the classes in the `java.text` package that provides locale-specific formatting. Therefore it is not necessary for a discovery server to attempt complex formatting.

Table 13.2. Attribute Formatting Parameters

Option	Description
`format-attr-trim-`*`name`*	If this parameter is `true` then whitespace is removed from the front and the back of attribute *`name`*.

Option	**Description**
`format-attr-lowcase-`*`name`*	If this parameter is `true` then the value of attribute *`name`* is lowercased.
`format-attr-lowcase-`*`name`*	If this parameter is `true` then the value of attribute *`name`* is lowercased.
`format-attr-type-`*`name`*	This parameter determines how attribute *`name`* is to be interpreted. If this parameter is `number` then the value is formatted as a localized number. If this parameter is `percentage` then the value is formatted as a percentage (scaled by 100).
`format-attr-integer-`*`name`*	If this parameter is `true` and the value is to formatted as a number then the value is treated as a signed number rather than a floating-point value.
`format-attr-scale-`*`name`*	If the formatting type is either `number` or `percentage` then the value of the attribute is multiplied by this value. For non-integer values this may be a fractional value that can be used to scale down the provided value.
`format-attr-bias-`*`name`*	If the formatting type is either `number` or `percentage` then this value is added to the value of the attribute. For non-integer values this may be a fractional value that can be used to scale down the provided value.
`format-attr-maxfrac-`*`name`*	If the formatting type is either `number` or `percentage` and this parameter is present the maximum number of fractional digits is set to its value. See the Java `NumberFormat.setMaximumFractionDigits` documentation for details.
`format-attr-minfrac-`*`name`*	If the formatting type is either `number` or `percentage` and this parameter is present the minimum number of fractional digits is set to its value. See the Java `NumberFormat.setMinimumFractionDigits` documentation for details.
`format-attr-maxint-`*`name`*	If the formatting type is either `number` or `percentage` and this parameter is present the maximum number of whole number digits is set to its value. See the Java `NumberFormat.setMaximumIntegerDigits` documentation for details.
`format-attr-minint-`*`name`*	If the formatting type is either `number` or `percentage` and this parameter is present the minimum number of whole number digits is set to its value. See the Java `NumberFormat.setMinimumIntegerDigits` documentation for details.
`format-attr-suffix-`*`name`*	If the formatting type is either `number` or `percentage` and this parameter is present then

Option	Description
	the value of this parameter is used to construct a `ChoiceFormat` object to format a suffix for the numeric attribute value. See the Java documentation for `ChoiceFormat` for details on the possible values of this parameter.
`format-attr-map-`*`name`*	If present this parameter allows the value from the server to be mapped to a different value. The map is parsed using the Java `StreamTokenizer`; this allows character quoting and escaping. The format of this parameter is a series of name-value pairs. For example if the value of this parameter is `critical="Critical Alarm\nImmediate Attention Needed",warning="Warnings present"` the values `critical` and `warning` from the discovery server will be mapped to the more verbose values.

Sorting Endpoints

By default endpoints are displayed with the most newly discovered endpoint on top. The sort order can be customized in the event many endpoints are expected. In fact multiple sort orders can be defined and the user can select amongst them.

Sort orders are specified with a simple specification string. Multiple sort terms can be specified, separated with commas. Endpoints are sorted according to the terms from left (most general sorting) to right (most specific sorting).

Attributes may be sorted according to the type of data they carry. For example to sort the list of endpoints based upon the `descr` field as a string followed by discovery time use the following:

```
<param name="discovery.ep-sort" value="string:descr,time">
```

The sort specifier `time` is used to sort by the discovery time. The specifier `string` sorts according to the current locale of the client. Any specifier can have its sort ordering inverted by placing a `!` or `~` in front of it. For example to sort the the description in the opposite direction but the same direction for the discovery time use:

```
<param name="discovery.ep-sort" value="!string:descr,time">
```

The difference between `!` or `~` is subtle and has to do with when the attributes are not present in endpoints. By default endpoints without the attribute are considered to be after all of the sorted records. In the case of `!` the ordering of all comparisons is reversed. This results in endpoints without the attribute coming first. If `~` is specified then only the comparisons for records with the attribute is reversed. This results in endpoints without the attribute remaining at the end of the sort order.

For attributes that are numeric values you can sort using a specifier of `double`. For example to sort based upon the value of an attribute called `temperature` use:

```
<param name="discovery.ep-sort" value="double:temperature">
```

Commonly attributes are used to specify certain device states as a string that is a set of possible enumerations. For these cases an arbitrary set of strings can specify a particular ordering. For example assume a device can send a state of `critical`, `warning`, `maintenance`, or `operating`. We would like to see them in that order so that devices needing attention appear on top. We could do this using the `enum` sort specifier like this:

```
<param name="discovery.ep-sort"
       value="enum:state:critical|warning|maintenance|operating">
```

In the previous examples we have been setting the parameter `ep-sort` to our sort specification. But what if you wanted multiple sort specifications and allow the user to select through them. In fact all of the examples above may be desirable in certain cases and we should really let the user choose what they want to see.

If we don't specify `ep-sort` but instead specify a list of sorts (as shown below) the user will be presented with a selection box of all the possible sorting options.

```
<param name="discovery.ep-sort.0"
       value="Newest::time">
<param name="discovery.ep-sort.1"
       value="*By Status::enum:state:critical|warning|maintenance|operating">
<param name="discovery.ep-sort.2"
       value="By Temperature::double:temperature,time">
```

In the example above the `By Status` configuration is selected by default because it begins with an asterisk. The display names are shown and the associated with the sort specifications that follow them.

Endpoint Icons

The discovery server can optionally provide a pointer to an icon that the Java client will display. The icon may either be served up by the instance of Seminole running on the endpoint or included in the JAR file. For security reasons an arbitrary URL is not allowed as this could potentially allow a malicious device to attempt a form of cross-site scripting attack.

In order to display an icon the discovery server must publish two parameters:

- `icon_loc` specifies the location of the icon. It can either be `server` to specify that the icon is on the discovered webserver or `resource` to indicate that the icon is a resource in the JAR. Alternatively, `icon_loc` can be `url` to refer to a URL from the configuration.

- `icon_file` specifies the path (if `icon_loc` is `server`) or the resource name (if `icon_loc` is `resource`).

If `icon_loc` is `url` and a configuration parameter exists by the name of `icon-url-`*`icon_fn`* then the value of that parameter is used as the URL for the icon. Additionally if the attribute `icon_name` is present it is considered to be an additional path component appended to the configured URL.

Class Filters

In some instances it may be desirable to allow a user to select which class of devices they are discovering. By default filtering can be set with the `classes` parameter. Rather than setting this parameter a list of class filters can be created that the user can select at runtime.

Creating a filter list involves two things: A user visible name and the list of class names to filter on. Each of these are specified as parameters with numbered names. For example, let us consider three possible filtered views for a security system: Cameras, door and/or window switches, and motion sensors. We could define these filters with the following parameters:

```
<param name="discovery.classes.cameras"
          value="cams">
<param name="discovery.classes.switches"
          value="door-switches,window-switches">
<param name="discovery.classes.motion"
          value="motion-sensors">
<param name="discovery.filter.0"
          value="cameras::Video Cameras">
<param name="discovery.filter.1"
          value="switches::Intrusion Switches">
<param name="discovery.filter.2"
          value="*motion::Motion Sensors">
```

The list of classes for a particular filter is given a name and the list of device class names (as configured in the discovery server) are assigned to that name. In the example above we see that the `switches` filter will find devices that belong to either the `door-switches` or `window-switches` device classes.

The list of selectable filters is defined with the `discovery.filter.`*`number`* parameters. The name of the filter set is followed by `::` and the display name of the filter. The filters are displayed to the user in numerical order. The filter with the name preceeded by an asterisk is the filter selected by default.

Change Highlighting

If a discovered device is displaying important statistics about itself it may be desirable to briefly highlight changes in the display so that they catch the user's eye. By default this feature is off but it can easily be turned via a configuration parameter: `change-highlight-time`

This parameter specifies the number of milliseconds that a changed attribute is to remain highlighted for. It is important to keep in mind that this interval is processed only in multiples of `DiscoveryConfiguration.AGE_PACING`; by default 1000ms.

Setting this parameter to `0` or not defining it disables change highlighting.

`HttpdDiscoveryServer` Reference

Introduction

`HttpdDiscoveryServer` implements the server side of the Seminole discovery protocol. The server can be automatically found by discovery clients (such as the Java Discovery Client).

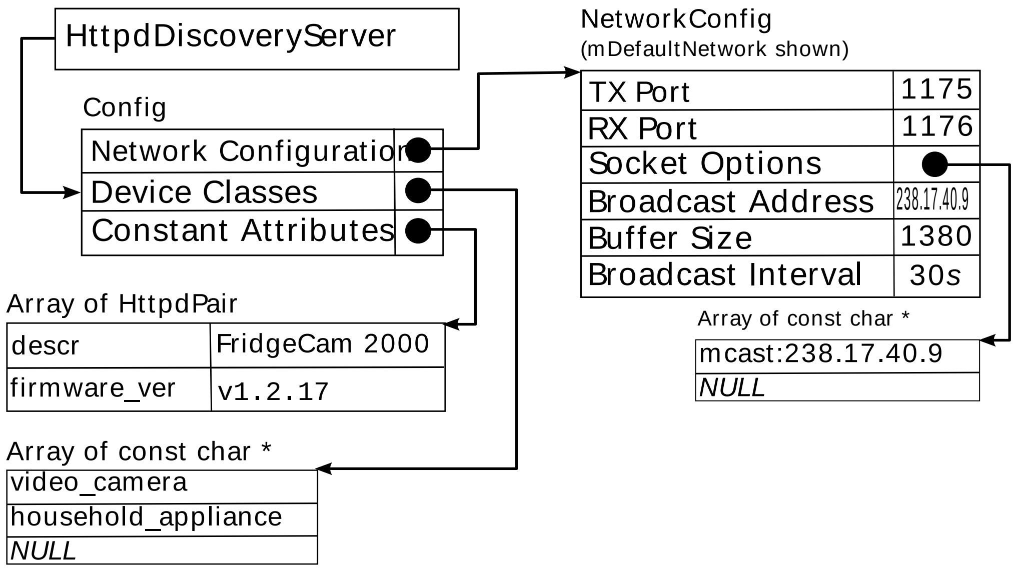
The server is also capable of sending along a set of named attributes. A static list can be provided to the server at creation time or the `HttpdDiscoveryServer` can be subclassed to allow the transmission of dynamic data.

Note

The `HttpdDiscoveryServer` class is only available if the portability layer provides the `HAVE_UDP_SOCKETS` feature.

Configuration Structures

The `HttpdDiscoveryServer` class requires several configuration structures. `HttpdDiscoveryServer` includes reasonable defaults for some of these strucutres. The figure below shows an overview of how the configuration structures are arranged:



The top-level configuration structure points to three different structures:

```
struct Config
{
  const NetworkConfig   *mpNetwork;
  const char *const     *mppDeviceClassList;
  size_t                 mParamCount;
  const HttpdPair       *mpParams;
};
```

The `mpNetwork` field points to the network configuration structure. The standard protocol configuration is available as `HttpdDiscoveryServer::mDefaultNetwork`. The device class list is a set of classes that best describe this device. This is most often used to to filter out unwanted endpoints in the discovery client. The device class list should be a `NULL`-terminated list of device class names. If you are unsure of what classes your device falls under then contact our support team. The `mpParams` pointer points to an array of name-value pairs that are sent out with each request. If `mParamCount` is `0` then `mpParams` may be `NULL`. Otherwise `mParamCount` should be the number of entries in the `mpParams` array. The HTTPD_NUMELEM macro can be used if the compiler knows the size of the array.

It is normally not necessary to declare the `NetworkConfig` as the default is almost always appropriate. The default network configuration, available as `HttpdDiscoveryServer::mDefaultNetwork`, can be used for the `Config` structure above.

If INC_IPV6_SUPPORT is enabled then `HttpdDiscoveryServer::mIPv6Network` is available for use on IPv6 enabled devices.

```
struct NetworkConfig
{
  HttpdIpPort          mTxPort;
  HttpdIpPort          mRxPort;
  const char *const   *mppSocketOptions;
  const char          *mpBroadcastAddress;
  size_t               mBufferSize;
  unsigned int         mBroadcastInterval;
};
```

The `mTxPort` and `mRxPort` members control the ports the server uses. The `mppSocketOptions` is passed to the `HttpdUdpServerSocket` encapsulated by the discovery server. The server transmits beacons on `mpBroadcastAddress`. Both transmission and reception share a buffer of `mBufferSize` bytes. Packets larger than this size can not be processed. The server will also send an unsolicited broadcast every `mBroadcastInterval` milliseconds.

Public Methods

HttpdDiscoveryServer

`HttpdDiscoveryServer::HttpdDiscoveryServer` `(Httpd *`*`p_server`*`, const HttpdDiscoveryServer::Config *`*`p_config`*`);`

This method constructs a discovery server and points it to the configuration described by *`p_config`*. The server is associated with the webserver instance *`p_server`*.

This constructor only initializes the object. To allocate all of the required resources you must call the `Create` method.

Note

The lifetime of the configuration structure and the webserver must be equal to or exceed the lifetime of the `HttpdDiscoveryServer` object.

Create

`int` **`HttpdDiscoveryServer::Create`** `(void);`

This method must be called before the discovery server can be started. It allocates all of the necessary resources for operation. These resources remain allocated until the discovery server object is destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Start

`int` **`HttpdDiscoveryServer::Start`** `(void);`

This method starts the discovery service.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Stop

`void` **`HttpdDiscoveryServer::Stop`** `(void);`

This method stops the discovery service.

Protected Methods

Request processing can be customized by subclassing `HttpdDiscoveryServer` and overriding the protected methods. All of the context involving a particular request are bundled up into a `Request` structure:

```
struct Request
{
  HttpdCgiParameter *mpQuery;
  HttpdIpAddress     mInquisitorAddress;
  HttpdIpPort        mInquisitorPort;
};
```

The fields are as follows:

`mpQuery`	This is the list of name/value pairs contained in the request packet.
`mInquisitorAddress`	This is the address of the host making the discovery request. The reply will be sent back to this address.
`mInquisitorPort`	This is the port of the host making the discovery request.

ShouldHandleRequest

`bool` **`HttpdDiscoveryServer::ShouldHandleRequest`** `(Request *`*`p_request`*`);`

This method determines if an incoming request for discovery should be responded to. The list of device classes in the request is intersected with the list of device classes in the server configuration. If the intersection is not empty then this method returns true. Otherwise false is returned.

Subclasses may override this to change the response criteria.

BuildResponse

`int` **`HttpdDiscoveryServer::BuildResponse`** `(HttpdCgiWriter *`*`p_writer`*`);`

This method writes the response packet to be sent to the machine running the discovery client. In particular name/value pairs are written to the *`p_writer`* object.

The default implementation adds the information necessary to derive the URL of the associated webserver plus the static array of `HttpdPair` elements in the `Config` structure.

Subclasses may override this to add dynamic data to the response. Subclasses should call the base class implementation and avoid writing an attribute named `port` or `scheme` to *`p_writer`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

PrepareResponse

`int` **`HttpdDiscoveryServer::PrepareResponse`** `(void);`

This method rebuilds the beacon packet if a rebuild is necessary (`mRebuildResponse` is `true`). It is called before a response packet is sent.

Subclasses may override this if they intend to send dynamic data (in order to set `mRebuildResponse` prior to calling the default implementation).

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

SendBeacon

`bool` **`HttpdDiscoveryServer::SendBeacon`** `(void);`

This method is called when the server needs to decide if a beacon should be sent for clients that can collect them without solicitation. The default implementation simply calls `BuildResponse` to prepare the response and returns `true` if building the response was successful.

Subclasses can use this method to inhibit the sending of the beacons if, for example, the device is not ready for administration at this time.

Protected Data

mRebuildResponse

`bool mRebuildResponse`

This member is checked at the start of `HttpdDiscoveryServer::PrepareResponse`. If it is `true` then the packet contents is rebuilt. Once built this member variable is set to `false`. This prevents the CPU overhead of regenerating the packet each time it needs to be transmitted.

Subclasses of `HttpdDiscoveryServer` can override `PrepareResponse` and set this variable to regenerate the packet. This is useful, for example, if it contains dynamic data (such as some form of device status).

`HttpdDiscoveryClient` Reference

Introduction

The `HttpdDiscoveryClient` implements a native (i.e. non-Java) client for the discovery server (`HttpdDiscoveryServer`). It manages a list of `HttpdDiscoveredEndpoint` objects where each one represents a discovered endpoint on the network.

Both `HttpdDiscoveryClient` and `HttpdDiscoveredEndpoint` are abstract. A native discovery client must subclass both of these classes and implement a user interface.

Note

The `HttpdDiscoveryClient` class is only available if the portability layer provides the `HAVE_UDP_SOCKETS` feature.

Configuration Structures

The `HttpdDiscoveryClient` class requires a configuration structure, NetworkConfig, to operate. Reasonable defaults for these parameters are provided by `HttpdDiscoveryClient::mDefaultNetwork`. If INC_IPV6_SUPPORT is enabled then `HttpdDiscoveryClient::mIPv6Network` is available for use on IPv6 enabled devices.

```
struct NetworkConfig
{
  HttpdIpPort              mTxPort;
  HttpdIpPort              mRxPort;
  const char *const       *mppSocketOptions;
  const char              *mpBroadcastAddress;
  size_t                   mBufferSize;
  unsigned int             mBroadcastInterval;
  int                      mMaxTimeToLive;
  const char *const       *mppClasses;
};
```

The `mTxPort` and `mRxPort` members control the ports the client uses. They should be the reverse of the server configuration. The `mppSocketOptions` is passed to the `HttpdUdpServerSocket` encapsulated by the client. The client transmits beacons on `mpBroadcastAddress`. Both transmission and reception share a buffer of `mBufferSize` bytes. Packets larger than this size can not be processed. The client will also send an unsolicited broadcast every `mBroadcastInterval`. `mMaxTimeToLive` controls, in seconds, the longest an endpoint is considered “alive” without any response from the server. Finally, `mppClasses` is a `NULL`-terminated list of device classes that is to be queried for. If `mppClasses` is `NULL` devices of all classes are discovered.

Public Methods

HttpdDiscoveryClient

`HttpdDiscoveryClient::HttpdDiscoveryClient` `(const HttpdDiscoveryClient::NetworkConfig *`*`p_config`*`);`

This method constructs a discovery client and points it to the configuration described by *`p_config`*.

This constructor only initializes the object. To allocate all of the required resources you must call the `Create` method.

Note

The lifetime of the configuration structure must be equal to or exceed the lifetime of the `HttpdDiscoveryClient` object.

Create

`int` **`HttpdDiscoveryClient::Create`** `(void);`

This method must be called before the discovery client can be started. It allocates all of the necessary resources for operation. These resources remain allocated until the discovery client object is destroyed.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Start

`int` **`HttpdDiscoveryClient::Start`** `(void);`

This method starts the discovery client.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Stop

`void` **`HttpdDiscoveryClient::Stop`** `(void);`

This method stops the discovery client.

Protected Methods

CreateEndpoint

`HttpdDiscoveredEndpoint *`**`HttpdDiscoveryClient::CreateEndpoint`** `(HttpdIpAddress` *`addr`*`, HttpdIpPort` *`port`*`, HttpdCgiParameter *&`*`p_attr`*`, bool &`*`free_addr`*`);`

This pure virtual method must be implemented by subclasses. This method should allocate memory (using `HttpdOpSys::Malloc`) sufficiently large to hold the desired subclass of `HttpdDiscoveredEndpoint` and construct the object.

If the object can not be allocated for any reason then `NULL` should be returned. Notice that *`p_attr`* and *`free_addr`* are passed by reference. For efficiency these structures (the address and the attributes) may be directly transferred to the endpoint. In this case the discovery client should not free them as it looses ownership. To prevent the discovery client from freeing *`addr`* this method should set *`free_addr`* to `false`. To avoid freeing the attribute list the method can simply point *`p_attr`* to `NULL` (or to a list of nodes that should be freed).

A typical implementation of this method for the `MyEndpoint` class would be as follows:

```
  void  *p_buffer;

  p_buffer = HttpdOpSys::Malloc(sizeof(MyEndpoint));
  if (httpd_rarely(p_buffer == NULL))
    return (NULL);
  HttpdCgiParameter *p_saved_attr = p_attr;
  p_attr    = NULL;
  free_addr = false;

  return (new(p_buffer) MyEndpoint(this, p_saved_attr, addr, port));
```

Notice that the parameters passed to the `MyEndpoint` constructor above all fall through to the `HttpdDiscoveredEndpoint` constructor.

DeleteEndpoint

`void` **`HttpdDiscoveryClient::DeleteEndpoint`** `(HttpdDiscoveredEndpoint *`*`p_endpoint`*`);`

This pure virtual method is responsible for releasing an endpoint that has been created (via `CreateEndpoint`) and displayed. The discovery client calls this method when *`p_endpoint`* is no longer needed.

Implementations should clean up any display or mention of the endpoint and then delete *`p_endpoint`*.

Note

It is important to keep in mind that all objects created by the `CreateEndpoint` method are destroyed by this method with one exception:

If the endpoint is never displayed (the `Display` method is never called) then the object is simply deleted rather than being passed to this method.

PurgeEndpoint

`void` **`HttpdDiscoveryClient::PurgeEndpoint`** `(HttpdDiscoveredEndpoint *`*`p_endpoint`*`);`

This method is called during shutdown of the client to remove endpoints. The default behavior, to simply call `DeleteEndpoint` can be overridden by subclasses to perform a more efficient “mass delete.”

For example: Consider a GUI client where deleting the endpoint has significant cost in terms of updating the display. This method could be overridden to avoid those updates if all the endpoints are being destroyed anyhow.

`HttpdDiscoveredEndpoint` Reference

Introduction

The `HttpdDiscoveryClient` object represents each discovered endpoint with an instance of this abstract base class. Pure virtual methods must be implemented in subclasses. A factory method must then be provided in `HttpdDiscoveryClient` to create subclasses of `HttpdDiscoveredEndpoint`.

Note

This class is only available if the portability layer provides the `HAVE_UDP_SOCKETS` feature.

Protected Methods

HttpdDiscoveredEndpoint

HttpdDiscoveredEndpoint::HttpdDiscoveredEndpoint (HttpdDiscoveryClient **p_owner*, HttpdCgiParameter **p_attr*, HttpdIpAddress *addr*, HttpdIpPort *port*);

This method constructs an endpoint object. The object is managed by *p_owenr*. The *p_attr*, *addr*, and *port* parameters describe the endpoint. These parameters are provided to the `HttpdDiscoveryClient::CreateEndpoint` method that is responsible for creating endpoint instances.

Update

virtual void **HttpdDiscoveredEndpoint::Update** (void);

This pure virtual method is called when any characteristics of the endpoint have changed. Typically this should result in refreshing the display of the endpoint.

It is important to realize that this method is called from a thread managed by the `HttpdDiscoveryClient`. Synchronization between other components may be necessary.

Display

virtual void **HttpdDiscoveredEndpoint::Display** (void);

This pure virtual method is called when a newly discovered endpoint is created and ready for display.

It is important to realize that this method is called from a thread managed by the `HttpdDiscoveryClient`. Synchronization between other components may be necessary.

~HttpdDiscoveredEndpoint

virtual **HttpdDiscoveredEndpoint::~HttpdDiscoveredEndpoint** (void);

The endpoint object is destructed when the endpoint is no longer discoverable. The destructor should remove any display of the endpoint.

Protected Data Members

mpAttributes

```
HttpdCgiParameter *mpAttributes;
```

This member holds a list of attributes from the endpoint. Some of the pairs in the list have a well defined meaning (such as `scheme` and `port`). Other pairs (those that begin with a leading _) are used for internal operation of the discovery service.

The endpoint is also free to put additional data describing it in various pairs. It is up to the client and server to figure out the meaning of these pairs (typically based upon device class).

`mpOwner`

```
HttpdDiscoveryClient *mpOwner;
```

This member holds a pointer to the discovery client that found (and is managing) this endpoint.

`mpUrl`

```
char *mpUrl;
```

This member is a string containing the URL of the discovered endpoint. It should not be modified by subclasses.

The Win32 Discovery Client

The Win32 discovery client uses the `HttpdDiscoveryClient` class to implement a discovery client application. The user interface of the discovery client is based upon HTML and an embedded browser object. The dynamic content is generated using the template engine.

This discovery client comes with full source code in `src/discovery/client/win32`. Almost all of the behavior is configurable by changing templates and resource scripts. Little knowledge of the Win32 *API* is required to modify the appearance of the client.

Compiling

Compiling the discovery client can be done using any compiler capable of producing Win32 executables. For Microsoft Visual C++ project and solution files are included. The project file (`w32dsclnt.vcproj`) will automatically build the `MSVC-DSCLNT` port of Seminole.

For other compilers the client builds similar to any other Win32 application. The settings in the included project file (described as follows) should be mimmicked for other compilers:

- The executable is linked statically. This helps ensure that the client runs on the widest range of systems as possible.
- Links against the `urlmon.lib`, `winmm.lib`, and `ws2_32.lib` import libraries.
- Builds the content template using SCPG.

Configuring the Client

All of the “configurable” portions of the client are in `src/discovery/client/win32/content`.

The most likely change necessary to the client is to add additional attributes to the display. Each endpoint is represented by a `DIV` HTML element. The body of that element is cleverly created by the template engine. The template for the body is named `src/discovery/client/win32/content/endpoint.thtm`. By default only a single attribute, `descr` (“description”) is supported. However additional attributes can be added. Consider the fragment that displays the description:

```
<table class="epattrs">
%{if:attr-exists name="descr"}%
 <tr>
  <th>Description:</th>
  <td>%{eval:attr-val name="descr" quote="html"}%</td>
 </tr>
%{endif}%
</table>
```

As evident from above, the `HttpdCgiSymbols` class is used to expose the attributes to the template with a prefix name of “attr”. Simply replicate the fragment producing the table row above for each attribute desired along with the descriptive name heading.

The parameters of the client are kept in `src/discovery/client/win32/content/client.cfg`. It is recommended that if you modify the parameters you change the value of the `CLIENT_PRODUCT` macro to something that uniquely idenfities that set of parameters. Most of the parameters are described by comments in this file.

Chapter 14. The Other Direction: An HTTP Client

The HTTP Client

Introduction

Seminole includes a simple HTTP client package that shares code with the server component. The client can be used for many things. For example a device could update its firmware in the background from a public HTTP server. Even more amibitious, combined with the discovery service and XML parser, a network of embedded devices could self-organize and communicate with one another using a form of RPC over HTTP.

The client is similar in design to the server component: Functionality can be traded for resources via compile-time options and settings. In fact many of the options that affect the server component (such as `XFER_BUF_SIZE`) also affect the client.

Performing HTTP Transactions

There are three main objects involved in performing an HTTP request. The request object, `HttpdClientFetch` contains all of the information specific to the particular HTTP resource being requested. This object is submitted to an instance of `HttpdClient`. Instances of this class contain all of the common resources required for performing HTTP requests. In most environments there is no need to have more than a single instance of the client. However if complete isolation between requests is desired then different requests can be directed to different client objects.

Once a request is processed the state of the transaction is represented by an instance of `HttpdClientTransfer`. Unlike the `HttpdClientFetch` object the transfer object holds resources needed only during the actual transfer. Instances of `HttpdClientTransfer` are created by the `HttpdClient` object and passed into the various methods of the `HttpdClientFetch` object.

It is normally not necessary to subclass the `HttpdClient` or `HttpdClientTransfer` classes. Application behavior is expected to be implemented in subclasses of `HttpdClientFetch`. In particular the `ResponseOk` method of the fetch object is called to process the response body.

Because of the limited amount of storage on most embedded devices HTTP responses can be processed in a “streamy” fashion. This is accomplished by subclassing `HttpdClientFetch` so that `ResponseOk` processes the data from the `HttpdOutboundTransfer` in an application-specific manner.

`HttpdClient` Reference

Introduction

The `HttpdClient` class implements a complete environment for performing HTTP requests. Although each HTTP transaction is self-contained it is expected that clients maintain a certain amount of state such as cookies, persistent connections, and cached redirects.

It is normally expected that only one instance of this class be created for all HTTP client operations. If it is of critical importance to isolate two different fetching environments then multiple instances of this class may be created.

Public Methods

Create

`int` **`HttpdClient::Create`** `(void);`

This method must be called before any fetching can be performed using this environment.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

SetSocketOptions

`void` **`HttpdClient::SetSocketOptions`** `(const char *const *`*`pp_options`*`);`

This method sets the platform-specific options used for socket connections.

SetCookieJarSize

`void` **`HttpdClient::SetCookieJarSize`** `(size_t `*`max_cookies`*`);`

This method sets the maximum number of cookies this client object will hold in its cookie jar.

Note

This method is only available if the INC_CLIENT_COOKIE_SUPPORT option is enabled.

SetProxyServer

`void` **`HttpdClient::SetProxyServer`** `(const char *`*`p_proxy_url`*`);`

This method configures the client to use a proxy server. The proxy server is specified in URL format. The URL allows a proxy to be used via alternative transports (e.g. SSL). Additionally the authentication information for the proxy may be part of the URL.

Note

This method is only available if the INC_CLIENT_PROXY_SUPPORT option is enabled.

NoProxyServer

`void` **`HttpdClient::NoProxyServer`** `(void);`

This method disables any prior use of a proxy server (configured via `SetProxyServer`).

Note

This method is only available if the INC_CLIENT_PROXY_SUPPORT option is enabled.

SetKeyRing

`void` **`HttpdClient::SetKeyRing`** `(HttpdClientKeyRing *`*`p_key_ring`*`);`

This method configures the client to use a key ring object. If client authentication is supported (INC_CLIENT_AUTH is non-zero) then a key ring object must be configured for the client before any fetches may be performed. The key ring object is responsible for maintaing cached authentication data. As such the key ring must have a lifetime that meets or exceeds the `HttpdClient` object. Additionally the key ring must not be changed while any fetches are in progress.

`Flush`

`void` **`HttpdClient::Flush`** `(void);`

This method flushes all cached data (except data being used by fetch operations in progress). The memory occupied by the cached data is released hence this method could be called to reduce memory pressure in other components using HttpdOpSys::Malloc.

It may also be useful to call this method during any kind of major reconfiguration event (e.g. IP address change) to avoid stale information from being used.

`HttpdClientFetch` Reference

Introduction

`HttpdClientFetch` objects represent a packaged request for an HTTP server. Instances of this object are submitted to a `HttpdClient` object for processing.

Note

It is required that applications subclass `HttpdClientFetch` to provide handlers for server responses.

Public Methods

`HttpdClientFetch`

`HttpdClientFetch::HttpdClientFetch` `(HttpdClient &`*`client`*`);`

The constructor creates the client fetch and associates it with a client context. The lifetime of *`client`* must exceed the lifetime of this request object.

`Fetch`

`int` **`HttpdClientFetch::Fetch`** `(const char *`*`p_url`*`, const char *`*`p_method`* `= "GET");`

This method performs a fetch using the associated client object. As the fetch progresses various protected methods of this class are called in an event-driven fashion.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). Additionally, HTTP-client specific error codes, such as HTTPD_ERR_TOO_MANY_REDIRECTS, may be returned.

`MaxRetries`

`void` **`HttpdClientFetch::MaxRetries`** `(HttpdClientCounter `*`v`*`);`

This method sets the maximum number of retries due to transient errors before the fetch is no longer retried by the client.

MaxRedirects

`void` **`HttpdClientFetch::MaxRedirects`** `(HttpdClientCounter` *`v`*`);`

This method sets the maximum number of redirects before the fetch is abandoned. This limit is in place to prevent an infinite loop of redirects. The default value is normally sufficient and comes from the RFC.

MaxLoginAttempts

`void` **`HttpdClientFetch::MaxLoginAttempts`** `(HttpdClientCounter` *`v`*`);`

This method sets the maximum number of authorization challenges that are not unlocked before the fetch is no longer retried by the client.

RetryDelay

`void` **`HttpdClientFetch::RetryDelay`** `(unsigned long` *`msec`*`);`

This method sets the delay time between retrying requests. The thread performing the fetch is suspended for this time period to avoid excessive network traffic in the event of network failure.

BodySource

`void` **`HttpdClientFetch::BodySource`** `(HttpdClientRequestBodySource` *`*p_source`*`);`

For HTTP methods where the request includes an entity body (e.g. `POST`) the body is provided by a class that implements the `HttpdClientRequestBodySource` interface.

If *`p_source`* is NULL (the default value if this method is not called) then the request body comes from a `HttpdContentSink` obtained via the `RequestBodySink` method.

To improve performance applications can implement the `HttpdClientRequestBodySource` interface to allow the client to decide the best way to send the request body.

BodyContentType

`void` **`HttpdClientFetch::BodyContentType`** `(const char` *`*p_source`*`);`

For HTTP methods where the request includes an entity body (e.g. `POST`) this method sets the content type of the request entity body.

RequestBodySink

`HttpdContentSink &`**`HttpdClientFetch::RequestBodySink`** `(void);`

If the request body source is NULL then this method may be called to obtain a sink object that can be used to store the request body.

Note

If a request body source object is employed this method should not be called.

Protected Methods

SendHeaders

int **HttpdClientFetch::SendHeaders** (HttpdClientTransfer &*xfer*);

This virtual method sends out the request headers for the request. Subclasses can override this virtual method to add additional headers. It is recommended that subclasses call the base class method after the custom headers are written.

This method returns `0` if successful or a a system dependent error value (see Table 4.1, “OS Abstraction Layer Error Codes”).

ProcessResponse

int **HttpdClientFetch::ProcessResponse** (HttpdClientTransfer &*xfer*);

This virtual method examined the `mStatus` field of *`xfer`* to handle the appropriate response from the server. The standard HTTP status codes are handled but this method may be overridden for additional error logging or the handling of non-standard server responses.

This method returns `0` if successful or a a system dependent error value (see Table 4.1, “OS Abstraction Layer Error Codes”).

ResponseOk

int **HttpdClientFetch::ResponseOk** (HttpdClientTransfer &*xfer*, HttpdOutboundTransfer &*body*);

This pure virtual method must be implemented by subclasses. If a `200` response is returned by the server this method is called. Of particular importance is the *`body`* parameter. This object can be used to process the returned entity body.

This method returns `0` if successful or a a system dependent error value (see Table 4.1, “OS Abstraction Layer Error Codes”).

`HttpdClientRequestBodySource` Reference

Introduction

The `HttpdClientRequestBodySource` interface is used to descibe the way that the request body is submitted during a fetch.

Public Methods

Traits

virtual int **HttpdClientRequestBodySource::Traits** (void); const

This method should return a combination of zero or more of the following flags describing the request body:

SRC_CHEAP	This indicates that the generation of the request body is “cheap.” A body is cheap to generate if little CPU

	time is needed to recreate it. The most obvious example of this is a request body that is simply sitting around in memory as a string. However depending on the application other sources (e.g. `HttpdXmlDomWriter`) may also be considered "cheap."
SRC_SIZE_KNOWN	This indicates that the size (in bytes) of the request body is known. If this flag is set then the client will call the `TotalSize` method to compute the size of the body. Again a perfect example of this would be a a request body that is a static string in memory.
SRC_LARGE_WRITES	This indicates that the implementation of the `Generate` method will perform mostly large writes. This is often set because the content is buffered rather than assembled on the fly. This flag should not be set if the request body is assembled with many small writes (e.g. building strings with `HttpdWritable::Printf`).

The default implementation returns `0`. It is expected that implementations of this interface optimize performance by returning the appropriate hints about their implementation.

`TotalSize`

`virtual size_t` **`HttpdClientRequestBodySource::TotalSize`** `(void);`

This method is only called by the client if the SRC_SIZE_KNOWN trait is present. When called it should return the size (in bytes) of the entity body for the request.

`Generate`

`virtual int` **`HttpdClientRequestBodySource::Generate`** `(HttpdWritable *`*`p_target`*`);`

This method is only called to generate the request body. It should write the request body to *`p_target`*. A a system dependent error value should be returned (see Table 4.1, "OS Abstraction Layer Error Codes").

Keep in mind that this method may be called multiple times by the client in the face of retries — especially if the SRC_CHEAP trait is present.

`HttpdClientBufferRequestBody` Reference

Introduction

The `HttpdClientBufferRequestBody` class implements the `HttpdClientRequestBodySource` interface for static, in-memory buffering. The advantage of storing the request body in memory is efficiency. Although keep in mind that if it is impractical to store the request body in memory (or that the request body be dynamically generated) then you should consider creating a custom implementation of `HttpdClientRequestBodySource` rather than this helper implementation.

Note

Only the methods in addition to those that are part of the abstract interface are documented here.

Public Methods

`HttpdClientBufferRequestBody`

`HttpdClientBufferRequestBody::HttpdClientBufferRequestBody` `(const void *`*`p_data`*`, size_t `*`size`*`);`

All that is needed to construct a buffered request body is a pointer to the data (*`p_data`*) and the size (in bytes) of the data (*`size`*).

`HttpdClientKeyRing` Reference

Introduction

An instance of `HttpdClientKeyRing` is used by the `HttpdClient` class to manage authentication data. In particular it caches authentication data based upon client URL to avoid round-trips during repetitive fetches.

This class may be subclassed to add support for additional authentication schemes or to obtain credentials in a non-standard way.

Public Methods

`HttpdClientKeyRing`

`HttpdClientKeyRing::HttpdClientKeyRing` `(size_t `*`max_keys`*`, long `*`max_key_age`*`, int `*`match_method`*` = HTTPD_CLIENT_KEY_MATCH_PATH_SUBSET);`

The constructor sizes the parameters of the key ring. The *`max_keys`* parameter controls the maximum number of cached keys. Keys are expired in LRU order when the cache fills up. For security reasons a keys should be periodically deleted after a certain amount of time. The *`max_key_age`* parameter controls the maximum duration (in seconds) that a key can exist in the cache.

The final parameter controls what keys are applied to what requests. Setting this parameter is a balance between security (preventing credentials leak out to a part of URL-space they shouldn't) versus efficiency (avoiding round trips because credentials are already known).

`match_method` may be set to one of the following values:

HTTPD_CLIENT_KEY_MATCH_EXACT	This is the most secure option. Every part of the URL (even the query string) must match before credentials are sent out.
HTTPD_CLIENT_KEY_MATCH_IGNORE_QUERY	This is similar to the above option except that the query string may be different. However the entire path (and host, port, and scheme) must match before credentials are given out.
HTTPD_CLIENT_KEY_MATCH_PATH_SUBSET	This option (the default) is what is specified by the RFC's: Paths that are “subsets” of the path (but with matching host, port, and scheme) are given credentials. The definition of a subset in this case ignores the final component in a path specifier (assuming them to be filenames within a directory tree).

HTTPD_CLIENT_KEY_MATCH_IGNORE_PATH	This is the least secure option. It requires only that the host, scheme, and port match before credentials are given out.

Create

`int` **`HttpdClientKeyRing::Create`** `(void);`

This method must be called before the keyring is used. It creates the internal objects used by the ring. Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”).

Protected Methods

GetAuthority

`int` **`HttpdClientKeyRing::GetAuthority`** `(HttpdClientFetch &`*`fetch`*`, const char *`*`p_header`*`, const char *&`*`p_authority`*`);`

This virtual method should return a pointer to the credentials needed for *`fetch`*. The `WWW-Authenticate` header line for the selected authentication scheme is also provided.

If credentials are available then *`p_authority`* should point to the credentials in the format of *`user`*`:`*`password`*.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If the authority must be dynamically allocated then subclasses may override `FreeAuthority` which the key ring will always call for all successful returns of `GetAuthority`.

The default implementation of this method obtains the authority from the fetched URL - for example:

```
http://user:pass@host:port/path/to/resource
```

GetKey

`int` **`HttpdClientKeyRing::GetKey`** `(HttpdClientFetch &`*`fetch`*`, const HttpdMimeParser &`*`mime`*`, HttpdClientKey *&`*`p_key`*`);`

This virtual method creates a key object for unlocking the fetch.

Upon success, `0` is returned; otherwise a system dependent error value is returned (see Table 4.1, “OS Abstraction Layer Error Codes”). If successful the key object will have its `Create` method called and it will be added to the key ring.

Subclasses may override this method to add additional calls to `IsScheme` to add support for alternative authentication schemes.

IsScheme

`const char *`**`HttpdClientKeyRing::IsScheme`** `(const HttpdMimeParser &`*`mime`*`, const char *`*`p_scheme`*`, size_t` *`scheme_len`*`);`

This static method searches for `WWW-Authenticate` heads with the specified scheme. The *`scheme_len`* parameter must be the length of the *`p_scheme`* string (not including the terminating null character).

If the scheme is present in *`mime`* then a pointer to the additional arguments in *MIME* line is returned. If the scheme is not offered then NULL is returned.

IsDefunct

`bool` **`HttpdClientKeyRing::IsDefunct`** `(HttpdClientFetch &`*`fetch`*`, const HttpdMimeParser &`*`mime`*`);`

This virtual method determines if a key object should no longer be used. The key ring maintains the LRU and aging properties. However this method may also cause a key to be removed from the cache by returning true.

The default implementation expires keys that have failed to unlock a request. Subclasses may override this method and return true in additional cases for added security.

Chapter 15. Integrating Seminole With An Application

Porting and Integrating Seminole

Introduction

Seminole is usable as a standalone webserver, but its primary purpose is to serve as one component of a whole embedded system. To that end, Seminole is provided with considerable support structure intended to ease its integration into an existing project environment. In particular, Seminole has:

- A generalized, modular build system
- A clear separation between portable and non-portable code
- Controlled resource utilization

As a result of these attributes, implementors will hopefully have more time to spend on their real goals rather than struggle with integration problems.

This section discusses the Seminole build system and portability layers in further detail.

Seminole compile-time parameters and options

For efficiency reasons many parameters are provided that control the various features of Seminole at compile time. Some of these options enable and disable features. Others control limits or parameters of normal operation. Options that enable and disable features typically begin with a `INC_` prefix. Setting a feature parameter to zero disables the feature and any non-zero value enables the feature.

These parameters are set when configuring the build system. All of these parameters have default values that are a best guess of what an ideal deployment environment is like. When putting Seminole into production it is important to review these parameters and adjust them accordingly.

- INC_QUEUED_HEADERS - This option enables support for queuing headers during the processing of a request to be delievered to the client when the response is sent. The default is `1`. Disabling this feature results in a minor reduction in code size.
- INC_SORTED_HEADERS - This option causes the *MIME* parser to sort the headers. If the headers are sorted they can be binary searched. The sorting is done using the `qsort()` library function. In certain obscure cases (i.e. a large number of *MIME* headers) enabling this option may result in improved performance. The default is `0`.
- INC_OPTIONS_METHOD - This option enables support for the HTTP`OPTIONS` method. If this feature is not required then disabling this option reduces code size. The default is `1`.
- INC_REQUIRE_HOST_HEADER - This option enables validation of the HTTP/1.1 behavior that a `Host` header must be specified. If strict validation is not required then disabling this option reduces code size. The default is `0`.
- INC_DYNAMIC_SERVER_NAME - This option makes the `Httpd::ServerName` method virtual so subclasses can override it. By default the method is static for maximum efficiency. The default is `0`. Disabling this feature results in a minor reduction in code size and greater performance.

- INC_FAST_URI_DECODE - This option makes the `UriDecode` family of functions faster at the expense of code size. The default is `0`.
- INC_TRACING - This option enables a run-time debugging facility that traces various operations in Seminole to help debugging during the integration phase. The default is `0`.
- INC_XML_TRANSCODE_SUPPORT - This option enables character set transcoding and automatic detection for the XML parser. Disabling this feature reduces code size and improves performance but requires that all XML content submitted to the parser be encoded as UTF-8. The default is `1`.
- INC_XML_NAMESPACES - This option enables namespace support for the XML parser. The default is `1`. Disabling this option results in a reduction of memory usage and code size.
- INC_XML_DOM_WRITE_CDATA - This option enables `XML_OPT_USE_CDATA` support in `HttpdXmlDomWriter`. The default is `1`. Disabling this option results in smaller code size.
- INC_WEBDAV_LOCKING - This option enables Class 2 support in `HttpdWebDAVHandler`. The default is `1`. Disabling this option results in smaller code size and reduced memory consumption.
- INC_WEBDAV_SHARED_LOCKS - This option enables shared locks for WebDAV. Shared locks are useful in some distributed authoring scenarios. This option only has meaning if INC_WEBDAV_LOCKING is enabled. The default is `1`. Disabling this option results in slightly smaller code size.
- INC_WEBDAV_TOKEN_TIMESTAMP - In an ideal world lock tokens are globally unique across all time. Enabling this option adds a timestamp component to lock tokens. If the entropy source backing `HttpdOpSys::Entropy` is strictly time based then enabling this option will not help. This option can be disabled if the `SESSION_NONCE_LEN` parameter is sufficiently large. This option has no effect unless INC_WEBDAV_LOCKING is enabled. Additionally this option also has no effect if the platform abstraction layer does not provide a real-time clock (i.e. `HTTPD_HAVE_CLOCK` is `0`). The default is `1`. Disabling this option gives a small reduction in code size and a minor increase in performance.
- INC_WEBDAV_QUOTAS - This open enables support for RFC 4331 (WebDAV Quotas). This symbol has no effect unless INC_FILE_QUOTAS is enabled. The default is `1`. Disabling this option gives a small reduction in code size.
- INC_SHUTDOWN - This option includes the code for graceful shutdown, including the `Httpd::Stop` method, socket force close logic, and memory cleanup logic. The default is `1`.
- INC_WRITE_BATCHING - This option batches writes to the HTTP socket to avoid lots of small packets. When enabled the `HttpdBatchWriter` is used to provide this behavior. It causes a small increase in code size in exchange for increased network throughput and efficiency. The default is `1`.
- INC_VERBOSE_RESPONSES - Disabling this option removes details from the HTTP responses. Normally the HTTP response includes a description of the response and (for error responses) a body indicating the error. Disabling this option makes the responses terse to save code space. The default is `1`.
- INC_MODIFIED_SINCE - Enabling this entry adds support for the `If-Modified-Since:` header. The default is `1`.
- INC_UNMODIFIED_SINCE - Enabling this entry adds support for the `If-Unmodified-Since:` header. The default is `1`.
- INC_SIMPLE_MODIFIED_SINCE - When processing conditional headers with timestamps the provided date must be parsed to implement the correct semantics. However a possible shortcut is to simply do an exact string comparison on the provided date with the last modified date. This results in smaller code size in exchange for less protocol conformance. Enabling this option enables this shortcut. The default is `0`.

- INC_BYTERANGE_SUPPORT - Define this entry to support partial content fetching (byte ranges) using the `Range:` header. The default is `1`.
- INC_PERSISTENT_CONN - Enable this option to include support for persistent connections. Persistent connections avoid TCP connection setup overhead for multiple requests. It is especially useful for pages with many images or references to other objects. It results in a code size increase. The default value is `1`.
- INC_OVERLOAD_PROTECTION - This option enables the overload protection feature of Seminole. This option only takes effect if INC_PERSISTENT_CONN is enabled and the platform has threads (HTTPD_HAVE_THREADS). The default value is `1` on platforms where this capability is supported.
- INC_ABORT_IDLE_SHUTDOWN - This option enables Seminole to close idle connections during a graceful shutdown. This option can only be enabled if INC_SHUTDOWN and INC_OVERLOAD_PROTECTION are enabled. The default value is `1` on platforms where this capability is supported.
- INC_ETAGS - ETags are a form of unique identifiers for HTTP objects. They can be used to assist in caching and conditional fetching. If it is expected that most browsers will support ETags they can be used in place of `If-Modified-Since:` header support. The default value is `1`.
- INC_FILE_QUOTAS - This enables support for quota information in the `HttpdFileSystem` class. The default value is `1`. Disabling this option if it is not needed will result in a minor code size reduction.
- INC_DIRECTORY_LISTS - Enable this option when directory indexing support is desired. This feature is typically only used for debugging or testing configurations and generally disabled for production use. Enabling this option results in additional code in the `HttpdFileHandler` class. The default is `1`.
- INC_LZRW1KH_COMPRESSION - Enable this option when the LZRW1/KH (codepoint `1`) compression algorithm is to be supported by the *ROM* file system. This compression engine uses very little memory and has a very small code footprint. It also performs very well and results in moderate space savings. The default is `1`.
- INC_LZJB_COMPRESSION - Enable this option when the LZJB (codepoint `3`) compression algorithm is to be supported by the *ROM* file system. This compression engine is very fast and efficient. The decompression process is very fast for the savings it gets. The default is `1`.
- INC_LZARI_COMPRESSION - This option includes support for the LZARI (codepoint `2`) compression algorithm is to be supported by the *ROM* file system. This compression engine requires more resources than the LZRW1/KH algorithm but may result in higher compression ratios. The default is `1`.
- INC_ROM_DIRECTORIES - If this option is enabled runtime support for directory indexes is enabled in `HttpdRomFileSystem`. Even with this feature enabled, SCPG must also be told to generate directory objects. The default is `1`.
- INC_ROM_SAFETY_CHECKS - If this option is enabled then the `HttpdRomFileSystem` package does more stringent checking of the content data. In most embedded systems this option can be safely turned off to reduce code size. The content data in these systems is usually stored in the same flash device as the code and is checksummed during startup. However in more complex systems where content packages can be loaded from other sources (such as disk files or plug-in modules) then it is wise to enable this option. The default value is `1`.
- INC_ROM_ATTRIBUTES - If this option is enabled then the `HttpdRomFileSystem` will support per-file attributes. With this option disabled the code size impact for the *ROM* filesystem may be smaller. This is because the attributes are encoded using URL encoding and enabling attributes will link in the CGI parser.

- INC_ROM_FAST_COMPRESSED_PUSH - Enable optimized pushing of compressed content stored in a `HttpdRomFileSystem`. Disabling this option results in reduced performance at the expense of code size. The default value is `1`.

- INC_ROM_FAST_RANGE_PUSH - This option includes an optimization that improves the handling of byte ranges for uncompressed files in a `HttpdRomFileSystem`. Enabling this option results in slightly larger code size. The default value is `1`.

- INC_TEMPLATE_MIME_TYPES - This option enables the ability to override the default *MIME* type emitted by `HttpdFSTemplateShell`. Disabling this feature results in miniscule performance and code space savings. The default value is `1`.

- INC_CHARCLASS_PATTERN_MATCH - Enabling this feature adds character class support to the `HttpdUtilities::MatchPattern` function. Disabling this feature saves code space at the expense of functionality. The default value is `1`.

- INC_HASH_PJW - Enabling this option changes the hashing algorithm used by `HttpdUtilities::Hash` to the “P.J. Weinberger” hash function. This hashing function results in a better distribution at the expense of slightly increased CPU consumption and code size. For very large numbers of CGI parameters in a `HttpdCgiHash` enabling this may be a performance win on 32-bit processors. If this option is disabled (the default) then a normal addative hash function is used instead.

- INC_MODIFIABLE_FILESYSTEMS - Enabling this option enables the methods in the filesystem *API* for writable file systems. The default value is `1`. Disabling this option reduces code size.

- INC_BACKGROUND_SESSION_PURGE - This option enables time-based purging of unused session objects. Without this option, instances of `HttpdSessionManager` will only destroy a session object when there is no additional room for a new session. When this option is set, a background thread periodically examines the session table and deletes any sessions that have an idle time exceeding a predefined threshold.

  For applications with very large session objects this option can help reduce the contention for memory by other tasks using HttpdOpSys::Malloc. Using this option also results in a significant increase in security because it makes replay attacks more difficult. However, in order to use this option the target platform must support threads and the ability to note the passage of time. The default value is `1`.

- INC_FAST_MD5 - If this option is enabled the code implementing the `HttpdMD5` class will be larger but much faster. The default is `0`.

- INC_FAST_SHA1 - If this option is enabled the code implementing the `HttpdSHA1` class will be larger but much faster. The default is `0`.

- INC_SECURE_MD5 - If this option is enabled then the `HttpdMD5` class will wipe all working data from memory when complete. This results in increased security at the slight expense of additioal CPU utilization. The default value for this option is `1` and it is recommended that this option not be disabled.

- INC_SECURE_SHA1 - If this option is enabled then the `HttpdSHA1` class will wipe all working data from memory when complete. This results in increased security at the slight expense of additioal CPU utilization. The default value for this option is `1` and it is recommended that this option not be disabled.

- INC_LOW_STACK_PRESSURE - For performance reasons Seminole allocates several large objects on the stack of the servicing thread. For systems where stack space is limited this option can be enabled. When enabled this option causes these objects to be allocated from the heap. The downside of this is that this reduces performance and increases code size (slightly). It also opens up a window of failure if the heap does not have enough free space to allocate request objects. In that case, the client socket is simply closed without even sending back a response. This may also result in slightly increased heap fragmentation. The default value is `0`.

- INC_LOW_HEAP_PRESSURE - This option causes Seminole to use less heap memory at the expense of code size and execution time. In low memory environments enabling this option is a worthwhile optimization. The default value is `1`. This option may also be set to `2` or `3` for a more aggressive reduction in heap usage at the expense of performance. The higher the value the less pressure placed on the heap.
- INC_LOW_CODE_PRESSURE - This option causes Seminole to use less code size at the expense of execution time. This option should be enabled when the expected HTTP load is light and code size footprint is important. The default value is `1`.
- INC_MULTIPLE_TRANSPORTS - This option enables the optional transport-selection mechanism. With this feature enabled Seminole allows several different network transport protocols to be selected at runtime. The most important reason to enable this option is to support SSL (which must be enabled separately using the INC_SSL option). The default value is `0`.
- INC_SSL - Enable SSL protocol support. This option should only be enabled if support is provided on your particular target. This feature requires the INC_MULTIPLE_TRANSPORTS feature also be enabled. There may be additional parameters that can be configured in the portability layer when this option is enabled. The default value is `0`.
- INC_IPV6_SUPPORT - Enable support for IPv6. This option should only be enabled if this protocol is supported by your target platform. The default value is `0`.
- INC_BUFFER_OUTPUT - There are some cases where the length of the response is unknown. In these cases the `HttpdDynamicOutput` class is used to control how the output is delivered. Enabling this option tells the `HttpdDynamicOutput` class to buffer the output in dynamic memory in order to avoid closing persistent connections. This results in increased use of run-time memory usage but allows greater network throughput. The buffering is performed by the HttpdContentSink class. The value of this option has no effect if the INC_PERSISTENT_CONN feature is not enabled. The default value for this option is `1`.
- INC_CHUNK_OUTPUT - Similar to the INC_BUFFER_OUTPUT feature this option allows the `HttpdDynamicOutput` class to use the “chunked” transfer encoding if possible. This encoding is only supported with HTTP/1.1 clients, older clients will be handled by other methods (if available) or by closing persistent connections. The chunked encoding is performed by a the `HttpdChunkedSink` filter class. The value of this option has no effect if the INC_PERSISTENT_CONN feature is not enabled. The default value for this option is `1`. The default value is `1`.
- INC_BUFFER_OVERFLOW_RECOVERY - If the INC_BUFFER_OUTPUT option is enabled and the server runs out of temporary storage for the content an attempt is made to recover by shutting down persistent connections when this happens. In the event of this failure the content is still delivered by sending what has been buffered until the heap was exhausted then continuing with data transmission and finally closing the connection when complete. Enabling this option results in a slight increase in code size. Enabling this option has no effect if the INC_PERSISTENT_CONN or INC_BUFFER_OUTPUT options are disabled. The default value is `1`.
- INC_FAST_STRING_SINK - Enabling this option causes `HttpdStringSink` to allocate reserve memory to avoid each `Write` call resulting in a call to `Realloc`. This option can also reduce heap fragmentation. It increases code size slightly but also results in a gain in CPU utilization. The amount of pre-allocated memory is bounded by the STRING_GROW_SIZE parameter. The default value is `1`.
- INC_BASIC_AUTH - This option enables the basic HTTP authentication mechanism in the `HttpdAuthenticator` class. If no authentication mechanisms are enabled then requests to authenticate a request will always fail with a `HTTPD_RESP_UNAUTHORIZED` (`401`) status. The default value is `1` and should not be disabled in most circumstances.
- INC_DIGEST_AUTH - This option enables HTTP digest authentication. Digest authentication prevents passwords from being transported in the clear across the network. This results in an increase in code size.

This feature uses the session manager therefore the security precautions involved in using the session manager (good quality entropy and background session scrubbing) should be employed if possible. The default is `1`.

Setting this value to `2` will result in presenting digest authentication ahead of basic authentication (when both are enabled). This normally is against the recommendations of the RFC's. However many versions of the Firefox® browser will always choose the first scheme presented rather than the strongest scheme presented (as is required). Setting this value to `2` works around this bug; at the expense of compatability.

- INC_DIGEST_AUTH_URL_MATCH - Digest authentication requires that the URL be a component of the password hash. The URL is passed in an attribute rather than taken from the actual HTTP request line. This prevents proxies from modifying it and thus corrupting the password hash.

  If this option is enabled then Seminole verifies that the URL is equivilent to the request URL. This verification requires additional processing overhead but reduces the effectiveness of man-in-the-middle attacks. Disabling this option reduces code size and memory requirements at the expense of security. With this option disabled passwords are still never sent across the network in the clear — providing secrecy. The nonce mechanism ensures that replay attacks will not succeed. However if there is a chance that an attacker can intercept (and modify) the request then this verification step prevents this.

  The default value is `1`.

- INC_PASSWD_BLINDING - This option reduces the risk of timing attacks when comparing passwords. HTTP basic authentication is particularly vulnerable to this kind of attack although in many environments the attack is difficult to mount. Enabling this option causes the `HttpdAuthenticator::SecureStrEqu` method to always take the same amount of time comparing strings irrespective of the contents of those strings. Enabling this option reduces performance and increases code size. In security critical environments this option should be enabled. For performance critical applications this option can be disabled. The default value is `1`.

- INC_CONDITIONAL_HINTS - Enabling this option adds conditional hints if the C++ compiler supports them. Conditional hints help to identify `if` statements that are used for infrequent events (such as total failure cases) and cause the compiler to generate code that is more efficient for the frequent cases. With the GNUGCC™ compiler this is accomplished with the `__builtin_expect` built-in. The default value is `1`.

- INC_ALIASING_HINTS - Enabling this option adds pointer aliasing hints if the C++ compiler supports them. Aliasing hints help identify pointers that don't alias and avoid reloading values from memory. This reduces code size and increases efficiency. The default value is `1`.

- INC_ALLOCATION_CACHING - Enabling this option causes some classes to cache memory allocations to avoid a performance penalty. For systems with very limited amounts of memory disabling this option reduces resource consumption at the expense of performance. The default value is `1`.

- INC_ALLOCATION_CACHE_PURGE - Enabling this option enables the ability to purge all allocation caches when the system is low on memry. The default value is `1`.

- `OVERLOAD_ABORT_RETRIES` - This parameter controls how many times overload protection will attempt to release an idle thread before failing the new request instead. This parameter only has any effect if INC_OVERLOAD_PROTECTION is enabled. The default value is `1`. In some high-volume configurations (especially when running under a POSIX operating system) it may be advantageous to increase this value.

- `OVERLOAD_ABORT_SLEEP` - This parameter controls how long the acceptor task sleeps (in milliseconds) during retries when in overload. This parameter only has any effect if INC_OVERLOAD_PROTECTION is enabled. The default value is `130`.

- INC_CLIENT_CONN_POOL - This option enables connection pooling in the `HttpdClient` class. Enabling this option makes HTTP fetching much faster at the expense of code size and memory (and socket) consumption. The default value is `1`.

- INC_CLIENT_REDIR_CACHE - This option configures the size and desire for redirect caching in `HttpdClient`. If the value is `0` then permanent redirects are never remembered. Otherwise this controls the number of slots in the cache. Disabling the cache reduces the code size of the client. The default value is `16`.

- INC_CLIENT_PROXY_SUPPORT - This option enables support for HTTP proxies. Disabling this option reduces code size and improves fetch performance. The default value is `1`.

- INC_CLIENT_COOKIE_SUPPORT - This option enables support for HTTP cookies on the client side. Disabling this option reduces code size and improves fetch performance. The default value is `1`.

- INC_CLIENT_AUTH - This option enables authentication support in the client. Disabling this option reduces code size. The default value is `1`.

- INC_CLIENT_AUTH_BASIC - This option enables basic authentication support in the client. The default value is `1`.

- INC_CLIENT_AUTH_DIGEST - This option enables digest authentication support in the client. Disabling this option reduces code size. The default value is `1`.

- INC_CLIENT_COOKIE_BUFFERING - This option prevents the HTTP client from writing cookie headers while holding the mutex of the cookie jar. This prevents the somewhat unlikely case of a socket write stalling for a long period of time while holding the cookie jar mutex and affecting other fetches. Enabling this option increases code size and memory consumption slightly. It is probably a good idea to enable this option if there are many threads performing simultaneous fetches against a single client object. The default value is `0`.

- `CLIENT_MAX_CONN` - When INC_CLIENT_CONN_POOL is enabled this parameter controls the maximum number of connections that will be pooled per instance of `HttpdClient`. The default value is `130`.

- `CLIENT_MAX_CONN_PER_HOST` - When INC_CLIENT_CONN_POOL is enabled this parameter controls the maximum number of connections to a single host per instance of `HttpdClient`. This value must be smaller than `CLIENT_MAX_CONN`. The purpose of this limit is not to reduce the resource consumption of `HttpdClient` — that is the purpose of `CLIENT_MAX_CONN`. This constant is to avoid causing excessive resource consumption on other servers. The default value is `5` as recommended by RFC 2616.

- `CLIENT_HASH_BUCKETS` - When INC_CLIENT_CONN_POOL is enabled this parameter adjusts the size of the hash table used to look up pooled connections. It should be a prime number. Larger values increase the memory footprint of `HttpdClient`. The default value is `17`.

- `CLIENT_AUTH_KEY_BUCKETS` - Authentication credentials are cached to help reduce round trip times. Increasing the value reduces search overhead at the expense of memory. Setting this value to less than `2` will disable hashing (thus saving code and data space). The default value is `19`.

- `CLIENT_AUTH_CNONCE_LEN` - This parameter controls the length of the client nonce when using HTTP digest authentication. Increasing this value will increase security provided sufficient entropy is available. The default value is `16`.

- `TMPL_MAX_INCL_DEPTH` - This is the maximum number of file includes that may be nested when processing a template with the `HttpdFSTemplateShell` mechanism. The default value is `16`. There is no cost to increasing this limit except that beyond a certain point of nested includes may cause a stack overflow.

- `TMPL_MAX_SYM_LENGTH` - This is the maximum length of a symbol identifying a template operation. This value only applies to the actual name of the action, not the associated attributes. The maximum theoretical value is `127` and the default value is `126`. Lowering this value may save a few bytes of storage during template processing.

- `CGI_HASH_SIZE` - This parameter controls the number of buckets in the `HttpdCgiHash` class. The more buckets the quicker parameters can be found (at a cost of space). In general, the number of buckets should be a prime number. But this is not a hard-and-fast rule if memory is in short supply. The default value is `7`.

- `BITSET_WORD_SIZE` - This parameter selects the word size used by classes such as `HttpdBitSet`. Ideally the word size selected should be the most efficient for the machine to manipulate. Setting this incorrectly only results in reduced performance. However computing the correct setting takes into account a number of factors such as the compiler, CPU, and memory bus width.

Value	**Word Type**
`0`	unsigned int
`1`	unsigned long
`2`	unsigned short

The default value is `1`.

- `MAX_MIME_ENTRIES` - This parameter controls the maximum number of name-value pairs on an incoming request. The default value is `48` pairs.

- `MAX_INPUT_LINE` - This is the maximum length of a line on an incoming request in bytes. The default value is `1024`.

- `XFER_BUF_SIZE` - This parameter controls the transfer buffer size. This buffer size is used in several places where the copying of data from one source (such as a file) to another (such as a socket) is performed. The default value is `1024` bytes. Increasing this value may result in increased efficiency in some usage scenarios.

- `MIN_BATCH_WRITE_SIZE` - This parameter controls the smallest write that will be sent to the target writable associated with a `HttpdBatchWriter`. If this constant is set to `0` then all writes will be at least `XFER_BUF_SIZE` bytes in size. Setting this parameter to a non-zero value results in a slight code size increase but may reduce CPU consumption. The default value is `0`.

- `SINK_BUFFER_SIZE` - This parameter is the size of the data buffers that HttpdContentSink uses to buffer dynamically generated content. The default value is `512` bytes. Setting this value higher results in fewer memory allocations by the content sink but may result in increased memory consumption.

- `STRING_GROW_SIZE` - If the INC_FAST_STRING_SINK option is enabled then this parameter controls the amount of extra memory allocated by the `HttpdStringSink` object to reduce allocation overhead and fragmentation. If the INC_FAST_STRING_SINK option is disabled then this parameter has no effect. The default value is `512` bytes but could be made smaller if memory is tight with little performance impact.

- `CHUNK_OUTPUT_SIZE` - This parameters controls the maximum size (in bytes) of a segment of data when using the HTTP/1.1 chunked transfer encoding. Chunked transfer encoding is provided by the `HttpdChunkedSink` class. The default value is `1024`.

- `DIGEST_WINDOW_SIZE` - This parameter controls the number of pending nonces that are allowed in a digest authentication session. Increasing this parameter can help avoid digest authentication failures in very high request rates at the expense of memory. The default value is `128`.

- `DIGEST_MAX_AGE` - This parameter controls the maximum age of a digest authentication session (in seconds). If the background session scrubbing option (INC_BACKGROUND_SESSION_PURGE) is not enabled then this parameter has no effect. The default value is `900`.

- `DIGEST_MAX_SESSIONS` - This parameter controls the maximum number of digest authentication sessions at any one time. Raising the number of sessions allows more simultaneous clients at the expense of memory. The default value is `32`.

- `DIGEST_BATCHSIZE` - This parameter adjusts the number of session objects processed for aging during a single cycle. If the background session scrubbing option (INC_BACKGROUND_SESSION_PURGE) is not enabled then this parameter has no effect. The default value is `8` sessions.

- `DIGEST_CYCLETIME` - This parameter determines how often the background scrubber is woken up to scrube digest authentication sessions. The default is `60000` milliseconds.

- `FIRST_TIMEOUT` - This parameter represents the amount of time to wait (in seconds) after the establishment of a TCP connection for the first line of the request. This timeout serves two purposes. First, it serves to prevent persistent connections from lasting indefinitely. Second, it functions as a denial-of-service attack recovery mechanism. The timeout for the first request line is distinct because often times a client may take longer to send the initial line of the request. Once the request is generated the headers typically are sent quickly. Therefore this timeout should be larger than the `MIME_TIMEOUT` parameter. The default value is `160`.

- `MIME_TIMEOUT` - This parameter represents the amount of time to wait after the first request line is received for the transmission of the *MIME* name-value pairs. Unlike `FIRST_TIMEOUT` expiration of this timer would more likely indicate a network or protocol error rather than an overloading condition. The default value is `30` seconds.

- `CGI_TIMEOUT` - This parameter represents the maximum amount of time that the CGI parsing classes (such as `HttpdCgiParameter` and `HttpdMultipartCgiParser`) will wait for new incoming data. The default value is `200` seconds.

- `ACCEPT_FAIL_DELAY` - With some TCP/IP implementations the `HttpdSocket::Accept` operation can return an error code that indicates a non-fatal but transient error condition (resource limits, insufficient buffers, link connectivity problems, etc). This is the amount of time the acceptor thread should wait before calling accept again to process additional incoming connections. It is mainly used as a throttling timer to reduce the likelyhood of similar setup errors ocurring in succession. The default value is `500` milliseconds.

- `MAX_REQUESTS_PER_CONN` - When persistent connections are enabled this parameter controls the maximum number of requests that can be processed on a single connection. Even if the `FIRST_TIMEOUT` timer is not exceeded the maximum number of requests can never be exceeded. This option has no effect if the INC_PERSISTENT_CONN feature is disabled. The default value is `5` requests.

- `MAX_PASSWD_LENGTH` - This is the maximum length of a password when using the `HttpdAuthenticator` framework. The default is `48` characters.

- `MAX_REALM_LENGTH` - This is the maximum length of the realm when using `HttpdAuthenticator` framework. The default value is `48` characters.

- `SESSION_NONCE_LEN` - This parameter controls the number of random characters included in a session identifier included to prevent spoofing. Making this value longer is only likely to increase security if a the target platform has a good source of entropy. The default value is `32` characters.
- `MAX_MODAL_DEPTH` - This parameter is the maximum number of widgets that may be stacked within a `HttpdWidgetStack` object. The default value is `4`.
- `WIDGET_TABLE_GROW_SIZE` - This value is the number of “widget slots” to allocate when the current widget manager (`HttpdWidgetManager`) runs out of available tracking slots. The default value is `64`. The default should be appropriate for almost all circumstances however reducing this value may result in decreased memory consumption when using the application framework.
- `GIF_HASH_SIZE` - This parameter controls the compression engine in the `HttpdGif87aRenderer` object. Adjusting this value may result in bandwidth reduction in exchange for higher memory consumption during image creation. The default value is `5003`.
- `PMATCH_MAX_RECURSION` - This parameter limits the default recursion depth of the `HttpdUtilities::MatchPattern` method. Patterns that involve “globbing” matches can make this routine recursive. To guard against stack exhaustion (especially when the pattern string comes from an untrusted source, such as a CGI variable) a limit is placed on the maximum depth of the recursion. The default value is `16` but this may have to be altered depending on available stack space.
- `DSC_TRANSMIT_JITTER` - This parameter controls the amount of “jitter” artificially added to beacons transmitted by `HttpdDiscoveryServer`. The purpose of the jitter is to prevent a storm of broadcasts which may result in a high collision rate on some networks. This value is in milliseconds and defaults to `90`.
- `DSC_BEACON_JITTER` - This parameter controls the amount of “jitter” artificially added to unsolicited beacons transmitted by `HttpdDiscoveryServer`. This value is in milliseconds and defaults to `20`.
- `DSC_REQUEST_JITTER` - This parameter controls the amount of “jitter” artificially added to request beacons transmitted by `HttpdDiscoveryClient`. The purpose of the jitter is to prevent a storm of broadcasts which may result in a high collision rate on some networks. This value is in milliseconds and defaults to `5000`.
- `DSC_ENDPT_HASH_SIZE` - This parameter controls the number of buckets in the hash table that `HttpdDiscoveryClient` uses to hold `HttpdDiscoveredEndpoint` objects. The default value is `171`.
- `DSC_CLIENT_SCRUB_BATCH` - This parameter controls the number of endpoints examined by `HttpdDiscoveryClient` during a scrubbing cycle. The number of endpoints is limited to a subset to avoid execessive bursts of CPU consumption. The default value is `16`.
- `XML_TAG_BUF_SIZE` - This parameter controls the number of characters that the internal name buffer of `HttpdXmlNode` holds. If a tag name exceeds this length a dynamically allocated buffer is used. The default value is `32`.
- `XML_PARSE_NODE_CACHE_SIZE` - This parameter controls the maximum number of node objects to cache during a parse. These nodes are temporary objects that are used during parsing. To avoid continuous memory reallocation the XML parser caches these objects as it parses the document. The higher this value the more temporary memory consumed during parsing. The default value is `24`.
- `XML_NAMESPACE_CHUNK_SIZE` - This parameter determines the size of space reserved when allocating storage for namespace URL's. To reduce the number of heap allocations memory is allocated in chunks when a namespace must be stored. As many URL's are packed into a single allocation as possible. Setting this parameter tunes the amount of internal versus external fragmentation. Setting this

parameter to `0` disables chunking and allocates each namespace URL in a separate object, reducing code complexity instead. The default value is `140`.

- `XML_NAMESPACE_MAX_SEARCH` - When a namespace URL is encountered the XML parser scans the list of namespaces that have already been seen to see if the string space can be shared. This sharing reduces memory consumption at the expense of CPU time. If this parameter is set to `0` then the entire list of namespace URL's is searched. Otherwise this value specifies the maximum number of URL's the XML parser will examine in an attempt to save space. The default value is `0`.

- `MACRO_MAX_ARGS` - This parameter controls the maximum number of arguments to a macro in the `HttpdMacroProcessor` class. The default value is `32`.

- INC_CACHING_FILE_DATA_SOURCE - If this option is enabled the code implementing the `HttpdFileDataSource` class will employ a caching mechanism. This caching mechanism is useful when the associated `HttpdFile` does not have any caching performed by the underlying operating system and system calls are expensive. For operating systems with no memory protection and tight memory requirements disabling this option may result in a reduction of code size and memory consumption. The default is `1`.

- `FILE_DATASRC_CACHE_SIZE` - This parameter controls the size of a cache block in the caching version of `HttpdFileDataSource`. It is recommended that it be a power of 2. See File Data Source Caching for details. The default value is `4096` bytes.

- `FILE_DATASRC_MAX_CACHE_BLOCKS` - This parameter controls the maximum number of cache blocks that a `HttpdFileDataSource` may hold. See File Data Source Caching for details. The default value is `16` blocks.

- `FILE_DATASRC_HASH_BUCKETS` - This parameter controls the number of hash buckets in the caching version of `HttpdFileDataSource`. It is recommended that it be a prime number. See File Data Source Caching for details. The default value is `7` buckets.

- `FILE_DATASRC_MAX_PINNED` - This parameter controls the maximum number of pinned cache blocks in a `HttpdFileDataSource`. See File Data Source Caching for details. The default value is `4` blocks.

- HAVE_GLOBAL_CONSTRUCTORS - This option determines if global constructors are supported on this platform. Some embedded systems do not call global constructors or call them at the wrong time. This problem can be worked around at a minor cost in efficiency by setting this option to `0`. If this option is set to `0` then the runtime environment must provide properly synchronized delayed construction of static locals. The default value is `1`.

The Seminole Build System

Overview

Like the rest of Seminole, the build system is oriented towards embedded systems developers. Using the build system is completely optional. If your project has a radically different build system from what Seminole offers, it can be used instead with little effort. The major features of the Seminole build system are:

- The compilation tools and build environment are easily changed.

- Multiple targets are easily selected and built.

- The output files for each target are isolated so one can have Seminole built for a variety of targets at the same time.

The Seminole build system consists of two logical pieces. The first piece is a Perl script (called `buildit`) which recursively descends the source tree and applies commands to the files within based on the particular quirks of the host operating environment. `buildit` obtains pertinent information about the host environment from the second logical piece of the build system, the ports files. These files, so named because they are located in the `ports` subdirectory of the distribution, simply contain Perl code which is directly evaluated at the beginning of the build process. One ports file may include other files, to form a hierarchy of overlapping definitions so that common information can be centralized in a single file. Thus, factors common to all POSIX systems are contained in the `POSIX` ports file, which is then included by other ports which are POSIX-like environments.

Once the system-specific declarations in the ports file(s) have been evaluated, `buildit` proceeds to look for a rules file called `Buildfile` in the current working directory. This file contains a series of Perl subroutines which are analogous to the names of **make** targets. If no target name is provided on the command-line, the `default` subroutine is executed (typically all source files will be rebuilt if the corresponding object file is stale). This process is often recursive, with `buildit` descending into each subdirectory in the tree and executing the same subroutine.

Extensive documentation on the Perl language is available on the World Wide Web or in book form, and no attempt will be made here to duplicate it.

Performing a Build

The first step in performing a Seminole build is to select a ports file that most closely describes your particular environment. Choosing an appropriate ports file greatly reduces the amount of customization that must be done to generate a successful build. See Table 15.1, “Standard Ports Files” for a list of ports files provided with the distribution.

Table 15.1. Standard Ports Files

Port	Description
`Linux`	GNU/Linux with GCC
`OpenBSD`	OpenBSD with GCC
`FreeBSD`	FreeBSD with GCC
`Solaris-gcc`	Solaris® with the GCC compiler
`Solaris-CC`	Solaris® with the Sun Studio compiler
`MSVC`	Windows NT® with Microsoft Visual C++
`Watcom`	Windows NT® with Watcom C++
`MacOSX-gcc`	MacOS X® development kit (using Apple-provided GCC
`MacOSX-xlc`	MacOS X® development kit (using IBM's VisualAge C++)
`Tornado`	Wind River Systems' Tornado® development kit for VxWorks®
`eCos`	eCos embedded development environment
`QNX6`	The QNX distributed operating system
`Android`	The Linux based Android platform

When using ports such as `Tornado` and `eCos` which represent cross-compilation environments, it is critical to properly identify the target architecture and binary type by assigning the appropriate variables.

In most situations the best method is to take advantage of the hierarchical nature of ports files and create a customized file matched to your specific needs. The file containing your customizations would then include other ports files "above" it in the hierarchy; for example, a new ports file intended to build Seminole for an ARM-based embedded target running the eCos® operating system might be called `eCos-arm`. Since all the default definitions acceptable and only a few parameters are being changed, `eCos-arm` might look something like this:

Example 15.1. Using Inherited Definitions in a Ports File

```
$ECOS_ARCH = 'arm-elf';
$CPUENDIAN = 'little';
$ECOS = '/ecos-tree';

# Load the definitions for eCos targets.
definitions(samepath($DEFINITION_FILE, 'eCos'));
```

Once a usable ports file has been selected or customized as described above, the build process can begin. Depending on the host operating system, the `buildit` script must be invoked slightly differently. In POSIX-like environments, something similar to the following command may be executed from the root directory of the Seminole distribution:

`./buildit ports/`*`PORTFILE`*

`PORTFILE` should specify the desired ports file to be evaluated. If your Perl interpreter is located in a non-standard directory (`/usr/bin/perl` is assumed), you will need to change the path manually in `buildit` with a text editor or create a filesystem link.

In Microsoft Windows NT® and related environments, the build system should be invoked as follows:

`PERL` **`buildit ports\`** ***`PORTFILE`***

In this case, *`PERL`* should be the name and/or path of your system's Perl interpreter (`perl` will probably work if you are unsure).

When complete, the results will be in `built/`*`PORTFILE`*. The header files needed by the client application are placed in `include` and the libraries needed are placed in `lib`. In addition, most ports provide a standalone binary that can be used to verify the build of Seminole.

The files that are created during the build are referred to as the SDK (Software Development Kit). Once Seminole is compiled the files from the SDK are the only things required to build applications that use Seminole. The location of the SDK can be changed by overriding the build system configuration variables in the `ports/Seminole` file.

After building Seminole for the first time, one should be verify that the resulting libraries are functional. When targeting a POSIX (or more generally, UNIX®-like) target, this can be done by invoking the Seminole executable.

Once proper operation is verified with a web browser, you have successfully built Seminole.

Build System Internals

The build system contains three major components. The first is a support library of subroutines that do basic operations useful for building software. The most important of these is `cx`. This function executes an external command such as running the C++ compiler or linker.

Another important support routine is `stale`. This routine takes two array references as arguments. Both are lists of files, the first argument describing a list of target files and the second argument describing a list of source files. If the source files are newer than the target files or the target files do not exist then this routine returns true.

The `definitions` is used to load a file containing auxiliary Perl code for building Seminole. This is how the hierarchy of port files are implemented. The top-level port file imports lower-level port files using this function. In turn those lower-level files may import other files. The filename provided to the `definitions` subroutine should be an absolute path. In general all of the build system files are in the same directory and the `samepath` subroutine can be used to make a full path name. The global variable `$DEFINITION_FILE` contains the full path name of the build file requested on the `buildit` command line.

The `subdirs` subroutine performs a recursive build in the named subdirectories. The file `Buildfile` in the root of the Seminole source tree shows how the `src` directory is built. The `optsubdirs` subroutine is similar to `subdirs` except that non-existant directories are non-fatal. The role of `Buildfile` is explained below.

The `cx` and `stale` subroutines can be combined to produce make-style functionality with the `and` operator. For example:

```
stale(['foo.o'], ['foo.cpp', 'foo.h', 'common.h']) and
   cx('g++', '-c', 'foo.cpp');
```

In the above example, `foo.cpp` will be recompiled if `foo.o` does not exist or is older than the source file or the two header files. This functionality is used to implement the second major component of the build system: the target routines. These are subroutine references that point to routines that perform basic tasks used to build Seminole:

- `$target_link` - Perform a link step if to produce a loadable or executable image from object modules and libraries.
- `$target_xform` - Generate the object file name of a source file. In most cases this is a straight transformation such as changing a file extension.
- `$target_binary` - This subroutine computes the output file name used for the link step.
- `$target_cxx` - This subroutine compiles a source module to an object module by invoking the C++ compiler for the target system.
- `$target_addlib` - This subroutine adds one or more object files into a library (or archive) file. If the library file does not exist it should be created.
- `$host_c_xform` - Translate the name of a C source module to the file name needed to run the C program. The host tools always consist of a single C source file that is compiled (using the host compiler) to run as a standalone program.
- `$host_compile_c` - Invoke the C compiler for the host system.

The final component of the build system is the `buildit` script. This script drives the build system by preparing the environment and then executing the build instructions for each source directory. The build environment is prepared by loading definition files that define subroutines and variables that are used during the build phase.

Once the environment is prepared the directory tree is traversed and the `Buildfile` files are loaded. The `Buildfile` defines one or more named subroutines that perform a build action. Depending on the build target (if any) specified on the `buildit` command line the subroutine in the `Buildfile` is executed to perform the build step.

It is best to think of the subroutines in the `Buildfile`s as make targets (such as `all` or `clean`). These subroutines use the variables and subroutines defined in the build environment to accomplish their tasks. In fact the traversal of the directory tree is done using the built-in subroutines `subdirs` or `optsubdirs` and is not explicitly part of `buildit`.

Building Seminole using an alternative build environment

There is nothing particularly unique that Seminole's Perl-based build system does that can't be done with other build environments. Sometimes in a large project the Seminole build system is not appropriate. For these situations it is possible to build the source code for Seminole with **make** or an IDE.

One important point to underscore is that even if the build system is not used it is still necessary to have an operating Perl environment if the Host Tools are used.

The first step in building Seminole manually is to copy over the files ending in a `.in` extension to a file with the same base name but ending in `.h`. Then edit the `.h` files replacing the text within the `${…}` directives with the actual configuration values desired.

The second step is to create an appropriate set of build options for your compiler. It is important to add any of the directories that contain header files to the include path of the compiler.

Finally using whatever is appropriate for your host environment add the `.cpp` files to your build list. Only add the files that are appropriate for your target; i.e. do not add the files under `src/targets/Win32` if you are targetting a POSIX system.

The above approach puts all of the effort of building Seminole and your application on the developer. A different alternative is building Seminole is a hybrid approach. For example. Build Seminole using the provided build system then simply reference the built libraries and header files in a completely separate build environment.

Another approach is to integrate the Seminole build system into the existing environment. For example, an external “Makefile” project can be created within Microsoft Visual Studio® and call the Seminole build system. This kind of integration gives the benefits of “push button” builds without requiring in depth knowledge of either build environment.

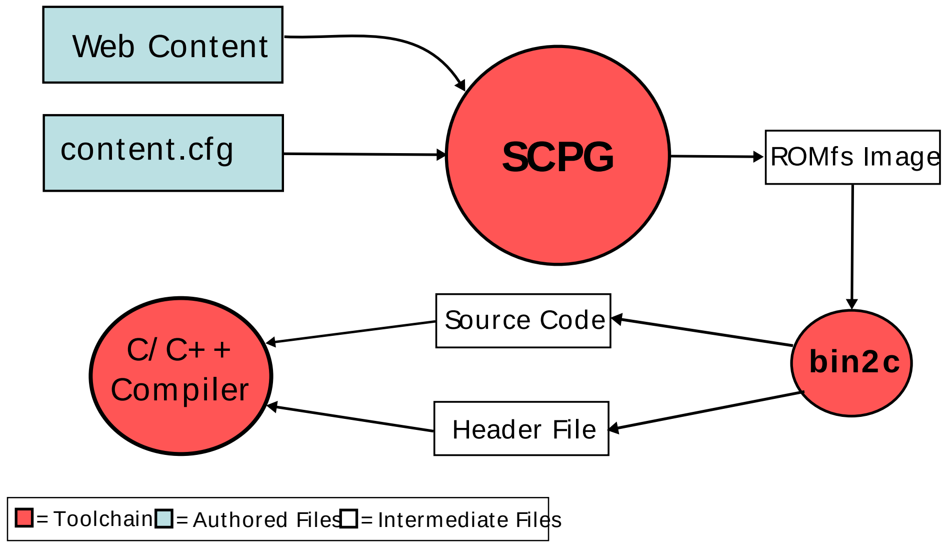
Whichever approach is taken for the build machinery it is important that the correct compiler options are set for efficient code generation. Seminole does not use templates, exceptions or run-time type information. So these features can be disabled if the code it calls (user-provided handler code) does not. This will often result in a performance increase and a code size reduction. Choosing the right level of optimization (size versus speed) is also important and depends on the performance requirements and the capabilities of the target platform. Often the only way to correctly choose these options for a complex project is by experimentation.

Toolchain

Regardless of what build system is being used builds are performed by calling tools to operate on files. Some of these tools, such as the C/C++ compiler are not part of Seminole. Other tools, such as **mimegen** or **bin2c** are part of Seminole. Understanding how the various tools interact is what this chapter is all about.

A very common usage of the toolchain is to take web content and package it for use by the *ROM* filesystem. This is typically done by embedding the content directly in the system image. This configuration is described by the following figure:

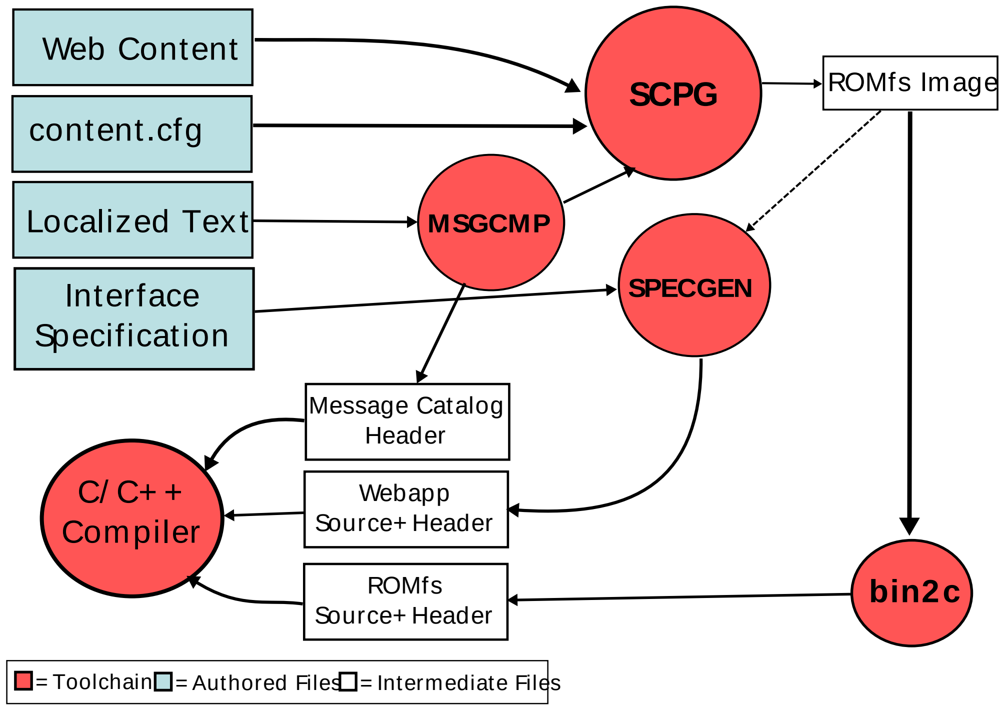
Figure 15.1. Toolchain for Combining Content in the System Image



As you can see each tool tries to do a single task. In this case SCPG processes the web content and builds an image that the *ROM* filesystem code can mount. To get this data inside the system image the **bin2c** command converts this binary file to a source and header file. The source file is compiled in to the system image and the header file allows the binary data to be referenced.

A much more complex orchestration of tools is involved in building an application using the application framework.

Figure 15.2. Toolchain for Building a Web Application



As you can see from the above figure there are quite a few tools involved with a complex web of dependencies. The **msgcmp** tool is used for localization support and builds a binary "string package" file that is included in the *ROM* filesystem along with the content. The output of SCPG is then utilized by **specgen** to verify the interface specification is correct.

The output of **specgen** which contains the support code for the web application is combined with the *ROM* filesystem image and compiled into a single system image.

Using SSL

Seminole supports SSL if the underlying socket stack provides an implementation of the SSL protocol. The OpenSSL [http://www.openssl.org/] library is a free implementation of SSL and works on most platforms. The Seminole build system currently has OpenSSL [http://www.openssl.org/] support in the following ports: `Linux`, `OpenBSD`, `Watcom`, `MacOSX-gcc`, `MacOSX-xlc`, `FreeBSD`, and `MSVC`. Adding support to other build environments should be relatively simple, by following the example of these ports files.

To use the SSL library on, check the appropriate ports file for a commented-out set of directives referencing OpenSSL [http://www.openssl.org/]. Follow the instructions and uncomment the necessary Perl directives. This will enable SSL support in all subsequent builds.

There are special considerations for SSL in the Win32 environment. Since there is no standard location for the OpenSSL [http://www.openssl.org/] library, the location of OpenSSL [http://www.openssl.org/] must be assigned to the `OPENSSL_DIR` variable in the ports file being used. If the SSL stack is to be dynamically linked using DLL files, then the variable `OPENSSL_DLL` should also be defined to a non-null value.

Once enabled, it may be necessary to supply additional parameters to the SSL protocol stack. This is done by passing an array of strings to the `Httpd::Start` method. A typical set of options for OpenSSL [http://www.openssl.org/] would be similar to this:

```
pem:server.pem
sock:ssl
rand-file:4096,/dev/urandom
```

The most important line is:

```
sock:ssl
```

That parameter tells the transport selector (enabled via INC_MULTIPLE_TRANSPORTS) to use the SSL socket type instead of the standard TCP type.

The line

```
pem:server.pem
```

instructs the SSL socket to use the private key and PKI certificates from the file `server.pem`.

Note

The options file and the certificate data can be generated with the makecert utility.

Using MatrixSSL

The OpenSSL [http://www.openssl.org/] library is quite large and not suited to many embedded systems. Seminole includes support for the MatrixSSL [http://www.peersec.com/matrixssl.html] commercial SSL library from PeerSec. This library is smaller and easier to work with in embedded environments.

To enable MatrixSSL™ support the `MATRIX_SSL_PATH` variable must point to the location of the library (called `libmatrixssl.a` on most POSIX-like systems) and header files. The `INC_SSL` and `USE_MATRIX_SSL` configuration variables also need to be enabled in the p variables must be defined in the port file. The fragment in the build file should be similar to the following:

```
MATRIX_SSL_PATH = 'location/of/matrixSSL';
config(INC_SSL => 1,
       USE_MATRIX_SSL => 1,
       INC_MULTIPLE_TRANSPORTS => 1);
```

Once enabled, MatrixSSL support is activated by passing a set of options to the `Httpd::Start` method. The most important of which is `sock:ssl` which informs Seminole to use the SSL transport layer.

When using MatrixSSL the following additional options should be specified in addition to `sock:ssl`:

- `cert:`*`certificate`* - This configures the server certificate as the inline contents of the string.
- `key:`*`key`* - This configures the encryption keys of the server as the inline contents of the string.

For more information on configuring MatrixSSL, see the documentation for the `matrixSslReadKeysMem`.

Operating Environment Abstraction Layers

Introduction

The Seminole portability layer insulates the portable code from the following system specific areas:

- Memory allocation
- Multi-tasking
- File system access
- Networking
- Entropy (randomness) generation
- Time accounting

Of the services listed above only the memory allocation and networking services must be provided by implementations of the portability layer. Multi-tasking in Seminole is optional and if a platform does not support it then requests are processed serially. Native filesystem access is rarely needed because the HttpdRomFileSystem class provides an efficient self-contained file service optimized for web serving.

The memory allocation abstraction is likely to be of particular interest to embedded systems programmers. Since all dynamic memory allocation in Seminole is obtained through the *API* in `HttpdOpSys` it provides a convenient place to constrain or measure memory usage. This is especially important to prevent denial-of-service attacks from disabling the primary function of an embedded system.

Another place where operating systems differ widely is in their support of networking models. Within Seminole these differences are abstracted by the `HttpdSocket` class which manages the creation of socket objects that implement the `HttpdSocketInterface` interface.

The interface described by `HttpdSocketInterface` resembles that of BSD sockets. On any platform offering the sockets interface, porting the provided TCP (and optionally SSL) implementations ought to require little or no modification. Other transports supporting at least the general concepts of addressable communications endpoints and bi-directional data flow should be capable of abstraction within via `HttpdSocketInterface`. If a given platform's interface does not support byte-oriented I/O, some adaptation will be required in the `ReadN()` and `WriteN()` methods.

The reference implementations for POSIX and Win32 include support for shutting down sockets in operation through a control interface. This makes these implementations more complex than a naive implementation but makes server shutdown more responsive. In many cases there is no need for a graceful shutdown concept and the implementation of the TCP socket abstraction can be greatly simplified.

The multi-tasking interface assumes very little in terms of operating system functionality. It is assumed that mutual exclusion and a flag-style semaphore are available for task synchronization. Care has been taken to not require recursive mutexes although they cause no harm. Event semaphores are one-shot flag style semaphores. Thus either counting or binary semaphores may be used. It is not required that one thread waits for the termination of another. This is explicitly done (using event semaphores) when necessary.

The generation of entropy is needed by a few Seminole support classes. Better quality randomness results in a more secure system although if a good entropy source is not present a few system variables (current thread id, stack pointer, tick counter) with some processing can result in acceptable entropy. If SSL is employed then it is important that entropy provided to the SSL protocol stack is of very high quality and that entropy should also be exposed via the `HttpdOpSys::Entropy` method.

Seminole is not critical about the resolution of the system clock as error timeouts are the main use of a system clock. If a real-time clock is present then Seminole can utilize the current time in the HTTP responses although it is not necessary.

Adding New Abstraction Layers

The process of porting Seminole to a new target or host platform generally involves one or both of the following steps:

- Create a new build framework definition file in `ports/`, named after the host environment, and populate it with appropriate **make** meta-commands and variables to carry out the Seminole build process. This step is only necessary if the included build system is used.
- Create the `sem_sys.h`, `sem_syssock.h`, and `sem_sysconfig.in` files. The `sem_sys.h` header must define the HttpdUint8, HttpdUint16, HttpdUint32, HttpdIpAddress (unless HTTPD_HAVE_BULKY_SOCKET_ADDRESSES is non-zero), and HttpdIpPort types. The `sem_syssock.h` file should define a class named `HttpdIpAddressObject` if HTTPD_HAVE_BULKY_SOCKET_ADDRESSES is non-zero
- The `sem_sys.h` header contains the declarations of the `HttpdOpSys`, and the optional `HttpdMutex`, `HttpdEventSemaphore` classes. The `sem_syssock.h` header file defines the `HttpdTcpSocket` and optional `HttpdSslSocket` classes.
- Within whatever source file structures are necessary the implementation of these classes should be written using the native services of the target platform.

As noted earlier it may often be simpler to use an existing portability layer as a skeleton for a new port. There are many ways to architect the portability layers and care has been taken such that the examples cover as many of the approaches as possible. When contemplating a new port it is often instructive to read the existing portability layers to gain insight into what techniques are ideal for the new target.

It is also helpful to realize there is nothing "magical" about the portability layer. It is simply a set of routines that Seminole uses to support its operations. Sometimes the best approach is to see where a particular portability routine is called and design it for the way in which it will be called. For example the POSIX and Win32 portability layers provide two different multi-tasking models. One approach simply creates a new thread in response to a call to `HttpdOpSys::Fork`. This approach is simple and compact but can be a performance limitation. So an alternative implementation is provided that keeps a pool of ready worker threads that can be assigned to a task rather than created when `Fork` is called.

Although this alternative implementation is useful for POSIX and Win32 it may not be for some real-time operating systems where thread creation is very fast or already pooled. Many of the decisions in the portability layer are like this.

Essentially, the job of the abstraction layer implementor is to rewrite the classes mentioned earlier in this section, keeping to the public interfaces defined elsewhere in this document. The total effort involved in writing or porting each method is highly dependent on the nature of the new target and what services it offers to applications. The more familiar an implementor is with the target platform the easier this task is likely to be. Keep these facts in mind when planning a port.

There are also some parameters that are defined by the portability layer that determine the capabilities of the target platform. These parameters may be adjustable depending upon the specifics of the portability layer. For example, floating point support is often an optional concept on many embedded operating systems but the concept is always present on the POSIX targets. Therefore the POSIX portability layer always enables floating point support. These parameters can be used in your own application code to make it more portable. When implementing a new portability layer the following macros must defined in `sem_sys.h`, `sem_sysconfig.h`, or in the configuration of the compiler:

- `HTTPD_OS_NAME` - This macro expands to a quoted string that is the name of the host operating system or target. It is sent in the `Server:`*MIME* header line in response to an HTTP request. If this macro is undefined then no additional data is appended to the server name.
- `HTTPD_HAVE_BIG_ENDIAN` - This macro should evaluate to a non-zero (true) value if the target platform uses "big endian" byte ordering.
- `HTTPD_HAVE_REENTRANT_LIB` - This macro should be defined to a non-zero (true) value if the runtime library supports the reentrant versions of the standard ANSI C library.
- `HTTPD_HAVE_NATIVE_FILE_SOURCES` - This macro should be defined to a non-zero (true) value if the platform has a native file system and that native file system is exposed via the `HttpdOpSys::NativeFileSystem` method.
- `HTTPD_HAVE_BULKY_SOCKETS` - This macro should be defined to a non-zero (true) value if the size of the `HttpdTcpSocket` is significant. Under normal circumstances a `HttpdTcpSocket` holds a handle to a TCP socket. However some very simple TCP implementations all of the protocol state is held within the `HttpdTcpSocket`. In these cases enabling this option when the INC_LOW_STACK_PRESSURE option is enabled the `HttpdTcpSocket` is stored on the heap instead of on the stack.
- `HTTPD_HAVE_CLOCK` - This macro should be defined to a non-zero (true) value if the target has a clock capable of keeping the current date and time.
- `HTTPD_TIMESTAMP_IS_TIME_T` - This macro should be defined to a non-zero (true) value if the `HttpdOpSys`::TimeStamp type is represented using time_t. If this is not known then it is always safe to leave this option `0` which results in some micro-optimizations being disabled.

- `HTTPD_HAVE_THREADS` - This macro should be defined to a non-zero (true) value if the target platform supports simultaneous execution of two or more threads.
- `HTTPD_HAVE_OPSYS_REALLOC` - If the porting layer has its own native implementation of `HttpdOpSys::Realloc` then this macro should be defined to a non-zero (true) value. If this macro is defined to `0` then a pre-defined implementation of `HttpdOpSys::Realloc` that uses `HttpdOpSys::SafeRealloc`.

There are several parameters that are common more than one portability layer implementation. Many of these parameters adjust the internal operation of the portability layer. These are configurable through the generalized configuration mechanism of the build system:

- `HCLOSE_TIMEOUT` - This parameter represents the maximum amount of time to allow the TCP connections managed by Seminole to exist in the `FIN_WAIT_2` state. The default value is `3600` seconds. If sockets are a limited resource on the target platform the lowering this value may help in recovery from failed network connections.
- `AVOID_LINGER` - Enabling this option causes the portability layer to manually keep closed sockets that still have undelivered data open. This option is necessary because on most platforms the SO_LINGER option does not work at all or do what is needed for an HTTP server. Disabling this option should only be done on TCP stacks that specifically implement the expected behavior with regards to HTTP pipelining.
- `CONN_BACKLOG` - This value is sent as the second argument to the `listen()` system call. Under most operating systems this controls the depth of the queue that holds incoming but not processed TCP connections. The default value for most targets is `5`.
- `MAX_HEAP_USAGE` - If non-zero this places a hard limit on the amount of memory Seminole will use. Attempts to keep more than this amount (in bytes) allocated will result in memory allocation failures that Seminole will handle gracefully.
- `MAX_THREAD_USAGE` - If non-zero this places a hard limit on the number of threads that Seminole may spawn.
- `MAX_WAIT_FREE_TASK` - If `MAX_THREAD_USAGE` is non zero (or thread pooling is enabled) then Seminole will wait for up to `MAX_WAIT_FREE_TASK` milliseconds for the number of running threads to go below `MAX_THREAD_USAGE`. If `MAX_WAIT_FREE_TASK` is `0` then spwaning a thread when over quota results in an immediate failure return from `HttpdOpSys::Fork` which Seminole handles gracefully.
- `INC_THREAD_POOLING` - If this symbol is defined to a non-zero (true) value then threads will be reused as requests come in (within certain limits). For devices that function primarily as web servers, significant gains in performance can be obtained with this option; depending on operating system and hardware platform.
- `MAXIMUM_FREE_THREADS` - If INC_THREAD_POOLING is enabled then this is the maximum number of free threads to keep on the free list. Any additional threads will eventually be scrubbed.
- `MONITOR_POLL_TIME` - If INC_THREAD_POOLING is enabled then this is how often (in seconds) the pool of threads is trimmed to no more than `MAXIMUM_FREE_THREADS` entries. Decreasing this delay will make Seminole less able to adapt the thread pool to sporadic load however it may reduce resource usage (at the expense of CPU time).
- `INITIAL_THREAD_COUNT` - If INC_THREAD_POOLING is enabled then this is how many threads will initially populate the free list.

Additionally, the provided POSIX target layer makes use of a few preprocessor symbols which may be relevant in adapting it to new POSIX variants. These are listed below and may be set as described previously.

- HTTPD_USE_SINGLE may be defined to a non-zero (true) value to cause Seminole to have only one thread of execution; each `Fork()`'ed function runs to completion from the caller. This option is very useful when debugging because many debuggers do not deal well with threads.

- HTTPD_USE_CLOCK_GETTIME tells the portability layer that the operating system supports the `clock_gettime` system call. Most modern UNIX® systems do. Define this macro to `0` if this system call is not available or broken on the target platform.

- HTTPD_HEAP_DEBUG can be used to enable a debugging version of the memory interface in the `HttpdOpSys` class (see HttpdOpSys::Malloc). The debugging version verifies pointers, collects statistics about memory usage, and tracks memory leaks.

- HTTPD_USE_POLL should be defined to a non-zero (true) value if the target platform supports `poll()`. The `poll()` system call is far more efficient than the `select()` system call. If your system supports `poll()` then this symbol should be defined to make use of it.

- Setting USE_INTEGER_DIFFTIME to `1` avoids the use of the ANSI`difftime()` routine. This routine subtracts two time_t values and returns the difference in seconds as a floating point value. In many embedded systems floating point is undesirable. In most POSIX environments the time_t type is understood to be a count, in seconds, from a well defined epoch. In this case a simple integer subtraction can be performed instead of calling `difftime()`.

- Enabling INC_PRIORITY_ADJUST causes the priority of the threads to be adjusted when they are performing work. This feature is only possible if the pthreads implementation supports scheduling parameters. The POSIX_PRI_ACCEPT, POSIX_PRI_WORKER, and POSIX_PRI_SCRUBBER parameters, if not set to `0` control the scheduling priority of the task.

- Enabling USE_USLEEP causes the `HttpdOpSys::TaskSleep` method to use the `usleep()` system call. If USE_USLEEP is disabled then `select()` or `poll()` is used to delay execution instead.

- If INC_OVERLOAD_PROTECTION is enabled the POSIX portability layer requires an unused signal to interrupt idle but in-use threads. This signal is configured with SOCK_INTR_SIG. The default value is SIGUSR1. If your application uses SIGUSR1 for itself then change this parameter to an available signal that has no significant side effects.

- If INC_OVERLOAD_PROTECTION is enabled the POSIX portability layer may encounter a small race condition window in sending the signal to the blocked thread. The race is won by retry with a delay. The SOCK_ABORT_POLL parameter controls how long (in milliseconds) to sleep during retries. This should be set to a very low time value. The default value of 120 milliseconds should be sufficient. For high volume processing this timer can be reduced to improve throughput at the (slight) expense of CPU time.

- If SOCKET_SEND_TIMEOUT is greater than zero the send timeout of the underlying TCP stack is set to this value (in milliseconds).

Extending Seminole

Introduction

For programmers used to the functionality offered by HTTP servers such as Apache, Seminole's sparse feature-set and intimate *API* may seem like liabilities. Embedded systems programmers, on the other hand, will recognize those attributes as its greatest strengths. However, since it is neither possible nor desirable to predict every application which will incorporate Seminole, extending Seminole's functionality is a natural requirement in most cases. This chapter attempts to answer basic questions implementors are expected to raise, and provide a starting point to begin making changes in a logical way.

As discussed in the introductory sections of this manual, Seminole's design is quite modular. Great care was taken to abstract mundane protocol issues and hide irrelevant complexity within a set of clean interfaces. For these reasons, the difficulty of adding code to Seminole ranges from trivial (adding handlers or tweaking isolated routines) to slightly involved (porting the environmental abstraction layers to a radically different system, or heavily altering a core class).

Adding Handlers

Basics

By far the easiest way to extend Seminole is by adding new handlers. In many cases, the combination of a few custom handlers with the utility classes provided out of the box are sufficient to successfully integrate Seminole into your application or system.

As discussed briefly in the first chapter, Seminole services incoming requests by calling each registered handler until one willing to service the request is found, or all handlers have declined. When handlers are instantiated during Seminole's initialization, one of the constructor arguments represents a URL-space prefix used to discriminate requests for which that handler is responsible. The handler chain maintained by Seminole is sorted in decreasing prefix order, such that the longest match for any given request will always be taken.

Two handlers are provided with Seminole because of their common necessity; the `HttpdRedirector` class provides for HTTP redirections, and the `HttpdFileHandler` class implements a POSIX filesystem reader suitable for providing file service through Seminole. Both classes are fully documented in the preceding chapters.

All Seminole handlers are derived from the abstract `HttpdHandler` class, which provides appropriate linkages for the handler chain as well as any common handler methods. Classes derived from `HttpdHandler` must provide their own version of the virtual method `Handle()`, which serves as the primary entry point and request dispatch routine for a handler. Needless to say, in threaded environments, multiple instances of a given handler may be processing requests simultaneously.

Conventions

Most conceivable types of handlers will need to follow certain conventions. Since they must be registered with a certain URL prefix, and the checking of each request's URL takes place within every handler (until a handler dispatches the request), it is necessary for each installed handler instance to know its own prefix. The prefix is given as a constructor argument when the handler object is instantiated, and subsequently used to populate the member `mpPrefix` (inherited from `HttpdHandler`). Care should be taken in deciding how to do this; if the implementor can be certain that the pointer passed into the constructor will remain valid throughout the handler's lifetime, then a simple assignment will suffice. Otherwise, a call to `StrVCat()` (or anything else allocating dynamic storage) may be appropriate, as shown in Example 15.2, “A Skeletal Handler”).

`mpPrefix` is used by the `IsMe()` method (also inherited from `HttpdHandler`) to determine whether a request is appropriate for the calling handler. Typically the first thing done in each handler's `Handle()` is a call to `IsMe()`, passing a pointer to the current request being examined by `Handle()`; if the return value is NULL, then there is no further work to be done, and `Handle()` returns false, indicating that Seminole should continue to traverse the handler chain looking for a better candidate. Otherwise, `IsMe()` returns the request path with the prefix portion removed (so in the case of a handler registered to service “`/abc/def`”, the handler-specific portion of “`/abc/def/ghi`” would be “`/ghi`”). Note that there are certain scenarios where the longest match according to `IsMe()` is not, in fact, the most desirable. For example, given the URL "`/productlogo.jpg`", with handlers installed on the prefixes "`/`" and "`/product`", requests for "`/productlogo.jpg`" would actually be accepted by the handler registered on "`/product`", which in this case is unlikely to be the intended behavior. For these situations, the `IsMyPath()` method is provided in `HttpdHandler`. The calling conventions are the same, except that `IsMyPath()` takes a second argument, a const char specifying the path delimiter (which is usually "/", but the option is left to the implementor). `IsMyPath()` is somewhat more discriminating; it checks the request path and ensures that prefix actually maps to one segment in a hierarchical URL rather than merely a matching substring of a path belonging to another handler. Thus, in the previous example, since "`/productlogo.jpg`" does not contain "`/product`" as a path segment (as a URL like "`/product/info.html`" would), that handler would receive a NULL return value from `IsMyPath()` and allow the succeeding handlers to pick up the request. The typical scenario in which this problem might be encountered is serving files from a hierarchical filesystem, but many other possibilities exist. Both interfaces are provided for flexibility, and they are almost equivalent in terms of processing cost.

Once the question of acceptance or rejection is settled, the handler is responsible for processing the request in its entirety; after `Handle()` returns, the client network connection is shut down with no further work done.

Example 15.2. A Skeletal Handler

```
class NewHandler : public HttpdHandler
{
  // some public data members
public:
  NewHandler(const char *p_prefix);
  virtual ~NewHandler();

  virtual bool Handle(HttpdRequest *p_request);
  // some public functions
};

// …

NewHandler::NewHandler(const char *p_prefix)
{
  mpPrefix = HttpdUtilities::StrVCat(p_prefix, (const char *)0);
}

NewHandler::~NewHandler()
{
  HttpdOpSys::Free((char *)mpPrefix);
}

// …

bool NewHandler::Handle(HttpdRequest *p_request)
{
  const char *p_req_path = IsMe(p_request);

  // or perhaps this instead, if we're worried about the IsMe()
  // ambiguities mentioned above:
  //
```

```
    // const char *p_req_path = IsMyPath(p_request);

    if (p_req_path != NULL)
    {
      // this request is a match.

      // perform some processing …

      return (true);
    }
    else
      return (false); // this request doesn't match our prefix.
  }
```

CGI Processing

Most non-trivial handlers needing to accept input from a dynamic source will need to use the Common Gateway Interface, CGI. By making use of the provided `HttpdCgiParameter` class, handler implementors can quickly take care of retrieving CGI input and concentrate on their real work.

CGI permits an arbitrary number of name/value pairs to be passed to an HTTP server, either as part of the URI (known as URL-encoded parameters) or as request data via the `POST` method. HttpdCgiParameter::ParsePostData handles `POST` data, and expects a pointer to the current `HttpdRequest`. HttpdCgiParameter::ParseUriString parses parameters encoded in a URI string, and expects a string pointer to same. In either case, the parsing method returns a pointer to a `HttpdCgiParameter` object representing the start of a chain. This chain is a singly linked list with each node containing the name and value (specifically, an HttpdPair member called `mPair`) of each parameter found. The `mpNext` member serves as a pointer to the next `HttpdCgiParameter`, or NULL if the end of the chain has been reached.

Example 15.3, “Parsing CGI Parameters” shows a chunk of code that might be found in a custom handler. In this simple example, we can accept up to three URL-encoded arguments, named `foo`, `bar`, and `baz`. Each of these in turn is used to set integer variables in the handler, and presumably to control behavior somewhere else. If the arguments were to be passed via the HTTP `POST` method, one need only call HttpdCgiParameter::ParsePostData instead with a pointer to the `HttpdRequest` being handled. In all other respects the example would be the same.

Example 15.3. Parsing CGI Parameters

```
// …

int foo = 0;
int bar = 0;
int baz = 0;
HttpdCgiParameter *paramhead, *paramcur;

// …

// Retrieve all the CGI parameters that were encoded in our URI,
// previously saved into *p_uri by Handle().

paramhead = paramcur = HttpdCgiParameter::ParseUriString(p_uri);
```

```
if (paramhead != NULL)
{

  // If we're here, then we must have gotten something.  Iterate
  // through the parameter list.

  while (paramcur != NULL)
  {

    // We only care about foo, bar, and baz.  Other parameters
    // are ignored (a real handler might throw a syntax/usage
    // error instead).

    if (strcmp("foo", paramcur->mPair.mpKey) == 0)
      foo = atoi(paramcur->mPair.mpValue);

    if (strcmp("bar", paramcur->mPair.mpKey) == 0)
      bar = atoi(paramcur->mPair.mpValue);

    if (strcmp("baz", paramcur->mPair.mpKey) == 0)
      baz = atoi(paramcur->mPair.mpValue);

    // Follow the forward link to the next parameter.
    paramcur = paramcur->mpNext;
  }

  // All done, so free the parameter list.
  // It's important that the *head* of the list be freed, obviously.

  HttpdCgiParameter::FreeList(paramhead);

}

// …
```

Dynamic Memory Allocation

Introduction

Seminole performs all memory allocation through an *API* provided by the `HttpdOpSys` class. The *API* is similar to the `malloc` package provided by ANSI C.

For efficiency reasons, when objects are allocated with the `new` operator, it is always done using “placement new”. In addition, vector construction and destruction are not used. These choices were made to allow Seminole to be comparable in code size to straight C code with as much efficiency as possible.

Creating Objects

Objects that are created on the heap are defined using a custom version of `new`:

```
class MyObject
{
public:
  void *operator new(size_t, void *p_buffer)
  { return (p_buffer); }

  void operator delete(void *p_buffer)
  { HttpdOpSys::Free(p_buffer); }
};
```

To instantiate a version of that object requires a sequence like the following:

```
void *p_buffer = HttpdOpSys::Malloc(sizeof(MyObject));
if (p_buffer == NULL)
  return (HttpdOpSys::ERR_OUTOFMEM); // Handle error

MyObject *p_obj = new(p_buffer) MyObject; // Construct object.
```

This approach allows error handling to happen before the actual allocation. In the case of Seminole, out of memory handling is critical to building robust systems. In most cases, the web interface is exposed to a (potentially) hostile network. Denial of service attacks against the web interface should not result in a failure of unrelated parts of the system.

By putting the error handling up front (before the constructors are called) it is easier to avoid partial construction of objects. The allocations of several objects can be batched and then the entire action can fail if insufficient memory exists before any constructors (which may modify state) are called.

Chapter 16. Host Tools

Introduction

Seminole attempts to do as much processing on the host as possible. Embedded systems are typically limited in space and speed and adding a web interface to an existing embedded system should have the smallest impact possible.

Most of the host-based tools are written in Perl. They were specifically coded to run on Perl `5.005_03` or better. Some of the tools (such as the compressors) are written in C (not C++) and need to be compiled with a C compiler for the host environment — not with the target's C++ compiler.

Note

It is important to make sure that the build system uses the correct tools for the correct modules when using a cross-compiler.

Host Tool Input Format

All of the host-based tools use a common preprocessor mechanism that is similar in function to the C preprocessor; providing conditional compilation, file inclusion, and compile-time variables. The most common syntax for the preprocessor is a line that begins with the bang character (`!`) and terminates with a newline. Although, depending on the way the preprocessor is used (such as an HTML filter), directives can be identified in different was depending on the input format.

Most strings in the preprocessor can use escape sequences similar to C strings.

Table 16.1. SCPG Escape Sequences

`\xXX`	This interprets the two characters following the `x` as a hexadecimal representation of the ASCII character.
`\{V}`	This extracts the environment variable named `V`. For example: "**`path \{HOME}/public_html`**"
`\p`	This is the current process identifier of the SCPG process. Commonly used with the `vmfile` directive to specify a temporary working file.
`\s`	A space.
`\n`	An ASCII newline.
`\r`	An ASCII carriage return.
`\t`	An ASCII tab.
`\q`	A single-quote character.
`\d`	A double-quote character.
`\\`	A literal `\` character.

The preprocessor allows sections of the input to be conditionally included or not. A simple conditional can be expressed like this:

```
!if env(INCLUDE_HW_DOCS)
path hwdocs
!endif
```

The above example would only apply the `path` statement if the environment variable `INCLUDE_HW_DOCS` were set to a non-zero or empty value. There are also much more complex things possible:

```
!if env(INCLUDE_HW_DOCS)
! if   env(HW_MODEL) eq 'X5530'
  path hwdocs/X5530
! elif env(HW_MODEL) eq 'X6001'
!  if env(INCLUDE_HOTPLUG) or env(LARGE_CHASSIS)
   path hwdocs/X6001/hotswap
!  else
   path hwdocs/X6001/nohotswap
!  endif
! endif
!else
path hwdoc_stub
!endif
```

The expression syntax is quite simple and includes the logical operators `and`, `or`, and `not`. Strings are single or double quoted and contain escape sequences (see Table 16.1, "SCPG Escape Sequences"). Strings can be compared with the `eq` and `ne` operators. Environment variables are queried with the `env` function. In addition, arithmetic can be performed using the traditional arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulus). The `osname` operator is the name of the operating system running the host tool.

Numerical comparisons are done using the `=` (equality), `!=` (non-equality), `>` (greater than), `<` (less than), `>=` (greater than or equal to), and `<=` (less than or equal to).

Expressions can test for the existence of a file in either the host file system (`hostexists`). In addition, tools can add their own functions. For example, the SCPG tool adds a function called `romexists`:

```
!if hostexists("/usr/share/special.html")
EXTRN hostman.html /usr/share/special.html
!endif

!if not romexists("secure/debug")
nodebug.html
!endif
```

In addition, Perl code loaded with the `!script` directive can be called. For example:

```
!script /usr/local/lib/capabilities.pl

!if perl:HasCapability('INCLUDE_HW_DOCS')
path hwdocs
!endif
```

The above example would call a Perl routine named HasCapability. The return value of that function determines the path of the conditional.

There is no reason to even limit the argument to literal values. For example:

```
!if perl:CheckFeature(env(FEATURE_ID) + 2, osname)
path somefeature
!endif
```

Will call the routine with the value of the environment variable `FEATURE_ID` plus two and the operating system name. Albeit, redundant (because those values are available directly in Perl), it is possible.

Environment variables can be set using the `!set` directive. There are two forms of this directive. The first form of `name = value` will set the environment variable called `name` to the value “value” (taking into account any escape characters). Double quotes are not recognized as special characters in the above form. For example, this:

```
!set my_var = "This is in environ[]"
```

Would actually set the variable `my_var` to the string containing the double quotes and spaces. Even though quoting is not relevant for setting environment variables, the escape sequences defined above work on the left hand side and can be used to put special characters into environment variables.

Alternatively, the `:=` operator can be used. This evaluates the right-hand side of the expression as if it were part of a `!if` directive. For example:

```
my_var := env(OTHER_VAR) + 2
```

Conversely an environment variable can be deleted using the `!unset` directive.

The `set` and `unset` commands manipulate the environment of the host tool. This environment persists for as long as the tool is running. For tools that process more than one file when run (such as SCPG) this may not be the most useful thing. Instead it would be better if a variable could be set only for the duration of processing a file. The `local` and `localunset` commands behave just like their global counterparts only the changes they make are undone after preprocessing is complete.

File paths can be manipulated somewhat portably with the `catfile` and `catdir` functions which concatenate the components of a file path and a directory path, respectively. The current directory can be obtained with the `cwd` function. The `updir` function returns the path component used to mean the previous directory.

For example, to construct a fully qualified path to a file in an environment variable:

```
!set CONFIG_FILE := catfile(cwd, updir, 'foo', 'bar.cfg')
```

The value of an expression can also be inserted “in line” using the `eval` directive. For example:

```
default_value {
```

```
!eval env(DEFAULT_VALUE) + 16
}
```

Files can also be included using the `!include` directive:

```
!include stdconfig.cfg

# Refer to the objects in stdconfig.cfg
```

Using the SCPG Tool

Introduction

SCPG runs on the development host and compacts content from the filesystem to form a single binary image that can be served from the embedded host's *ROM* filesystem. The term “*ROM* filesystem” is really a misnomer, because the content package can actually be stored anywhere as long as it can be accessed as a HttpdDataSource. Optionally, SCPG can even compress certain components of the content so that it takes less space. The data source is then interpreted by a HttpdRomFileSystem object.

SCPG is written in Perl but calls upon other tools for certain things (such as compression). In addition to the main tool, other tools such as bin2c are provided that can be useful for dealing with the content package after it's assembled.

Usage

When Seminole is built, SCPG is placed in `built/tools`. It can be run from that directory and can (usually) locate all of its required files from the path that it is executed in.

During the build process of some of the example applications a small content package is built from the content in the `html` subdirectory. The `content.cfg` in the examples directory is a good starting point for creating your own content packages.

The following command-line options are accepted by SCPG:

Table 16.2. SCPG Command Line Options

`-h`	Show help and usage information for command line arguments.
`-v`	Verbose. Give a summary of the content as it is processed.
`-o`	Set the output filename. If this option is not specified, it defaults to `content.pkg`.
`-c`	Set the configuration filename. If this option is not specified, if defaults to `content.cfg`.
`-x`	Treat any error as fatal. SCPG removes the output file and exits with an error code in the case of any warnings or errors.
`-T`	Preprocess and compile a template file. This option overrides the normal behavior and is used for per-file processing. See Standalone Templates.

`-t`	Compile a template file. This option overrides the normal behavior and is used for per-file processing. See Standalone Templates.
`-w`	If `-t` or `-T` is specified this option also causes whitespace to be minimized. See Standalone Templates.

Input Configuration File Format

The format of the configuration file is similar to most UNIX® command line shells. Tokens of one or more non-whitespace characters are separated by one or more whitespace characters. Both single and double quotes may be used to specify tokens containing whitespace. The beginning and ending quoting characters must match. If single quotes are used, then double quotes are ignored inside quoted strings. If double quotes are used, then single quotes can be used inside a double quoted string.

Comments are allowed. They are begun by the # character and extend to the end of that line. Comments can begin at any point in a line and terminate with the end of the line.

As with most other host tools, the input files are first processed by the host tool preprocessor. SCPG adds an additional expression function to determine if a file exists in the file system that is being generated for the target (`romexists`).

Lines in the configuration file that are neither blank nor comments are interpreted as directives by SCPG. Long lines may be continued to the next line with a trailing backslash (`\`) on the end of the previous line. This does not apply to conditionals (conditionals must be on a single line). This allows conditionals to work on a single, continued line. Line continuation does, however, apply to variable assignments.

The following directives are available:

Table 16.3. SCPG Configuration File Directives

`use`	Set or clears options and flags. One or more options may be listed on the command line following this directive. If the option is preceded by a "-" then the option is disabled, otherwise it is enabled. Available options are listed in Table 16.4, "SCPG Configuration File Options".
`mime`	Add an entry to the extension-to-*MIME* type mapping table. As SCPG processes the directory hierarchy it will try to guess the *MIME* type of any files that aren't explicitly given *MIME* types. The guess is done based upon the extension of the file. Each `mime` statement adds one or more extensions to a *MIME* type. For example, "**`mime text/html html htm hypertxt`**" would mean that files ending in `.html`, `.htm`, and `.hypertxt` are assumed to be have the *MIME* type `text/html`. More than one `mime` statement can be issued for each *MIME* type; each additional mapping is added to the list (the example statement could easily be written as three statements, one for each extension).
`mime_map`	Specify an Apache-style `mime.types` file. This is an alternative to specifying the *MIME* mappings manually using the `mime` directive described

	above. If you have a `mime.types` file in the standard Apache format, then this directive causes it to be sucked in and thus populate the *MIME* mapping table. For example, "**`mime_map /var/www/conf/mime.types`**" would read in the common *MIME* mappings from a standard Apache installation on an OpenBSD system.
`filter`	Specify a series of filters to apply to a specific *MIME* type. The first argument is the *MIME* type to trigger on. The second argument is the *MIME* type to actually encode the file as (this is what the HTTP client gets). Optionally, the output *MIME* type may be a `-` which means the input *MIME* type is the same as the output *MIME* type. The remaining arguments are all filter specifiers that are executed from left to right. See Filters.
`encoding`	Specify the recommended encoding for a specific *MIME* type. The format of the *ROM* filesystem allows for different files to be encoded using different means. For example, some files may use a compressed encoding or perhaps a tokenized form. As with *MIME* types, SCPG tries to guess the correct encoding based upon a file's *MIME* type (of course, this can be overridden on a file-by-file basis). Its syntax is otherwise similar to the `mime` directive.
`path`	Specify the directories containing the root files for the package. Using the example of an average Apache installation (crunching a document root into a *ROM* filesystem): "**`path /var/www/htdocs`**". More than one directory may be specified in this statement. In that case the content package contains the union of all the files in all the root paths.
`listing`	Specify the filename of the listing files. Each directory that is to contain actual files (not intermediate directories) must have an associated "listing file". The listing file explicitly determines which files in a directory go into the *ROM* image (to avoid working files being mistakenly added). Listing files also allow *MIME* and encoding parameters to be overridden on a file-by-file basis. By default, listing files are called `content.lst` (one per directory). However, some people may wish to name them something else, such as `.content.lst` so that they are hidden files. This directive permits such a change.
`define`	This statement is used to define new constructs (such as encodings) to SCPG. The first argument to `define` is the type of the object to define. Subsequent arguments are depend on the type of construct. Further discussion of encoding types (for

	when the second argument is “encoding”) is found in the next section. Alignment is covered in its own section as well.

Table 16.4. SCPG Configuration File Options

`subdirs`	Setting this option will include additional information in the *ROM* image so that directory listings can be constructed. By default, `subdirs` is disabled to save image space.
`have-attribute-decoder`	This option tells SCPG that the `INC_ROM_ATTRIBUTES` option is enabled. Both of these options should be enabled if you plan to use attributes for files (such as `charset`). There are also certain corner cases that require this option for extremely large *ROM* file systems. If this option is not enabled and it is required then SCPG will inform you with a fatal error.
`bad-filter-error`	Normally if a filter does not complete successfully construction of the content package is aborted and an error is returned. If this option is set and a filter fails, it is just skipped and processing continues normally.

Filters

SCPG allows preprocessing to be done on content before it is packaged. For each possible *MIME* type one or more filters may be run on the input. The output of each filter is passed as the input to the next in succession. After all filters have been applied to a file the *MIME* may optionally be changed and the file is passed to the appropriate encoder.

Filters are specified with the `filter` keyword and attached to a specific *MIME* type. The following are the built-in filters:

Table 16.5. SCPG Filter Types

`html-squish`	This filter should only be applied to HTML (or similar files). It removes redundant whitespace when possible to shorten the final content length. This reduces storage requirements and transmission time of the file. For a further reduction in storage the file can also be encoded with a compressor. This encoding can optionally take an argument of `keep-comments` to prevent removal of HTML comments. An argument of `avoid-tokens` can be used to handle the tokens of the `preproc` or `template` filters appropriately. Both options may be specified with a comma.
`css-squish`	This filter should only be applied to CSS files. It removes redundant whitespace when possible to shorten the final content length. This reduces

	storage requirements and transmission time of the file. For a further reduction in storage the file can also be encoded with a compressor. This encoding can optionally take an argument of `keep-first-comment` to prevent removal of the first comment in the file. This is useful to keep a copyright notice or other important comment at the start of the file but to remove programming comments.
`external`	This represents a filter operation that is performed by an external program. The supplied argument is the operating system command to run. If the argument to this filter contains the string `__input__` and `__output__` then those strings are substituted with the input and output file names of the filter, respectively. Otherwise the external filter is given its input on standard in and the output is read from standard out.
`perl`	This filter relies on the fact that SCPG is written in Perl. Using code loaded with the `!script` preprocessor directive, subroutines in that code can be called. If the subroutine returns true the filter operation is considered successful, otherwise failure is assumed.
`preproc`	This filter provides compile-time preprocessing for textual content, typically HTML. If the `html-squish` filter is used, be sure to enable the `avoid-tokens` option. For a complete reference on the syntax of the preprocessing directives see Content Preprocessing.
`template`	This filter compiles a template into binary form. Because the output of this filter is binary it should always be the last filter applied to content. If the `html-squish` filter is used, be sure to enable the `avoid-tokens` option. For a complete reference on the syntax of the template directives see Template Syntax.

The `external` filter can be quite useful for quick transformations using UNIX® tools such as **tr**. For example, to remove all of the `$` from a document, use the following filter rule:

```
        filter text/html - "external:tr -d '$'"
```

The `perl` filter is quite powerful because all of the constructs of Perl are available for processing content. The syntax of the `perl` filter argument is similar to that of normal Perl subroutine calls:

```
perl:mysub(1,2,foo)
```

The above filter argument would result in a call to an included subroutine called mysub with four arguments. The first argument is always passed in by SCPG and is a reference to a hash that contains the following members:

Table 16.6. SCPG Perl Filter Hashref Contents

input_file	This is the name of the input file, which is also opened for reading with a handle of INPUT.
output_file	This is the name of the output file, which is also opened for writing with a handle of OUTPUT.

The remaining parameters are 1, 2, and the literal string “foo”.

Encoding Types

As briefly described in the previous section, the `define` directive can be used to add new encoding types. This section describes the particulars of how SCPG implements encoding.

By default, there is one encoding type predefined by SCPG: `stored`. This is the most basic encoding method. The content is stored in the *ROM* image “as is”. This is the most efficient encoding in terms of speed but the least efficient in terms of space. In fact, on some operating systems that map the *ROM* package into the address space a direct send to the TCP/IP stack can be performed without any copying overhead.

However, other encodings can be established by use of the `define` directive. Seminole supports several compression schemes that cover a range of performance characteristics. A simple but effective compression scheme based upon the LZRW1/KH algorithm that has been floating around the net for some time. Alternatively the LZJB algorithm gets good compression while being very easy to decompress. Seminole also supports a more agressive (but slower) compression engine based on LZ with arithmetic coding. The compressors are bundled as helper applications compiled during the normal Seminole build. A description of how to add the compressors as encoding types will be illustrated below.

The initial part of an encoding type definition in the input configuration file is `define encoding`. After `encoding`, the next argument is the symbolic name by which the encoding is known to SCPG with the ID number used by Seminole to decode the data in parentheses.

Following the symbolic name and ID is the encoding access method. Currently, all encodings are accessed as external “helper” applications signified by the argument `helper`. However `stored` may also be used to indicate no transformation. Stored encodings should have an ID of 0.

The arguments following the access method are the command and arguments to execute. Certain tokens are replaced during execution of the helper:

Table 16.7. SCPG Encoder Symbols

`__is_ascii__`	Set to either the string “ascii” or “binary” depending on whether the input file's *MIME* type requires any format conversion.
`__source_file__`	This is expanded to the name of the input file that is to be used as the source of data.
`__output_file__`	This is the name of the file that the encoded data should be written to. This file should be opened in binary append mode as there may be existing data that can not be clobbered in the file.

Additionally, it is very important that the encoder report any changes to the content such that the decoded content is a different length than what is delivered (e.g. by altering the line endings of ASCII files). This is done by having the encoder emit the string `content-length: NNN` to standard output.

For example, to attach the supplied compression helper apps to encodings called `lzrw1kh`, `lzjb`, and `lzari`, you can add the following directives to your input configuration file:

```
define encoding lzrw1kh(1) helper lzrw1kh_compress 16384  \
                                          __source_file__ \
                                   __output_file__

define encoding lzari(2)   helper lzari_compress   16384  \
                                          __source_file__ \
                                          __output_file__

define encoding lzjb(3)   helper lzjb_compress   16384  \
                                        __source_file__ \
                                        __output_file__
```

The number in parenthesis after the encoding name is the codepoint that is used to reference the encoding in the runtime portion of Seminole. It is important that the numbers always are associated with the correct encoding. It is correct to define two lzari encodings with different block sizes as long as the code point is always 2.

The first parameter is the compressor (**lzrw1kh_compress** or **lzari_compress**), `16384`, is the block size that the tool tries to compress with. There is a fixed amount of overhead per compressed block, however; the larger this value, the more memory is required by Seminole during decompression of the file.

Alignment

For efficiency reasons, it is often desirable to align content on specific addresses. This can be especially true of certain kinds of flash memory. Alignments can be specified either globally or on a per-mime-type basis.

To define an alignment specific to HTML pages of 16 bytes, for example, add the following to the configuration file:

```
define alignment text/html   16
```

To define an alignment for all content that does not have a specific alignment of 8 bytes, the following may be added to the configuration file:

```
define alignment * 8
```

Note

An alignment of 0 means no alignment. This can be used to override a default alignment to not align various infrequently used *MIME* types.

Note

Currently, no attempt is made to optimize content by placing files with larger alignments first. It is expected that some common sense is used when assigning alignments.

Listing File Format

After setting up an appropriate input configuration file, the only remaining step is to create listing files in each of the subdirectories beneath the directory named by the `path` directive.

The format of a listing file is similar to the SCPG configuration file format. Each line starts with a directive and then a series of parameters. The directives define files that are to be included in the content package as well as optional parameters (such as *MIME* type or encoding). Comments are designated using # to the end of the input line. Preprocessing directives are also allowed throughout the listing file.

The simplest directive is `file`. This directive includes a single file. By default the file is included from the current directory of the host filesystem into the same (relative) directory of the target filesystem. For example, to include several HTML files:

```
file "order.html"
file "pizza.html"
file "sandwich.html"
file "frootloops.html"
```

Note

The same escape sequences defined in Table 16.1, “SCPG Escape Sequences” are allowed in the filenames placed in the listing file. If the string constants do not include any characters outside normal alphanumeric characters and a period (`.`) then the quotation marks may be omitted and the value is not subject to escape sequences. Adjacent string constants are concatenated just as in C.

Options that override the defaults may follow the filename. For example, let's assume that a directory contains a `tar.gz` file. In that case, you would like to override the default *MIME* type that SCPG normally guesses from the file name:

```
file "testdata.tar.gz"  mime "application/x-funky-tar"
```

In addition it is also possible to use a different encoding type (such as `lzrw1kh`, for example). In this case both overrides can be specified separated by a comma (`,`).

```
file "testdata.tar.gz" mime "application/x-funky-tar", \
     encode lzrw1kh
```

Note

Notice how the single line was continued to the next using a backslash as the last character on the line.

The `HttpdFileInfo` class also supports arbitrary name-value pairs called “attributes” to describe additional data about a file. These attributes can be set in the listing file using an assignment-like syntax. Adding to the example above two attributes are set on the file:

```
file "testdata.tar.gz" mime "application/x-funky-tar", \
     encode lzrw1kh, security = "restrict", password = "my file"
```

By default files are placed in the *ROM* file system in the directory of the listing file that describes them relative to the `path` directive in the configuration file. For example, if the `path` directive is `content/webapp` and the listing file is located in `content/webapp/settings/hardware` then the files would be placed in the *ROM* filesystem under `settings/hardware`.

This default location can be changed with the `location` directive. Continuing with the above example:

```
file "testdata.tar.gz" mime "application/x-funky-tar", \
     encode lzrw1kh, security = "restrict", password = "my file", \
     location "/downloads"
```

With very complex content descriptions two or more listing files may try to insert the same file. Ordinarily this results in an error from SCPG. However the `ignoredups` keyword prevents this from happening.

It can become tedious to place every file in the listing file. SCPG allows filename globbing with a different directive:

```
glob "*.html"
```

It is possible that a particular pattern may match nothing. This is especially possible when generic listing files are used with automatic content generators. For these situations the `optional` attribute ignores any file patterns that do not match any files.

When using the `glob` directive any attributes or parameters associated with the directive are applied to all the files.

Sometimes it is necessary to ensure a file has a different name in the *ROM* filesystem than on the host filesystem. This can be accomplished using the `extern` directive. This directive is identical to the `file` directive except that two file names must be specified. For example:

```
 extern "romfs.name" ("data.html")
```

The above example includes `data.html` from the host filesystem as `romfs.name` in the *ROM* filesystem. Of course, as with other directives modifiers and attributes can follow the left parenthesis.

Standalone Templates

For development or where the *ROM* filesystem is not used it may be desirable to use the template mechanism independently. With the correct command-line options SCPG can generate binary template files from one or more input files without going through the content packaging or compression steps.

To simply compile a series of template files the following command would suffice:

```
scpg -c std.cfg -t t1.thtm t2.thtm t3.thtm
```

Note

Notice that it is still valid to specify a configuration file so that global options can be set. Directives specific to packaging and compression are ignored when the `-t` (or `-T`) options are specified.

Sometimes compile-time pre-processing is also desired. For those cases the `-t` option can simply be substituted with a `-T`. In addition, with either `-t` or `-T` the `-w` option can be added to remove redundant whitespace as if the `html-squish` filter was applied.

Content Preprocessing

In many embedded systems multiple models of the same product require slight alterations to content. This can lead to the annoying situation of maintaining similar but slightly different versions of content for each product.

SCPG provides a filter, called `preproc` that provides a mechanism for preprocessing content similar to the C preprocessor. This is the same mechanism that is employed in the SCPG configuration and listing files.

As a quick example let us assume a PBX as our embedded device. Smaller models store all of their data in flash memory, while larger models offer a hard disk. The first issue is that commands relating to a hard disk are not present in some models, so we want to be able to select this at compile time. So we assume that an environment variable named `MODEL` contains the model number of the PBX that we are building the content for.

Using the filter directive in the `content.cfg` we tell SCPG to preprocess all `text/html` files:

```
filter   text/html  text/html    preproc
```

We then use a special sequence to denote preprocessor commands in the content:

```
<html>
 <body>
  <h2>Actions</h2>
  <ul>
   <li>Reboot system
   <li>Configure line card
   <li>Update dialing plan
  %[if (env(MODEL) eq 'P3000') or (env(MODEL) eq 'P3500')]%
   <li>Format hard disk
   <li>Check consistency of hard disk
   %[if env(MODEL) eq 'P3500']%
    <li>Copy disk to spare
   %[endif]%
  %[endif]%
  </ul>
 </body>
</html>
```

When SCPG runs the `preproc` filter on this file it evaluates the `%[` directives. If the model is `P3000` or `P3500` (the models with hard disks) the extra options are included. Furthermore, if the model is `P3500` (two hard disk slots) an additional option of backing up the hard disk is included.

The preprocessor can do much more than just conditionally select content. The same expression engine used for SCPG configuration file format is used for the content preprocessor.

Table 16.8. SCPG Content Preprocessing Commands

Command	Description
`eval`	Evaluate an expression and substitute the value for this directive.
`include`	Include (and additionally pre-process) another file.
`if`, `elseif`, `else`, and `endif`	Conditionally include sections of content.

Using the bin2c Tool

Introduction

The **bin2c** tool is a simple utility that can be used to take binary files and encode them as statically initialized C arrays. This is mainly useful for encoding content packaged with SCPG, which produces a single binary file as output.

This is often the most efficient (and easiest) way to get content included into an embedded system. The included data can then be encapsulated as a HttpdMemoryDataSource and passed to an instance of HttpdRomFileSystem.

Usage

When Seminole is built, the **bin2c** tool is placed in `built/tools`.

The following command-line options are accepted by **bin2c**:

Table 16.9. bin2c Command Line Options

`-o`	Set the output filename. This option is required. A usage message is generated if it is not set.
`-h`	If this option is specified, a header file is generated. The filename of the header file must follow this option.
`-p`	Use this option to generate C++ code. There is a subtle difference in the way constant data is declared between C++ and C.

Anything else is taken to be the file name of a file to turn into an initialized array and the corresponding symbol name to call that array (see below). More than one file may be generated in a single output file.

Typically, **bin2c** is used to encapsulate a `content.pkg` file:

```
bin2c -o content.c -h content.h website=content.pkg
```

The above would generate a header and source file that define an array called website. The header file would look similar to the following:

```
unsigned const char website[16384];
```

Because the size of the array is included even in the header file, the size of the file can be easily determined with a construct such as:

```
    HttpdMemoryDataSource website_data(website, sizeof(website));
```

Using the makecert Tool

Introduction

The **makecert** tool is a simple utility that helps to create SSL *certificates* and private keys. This tool requires that the **openssl** program be correctly installed.

The tool performs several steps including generating a private key, creating a certificate request, and self-signing the certificate. Once complete, the certificate and private key are in separate files as well as being available in a single PEM file that can be given to Seminole

The server certificate is also generated in DER format. This format is sometimes needed to install the certificate in a browser. In particular, this is the format expected by Microsoft Internet Explorer.

Usage

makecert takes no command-line parameters. It is interactive. To automate the generation of server keys and certificates the **openssl** tool should be used directly.

Once executed, **makecert** will ask several questions. The most important one is:

```
    What should the cert be called?
```

This is the base name of the generated files. If "**`foo`**" is entered, then the files generated will be:

`foo.key` (Server private key)
`foo.csr` (Certificate signing request)
`foo.cert` (Server identification certificate)
`foo.pem` (Server key and certificate)
`foo.der` (DER-encoded certificate for distribution)
`foo.opts` (Seminole options file)
`foo_dh1k.pem` (1024-bit DH parameters; only if Diffie-Hellman is enabled)
`foo_dh512.pem` (512-bit DH parameters; only if Diffie-Hellman is enabled)
Only `foo.opts` and `foo.pem` are required to start the server.

At some point during the generation of the certificates the script will ask for some geographical and identification parameters. The most important of these is the "common name." This field must be the DNS name or IP address that the server will be identified as to the browser. The browser verifies the information about the current page with the value of this field in the certificate.

A typical run of **makecert** would be:

```
Do you want to see the commands used for this run? [y/N] n
What should the cert be called? test1
Do you want a password protected key? [y/N] n
Generating a 1024 bit RSA private key
.++++++
..................................................++++++
writing new private key to 'privkey.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:US
State or Province Name (full name) []:Florida
Locality Name (eg, city) []:Boca Raton
Organization Name (eg, company) []:Acme General Widgets, Inc.
Organizational Unit Name (eg, section) []:Engineering Department
Common Name (eg, fully qualified host name) []:www.example.com
Email Address []:jrandom@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
writing RSA key
Signature ok
subject=/C=US/ST=Florida/L=Boca Raton/O=Acme General Widgets, Inc./OU=Engineering
Getting Private key
Generating default options file test1.opts


For more security, symmetric key encryption should not use the
server's private key. Instead, a key should be exchanged using
the Diffie-Hellman algorithm. Generating the parameters for this
algorithm may take some time but it will not adversely impact
server performance.

Do you want to use DH ephemeral keying? [y/N] n


The files test1.opts and test1.pem contain all that is
necessary for the server to operate in SSL mode.
```

Using the msgcmp Tool

Introduction

The **msgcmp** tool is used to compile a textual message catalog into a binary file that can be accessed using the string bundle class (`HttpdStringBundle`).

The input to this tool is a single file containing the logical names for the strings (how they are represented in the code) as well as one or more physical strings in different languages. The tool builds one or more binary files, each containing one particular set of physical strings. Optionally, a header file may be generated containing the identifiers for the logical strings defined as constants (with a prefix of `MSG_`).

The idea is to generate a different binary file for each locale that is to be supported but only a single header file. The same header file is always produced for the same input set of logical messages. Therefore, a generic code image can be compiled for all locales. Then a specific locale can be bound at a later time (typically by with the HttpdRomFileSystem). Alternatively, multiple binary files can be kept in a single device so the locale can be switched at runtime.

Usage

When Seminole is built, the **msgcmp** tool is placed in `built/tools`.

The following command-line options are accepted by **msgcmp**:

Table 16.10. msgcmp Command Line Options

`-r`	If this option is specified, the binary file is generated. The filename of the binary file must follow this option.
`-h`	If this option is specified, a header file is generated. The filename of the header file must follow this option.
`-l`	This option must be present if the `-r` option is specified. The requested locale name must follow this option.
`-d`	This option specifies that all of the locales in the input file should be build into appropriately named files and placed in the directory specified by this option. This is most commonly done to produce images containing all specified languages. If this option is specified then the `-l` and `-r` options are not allowed.

At least one of `-r` or `-h` must be specified. Alternatively, both can be specified to generate all of the required files in one pass.

Anything else is taken to be the file name of a file to process. At a minimum one file must always be specified. To keep things consistent the same set of filenames must always be specified in the same order between each invocation of **msgcmp**.

Input File Format

The input file is a text file that contains one or more “message definitions.” Each message definition contains a name and one or more physical strings associated with locale names.

```
[INVALID_CHARACTER_IN_NAME]
english: Invalid character in name
german: Unzulässiger Buchstabe im Namen
italian: Carattere non valido nel nome
```

A locale of `*` can be used to mean all other locales. So if for a particular message was the same for everything except English, a shortcut would be:

```
[FILE_SYSTEM_FAILURE]
english: Please contact us at 1-800-BAD-HARDWARE.
*: Please contact our overseas offices at 1-561-212-5555.
```

Using the specgen Tool

Introduction

The **specgen** is an extensible tool for generating complex code sequences from clear, concise specification files. In particular, **specgen** is well suited for generating some of the code for interfacing with the more complex API's of Seminole.

The format of specification files is similar to C or C++. In addition, host tool preprocessor directives are understood. The actual syntax of specification files is open ended. Initially, a few commands are defined by **specgen** internally. The most important of which, `package` loads additional capabilities into **specgen**. The `package` directive loads a Perl module from a file. That module can then add new directives to **specgen**.

By default, the **specgen** tool always creates a header file and a C++ (or C) source file. Typically the header file is included in other (hand-written) source modules to use the definitions declared by **specgen** in the source file.

The produced source file is then compiled and linked with the resulting application. The most common uses for **specgen** are for generating template symbol maps (`HttpdSymbolMap` parameters) or application framework objects.

Usage

When Seminole is built, the **specgen** command is placed in `built/tools`.

The following command-line options are accepted by **specgen**:

Table 16.11. specgen Command Line Options

`-c`	This mandatory option should be followed by the filename or the source file that will be generated.
`-h`	This mandatory option specifies the filename of the generated header file.

Anything else is taken to be the file name of a file to process. At a minimum one file must always be specified.

Input format

General conventions

As with most other host tools, comments are indicated with the pound character and terminate at the end of the line. Identifiers follow the rules of C++ identifiers. In particular, the scoping operator (`::`) can be part of an identifier. For example, `System::Heap` is a valid identifier but `System:1234` is not. Quoted strings and numeric constants also follow the rules of C and C++ as well.

Directives are identifiers with special meanings. Similar to C keywords they are almost always all lower case (although the directives are at the discretion of the package and not under the control of **specgen**). All directive bodies should be terminated with a semicolon just as C++ statements are terminated with a semicolon. Blocks are typically indicated using curly braces. Unlike C++, components of a block must also be terminated by a semicolon.

So a typical structured block in a specification file would look like:

```
object myObject
{
  anattribute 0x100;
  blockattribute
  {
    value      1;
    othervalue 2;
  };
};
```

Often times it is necessary for specification files to contain small snippets of C or C++ code. This is done using an arrow operator (`<-`). After this operator **specgen** will scan forward and absorb a single statement. The code fragment can contain nested blocks; **specgen** will copy a full statement, including any nested blocks.

For grouping multiple statements, the **<=** operator begins a block of native C or C++ code that is terminated by the `end` keyword.

```
pass source <=

  static unsigned int gCounter      = 0;
  static char         gFileName[16] = "default.file";
end;
```

Built-in directives

When initially processing an input file, **specgen** understands a few initial directives.

Table 16.12. specgen Default Directives

`package`	This directive loads a **specgen** “package” specified by the identifier name following the directive. There are several pre-built packages and additional packages can be built by a skilled Perl programmer.
`include`	This directive is used to specify a header file that should be included in the generated output. A quoted filename should follow the directive. There are also two modifiers that can precede the filename. The first, `standard` instructs **specgen**

	to use an include directive with angle brackets. Typically this tells compilers to find the include file using the specified include path. The second modifier, `header` causes the include declaration to be emitted to the generated header file as well as the generated source file. If both modifiers are present, the `standard` modifier must always precede the `header` modifier.
`pass`	The `pass` directive is used to simply pass code straight through to the output files. It must be followed by one of: `source` (send the following block to the generated source file only), `header` (send the following block to the generated header file only), or `all` (send the following code to both files). The option is then followed by a C or C++ code fragment preceded by the arrow operator.

Typical specification files will first use the `include` directive to pull in the appropriate header files (the Seminole *API* and whatever application-specific header files are necessary). The specific packages are then loaded using the `package` directive, followed by the actual specification bodies.

Included Packages

Seminole installs a few packages by default that can be used without any Perl programming. Each of these packages provides a few new directives that assist in programming a particular *API*.

The templates package

The `templates` package provides directives for programming some of the more tedious interfaces to the template engine.

The `template_constants` directive will build the necessary tables for using the `HttpdConstantSymbolTable` class:

```
template_constants PresentationParameters
{
  style  = "border: 2px; margin: 1em;";
  theme  = "/themes/slate.css";
  attrs  = "readonly maxlength=\"25\"";
};
```

That specification will result in the following declaration:

```
extern const HttpdPair PresentationParameters[3];
```

If the HttpdPair table were to be declared manually, it would have to be sorted by key (`HttpdConstantSymbolTable` uses a binary search). However, when using the `template_constants` directive, the tool automatically sorts the entries for you. Removing these kinds of error prone, tedious tasks is the primary reasoning behind **specgen**).

The `symmap` directive is used for generating HttpdSymbolEntry tables to support the `HttpdSymbolMap` and `HttpdScopedSymbolMap` classes. Briefly, a symbol table map is an array of named fields that specify an offset in a structure and one or more handler procedures. These maps make displaying data from C (or C++) data structures easy.

As with `HttpdConstantSymbolTable` tables, the sorting is done automatically and fields can appear in any order. A structure name must be associated with the map name, as an example we will assume a structure named `Person` is defined in an application-specific header file as follows:

```
struct Person
{
  char           first_name[64];
  char           last_name[64];
  unsigned long  age;
  bool           married;
  const char    *occupation;
  char           sex;
};
```

Given the structure above, we can use the `symmap` directive to map this into template directives. Most of the fields can be handled using the standard handlers provided by `HttpdSymbolMap`, except `sex`. Of course, we can write some simple code to handle the character field and do anything we want. In fact, we can make that particular field more complicated. It can be `M` for male, `F` for female, or zero if the sex is not known.

We can make the identifier `sex` both a template conditional (not zero) and a template evaluation (the appropriate label).

```
symmap PersonMap: Person
{
  first_name       = stringbuf;
  last_name        = stringbuf;
  age              = ulong;
  married          = bool;
  job (occupation) = string;
  sex
  {
    cond <-
    {
      const char *p_char = (const char *)p_data;
      return (HttpdSymbolTable::ReturnBool(*p_char != 0));
    };

    eval <-
    {
      const char *p_char  = (const char *)p_data;
      const char *p_label = (*p_char == 'M') ? 'Male' : 'Female';
      return (p_eval->Output()->WriteString(p_label));
    };
    };
  };
```

For the types that `symmap` knows about we can use the simplified sequence as is done for the first four fields. The field `occupation` is mapped to `job` in the template, but it is still a predefined type.

For the `sex` field, we provided a conditional code fragment and an evaluation code fragment (we also could have provided code for a `loop` fragment). Therefore, the template symbol `sex` can be used in conditionals (to determine if it is present) and can be evaluated to produce the actual value.

Alternative names can also be provided for specifically defined types. In addition, in place of the code blocks, an identifier can be provided. Care must be taken to ensure that the prototype of the provided identifier is included and matches what is needed. With those two additional changes in mind, the last field could be specified like this:

```
gender (sex)
{
  cond <-
  {
    const char *p_char = (const char *)p_data;
    return (HttpdSymbolTable::ReturnBool(*p_char != 0));
  };

  eval OtherClass::FormattingMethod;
};
```

Table 16.13. symmap predefined types

Type	Function
`string`	`HttpdSymbolMap::EvalString`
`stringbuf`	`HttpdSymbolMap::EvalStringBuffer`
`ulong`	`HttpdSymbolMap::EvalUlong`
`long`	`HttpdSymbolMap::EvalLong`
`hexlong`	`HttpdSymbolMap::EvalHexUlong`
`bool`	`HttpdSymbolMap::CondBool`

Appendix A. Obtaining Support

All Seminole licenses include 8 hours of support. Additional support can be purchased at a cost of $90US per hour. Please contact a sales representative for more information.

(+1) 1-561-213-6177
E-mail `<sales@gladesoft.com>`
http://www.gladesoft.com/

Glossary

A

Alignment	Locating data such that it is at an address that is appropriate for its type. For example, many CPU architectures can only access words on their natural boundary. Thus, a 16-bit value can not be accessed at an odd address (on a byte-addressable machine). Unaligned data may not always result in failure but may often result in performance degradation.
ANSI	The American National Standards Institute. A standards body responsible for various standards incuding those in computers and engineering. Typically the acronym ANSI is used to refer to the C programming language standard.
Application Programming Interface (API)	The interface that to developers who are utilizing Seminole to build web interfaces and applications see. The term “Application Programming Interface” is used in this manual to refer to all the documented public interfaces of Seminole.
Application	Any code that is not part of the Seminole library. Typically this term is used to refer to user-written code that implements a web-based interface.
ASCII	American Standard Code for Information Interchange. A 7-bit character encoding that assigns the letters of the Roman alphabet, the decimal numbers and various special symbols and control sequences to numeric codepoints.

B

Base-64	An encoding scheme used to make 8-bit (binary) data safe for transfer over protocols and interfaces that can only send ASCII text. This is a common encoding for large binary data when transmitted using older Internet protocols that do not tolerate binary data well.
Blocking	When an operation (such as reading from a socket) halts the current thread until the operation can be completed. An operating system may perform other tasks while the thread performing the blocking operation is suspended. If an operation is said to be non-blocking then it will return immediately (often with failure) if the operation can not be completed at the current moment. See Also Thread, Real-Time Operating System.

C

Certificate	A cryptographically signed blob of data that is used for identification. SSL-enabled webservers should present a certificate that allows the client to prove the validity of the server. Typically server-side certificates are signed by a hierarchy of third-party registrars where some type of physical proof was presented. It is also possible for servers to verify clients with certificates when using SSL.
CGI	Formally a standard for external software to interface with a web server. Informally this term is used to refer to any kind of dynamic web page generated with parameters sent along with the request. In Seminole there is no concept of a process or separate address space so the formal meaning does not apply.

Cookie	A small chunk of data that is stored within the HTTP client and sent back to the HTTP server on subsequent requests. A cookie is often used like an ID card or the key to a building. It allows the stateless HTTP protocol to associate a particular client with incoming requests.

D

DOM	A tree data structure representing a structured document. This data structure is typically created from an XML representation.

E

Endian	The organization of multi-byte words in computer memory. There are two very common byte orderings used by modern CPU's today. In big endian byte ordering the most significant byte (the big end) comes first (at the lowest address of the word). Little endian byte ordering is the opposite of big endian, the least significant byte comes first.
Entropy	Randomness, usually in the form of random byte values. Typically when referred to as “Entropy” it is being used in a cryptographic context where high quality randomness is essential.

H

Hash Function	A function that reduces a large amount of data (call the input) to a smaller sample of data (called the hash result). Typically the length of the hash result is fixed. The larger the input is compared to the length of the hash result the higher the chances of a “collision” are. A collision is when two different inputs produce the same hash result. The hash result, although not unique, can in many cases be used as a shorthand for the input. There are two principle uses of hash functions: hash tables and cryptographic purposes. In the cryptographic case a hash can be used to detect the tampering of data (such as a digital certificate). Hash table use the hash result as a hint to make searches much more efficient.
Hash Table	A data structure used for quick lookups of exactly-matching keys. A typical style of hash tables, open-chained, consist *N* linked-lists (called buckets) and a hash function that produces a result from 0 to *N*-1. A key value can then be placed through a hash function and used to identify which list the associated record can be found in. The larger the value of *N* the less nodes per bucket therefore the less time spent searching for the correct record.
Host	The machine where development with Seminole takes place. In embedded systems this is often not the same machine where the resulting software is executing. Seminole is designed with the idea that the host system has much greater performance and resources than the target system. This is typical of embedded development environments. See Also Target.

I

Idempotent	The property of an action where the same results are obtained reguardless of the number of times the operation is performed.

M

Multimedia Internet Mail Extensions (MIME)	A standard encoding mechanism for E-mail extensions. Portions of this standard have been employed in the HTTP protocols. In particular requests and responses include name-value pairs encoded the using MIME header format.
Multicast	A multicast packet is an IP packet that is directed to a group of hosts rather than a single host. Multicast packet delivery takes advantage of the properties of broadcast networks (such as Ethernet) to efficiently transmit data in a one-to-many fasion.

N

Nagle Algorithm	An algorithm that delays the sending of a packet in a TCP socket in the face of single-byte writes to reduce the number of packets that are transmitted. The Nagle algorithm is often a benefit for interactive data transfer but a detrement for bulk transfers. Seminole attempts to transmit data intelligently when possible and does not require the Nagle algorithm.

P

Perl	Practical Extraction and Reporting Language. Perl is a powerful scripting language with many text processing features. Well written Perl scripts are independant of the host operating system and can be run on any host platform without modification. Most of the host tools are written in Perl for portability reasons. See Also Host.
Porting	The process of adjusting Seminole so that it can run in a new environment (e.g., different CPU, operating system, or compiler). Often time this is simply accomplished by modifying the porting layer or re-implementing it for the new target. This process is described in detail in the section called “Operating Environment Abstraction Layers”.

R

RFC	Request For Comment. A forum of peer-reviewed documents that are used to define and develop Internet protocols.
ROM	A form of non-volatile storage that maintains its data even in the even of a power loss. In this document ROM is used to refer to the type of storage used to hold compiled code and constant data. In most cases this is typically flash or disk.
Real-Time Operating System	An operating system, typically designed for embedded systems, that provides certain guarantees about the scheduling of tasks. Although Seminole does not require real-time behavior it is often necessary for the kinds of environments

	Seminole is used in. This term is often used to describe the operating system that is supervising the execution of Seminole even if that operating system does not provide real-time guarantees. See Also Thread, Blocking.

S

Seminole	A tribe of Native American Indians that have since settled in South Florida, where Seminole was written. The name Seminole means “run away.” Aside from the similarity to the Apache webserver we hope that Seminole can run-away web-interface problems.
Socket	An endpoint of the TCP protocol that is either used to accept new incoming connections (a “listening socket”) or to transport data to another socket elsewhere in a TCP/IP network.
Static Class	A class which is only used to provide a namespace. Instances of a static class should never be declared.

T

Target	The CPU that is executing Seminole. In embedded systems this is often not the same machine where development takes place. See Also Host.
Thread	One instance of code in execution. In many operating systems multiple threads of execution exist and execute simultaneously. When one thread must wait for an event an operating system can make more efficient use of the CPU by running other threads during the wait. See Also Real-Time Operating System, Blocking.
Transport	The layer(s) of a protocol stack used to perform the reliable stream-oriented data exchange that is used by the HTTP protocol. Normally this is either TCP or SSL but there is nothing inherent in the HTTP protocol that prevents the use of other transports if appropriate.
WebDAV	WebDAV is an extension to HTTP to allow for distributed authoring and versioning. New methods are defined to upload resources (files), create collections (directories), delete resources, as well as iterate collection listings in a machine readable way.

Colophon

This book was produced using the XML DocBook [http://www.docbook.org/] schema and the xsltproc processor to create an XML FO (Formatting Object) file. The FO file was rendered to PDF using Apache™ FOP. Illustrations were created using Inkscape and saved in SVG format.