Embedded Web-Server Development

A crash course in writing embedded servers using the Snorkel API

Version .09

Walter E. Capers

01.19.2010

Table of Contents

About Snorkel

Snorkel provides basic web server capabilities for natively built applications. The library supports both dynamic and static content, has a very low memory profile, and provides features that simplify the creation of web based application interfaces and/or proprietary protocols. Developed in C, adhering to POSIX and Windows standards, Snorkel is a highly portable runtime library. It is also one of the fastest embedded web-servers available, capitalizing on multi-core technology.

There are many advantages to using embedded web servers:

- HTTP is well-studied cross-platform protocol.
- HTTP clients, web-browsers, are readily available on all modern computers.
- Intranet routers seldom block HTTP.
- A growing trend in using embedded web servers within applications mirrors ubiquitous computing.
- Natively compiled embedded web servers require minimal system resources and are self-standing requiring little or no software perquisites such as a JVM, Apache, Eclipse, or complex frameworks, which may not be readily available or consistently supported on target platforms.

While there are several embedded web servers available on line, many have restrictive licenses, limited platform support, or require large learning curves.

Keeping It Simple

Snorkel is not an end all be all web server solution. Its capabilities are limited in comparison to mainstream web servers like Apache and Tomcat. Then again, Snorkel is an embedded server and does not have to be. Embedded servers only need to provide enough functionality to expose or provide access to the underlying business logic of the linked application. Further, by eliminating unnecessary server side functionality, Snorkel's contribution to an application's resource consumption is minimal making it one of the leaner solutions.

SNORKEL SUPPORTS THE FOLLOWING

- HTTP 1.0
- Open SSL[1]
- IPV4 and IPV6
- URI overloading

[1] Open-SSL support requires the Open-SSL runtimes and Snorkel-SSL runtime library, (snorkel32ssl or snorkel64ssl). The Snorkel-SSL runtimes are not currently distributed outside of the US due to US trade laws.

    - URIs can be mapped to user defined functions
- mime overloading
    - mime types can be mapped to user defined functions
- mime creation
    - users can define their own mime types and map the mime types to user defined handler functions
- asynchronous message processing and unlimited scalability
    - requests are processed asynchronously by multiple threads allowing hundreds of networked users to access the same application
    - Snorkel threads and memory allocation schemas leverage NUMA topology for improved performance
- Simple logging
- IP filtering
- All client side technologies
- Supports all common browsers
- Plug-in framework

Supported Platforms
AIX
SunOS
HP-UX
LINUX
OSF

Figure 1, example of a page served by Snorkel

Getting Started

This guide is primarily a how to guide for using Snorkel. It does not go into detail about Snorkel's inner workings or architectural design except in cases that require additional insight. Instead, this guide provides a hands-on approach to using Snorkel. All of the examples presented are in C and HTML. The sources presented were fully tested. To avoid explaining the basics of writing applications in C/C++ and web site development/design, we assume that you have some knowledge of programming in C/C++ and web development.

Naming Conventions

The Snorkel API follows a simple naming convention, diagrammed below.

API Naming Convention
snorkel_*noun*
snorkel_*noun*_*verb*

Where "noun" defines the data type/entity that a function operates upon and "verb" defines the action. For example, the function *snorkel_obj_set* sets a Snorkel object's attribute. Snorkel variable types follow a similar naming convention.

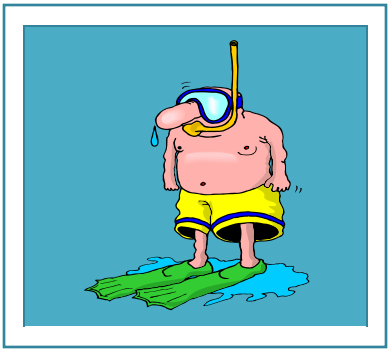
snorkel_*noun*_**t**

Most API calls return an integer value of SNORKEL_SUCCESS on success and SNORKEL_ERROR when an error occurs. In cases where an API call fails, Snorkel also sets the *errno* value to reflect the cause.

Asynchronous Message Processing

Before we get started, it is important to note that Snorkel embedded-servers' run within an application under their own threads – processing events asynchronously. When providing access to business logic in your application, it is important to use Snorkel's synchronization objects or system level synchronization objects to prevent gridlock and/or race conditions. Before you start coding, be sure to read the Mutual Exclusion section of this manual.

1 – Diving in, Your First Embedded Server



In this section, we introduce the Snorkel API. You will learn how to embed a Snorkel server in a simple application.

1.1 Embedding Snorkel

Before you embed Snorkel in your application, you need to determine how your application will interface with Snorkel. Here are a couple of examples:

- Your application provides a service and Snorkel provides an HTTP based UI to manage the service on a network.
- Your application runs locally and Snorkel provides an HTTP based UI as the primary end-user interface.

We begin with a simple example – a program that launches an embedded server that streams the phrase “Hello from Snorkel” when connected to a web-browser.

Example 1-1

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <snorkel.h>
4
5   void
6   syntax (char *pszProg)
7   {
8     fprintf (stderr, "syntax error:\n");
9     fprintf (stderr,
10            "%s [-i <index_file_directory>] [-p <port>]\n",
11            pszProg);
12     exit (1);
13   }
14
15
16   void
17   main (int argc, char *argv[])
18   {
19     int i = 1;
```

```
20  int port = 80;
21  char *pszIndex = 0;
22  char szExit[10];
23  snorkel_obj_t http = 0;
24
25  for (; i < argc; i++)
26    {
27      if (argv[i][0] == '-' || argv[i][0] == '/')
28        {
29          char carg = argv[i][1];
30
31          switch (carg)
32            {
33            case 'p':          /* port number */
34              port = atoi (argv[i + 1]);
35              i++;
36              break;
37            case 'i':          /* index directory */
38              pszIndex = argv[i + 1];
39              i++;
40              break;
41            default:
42              syntax (argv[0]);
43              break;
44            }
45        }
46
47    }
48  if (!pszIndex)
49    syntax (argv[0]);
50
51  /*
52   *
53   * initialize API
54   *
55   */
56  if (snorkel_init () != SNORKEL_SUCCESS)
57    {
58      perror ("could not initialize snorkel\n");
59      exit (1);
60    }
61
62  /*
63   *
64   * create a server object
65   *
66   */
67  http = snorkel_obj_create (snorkel_obj_server, 2,     /* number of handler threads to create */
68                             pszIndex   /* directory containing index.html     */
69    );
70
71  if (!http)
72    {
73      perror ("could not create http server\n");
74      exit (1);
75    }
```

```
76
77    /*
78     *
79     * create a listener
80     *
81     */
82    if (snorkel_obj_set (http,    /* server object */
83                         snorkel_attrib_listener, /* attribute   */
84                         port,    /* port number */
85                         0 /* SSL support */ )
86          != SNORKEL_SUCCESS)
87        {
88          fprintf (stderr, "could not create listener\n");
89          snorkel_obj_destroy (http);
90          exit (1);
91        }
92
93    if (snorkel_obj_set
94          (http, snorkel_attrib_ipvers, IPVERS_IPV4,
95           SOCK_SET) != SNORKEL_SUCCESS )
96        {
97          fprintf(stderr,"error could not set ip version\n");
98          exit (1);
99        }
100
101   /*
102    *
103    * start the server
104    *
105    */
106   fprintf (stderr, "\n\n[HTTP] starting embedded server\n");
107   if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
108     {
109       perror ("could not start server\n");
110       snorkel_obj_destroy (http);
111       exit (1);
112     }
113
114   /*
115    *
116    * do something while server runs
117    * as a separate thread
118    *
119    */
120   fprintf (stderr, "\n[HTTP] started.\n\n"
121            "--hit enter to terminate--\n");
122   fgets (szExit, sizeof (szExit), stdin);
123
124   fprintf (stderr, "[HTTP] bye\n");
125
126   /*
127    *
128    * graceful clean up
129    *
130    */
131   snorkel_obj_destroy (http);
```

```
132     exit (0);
133   }
```

```
<!--HTML source c:\snorkel\index.html -->
<html>
        <body>
        <h2>Hello from Snorkel</h2>
        </body>
</html>
```

Let us take a closer look at our first embedded server program. Prior to calling any Snorkel API, we initialize the API by calling **snorkel_init**. As with most Snorkel functions, the function **snorkel_init** reports errors by returning a value of *SNORKEL_SUCCESS* on success or *SNORKEL_ERROR* on error. The *perror* function and the *errno* value provide additional information for failed calls. On Windows, the *GetLastError* function can also retrieve errors reported by the API. For portability, we recommend sticking with the *errno* family of functions.

```
#include <snorkel.h>

int  snorkel_init()
```

Figure 2, the function snorkel_init initializes the Snorkel runtime and must be called prior to calling any API function

Following initialization, we create a Snorkel server object using the **snorkel_obj_create** function.

http = snorkel_obj_create (snorkel_obj_server,2,pszIndex)

This function provides object creation for all Snorkel objects. The term object can be misleading; objects in Snorkel are not derived from classes as in C++ or JAVA and do not contain member functions to operate on data. Instead, they are encapsulations of allocated data obfuscated by pointer casting. Think of a Snorkel object as a black box containing private data and a description of the data for the Snorkel runtime.

```
#include <snorkel.h>

snorkel_obj_t  snorkel_obj_create(snorkel_obj_type_t  object_type,  default_attribute0,
default_attribute1…)
```

Figure 3, syntax of snorkel_obj_create

The following table illustrates the various object types supported by the **snorkel_obj_create** function.

Snorkel object types	Description	Arguments
snorkel_obj_server	Creates an embedded server object	*(int) number_of_ threads, (char *)index_directory*
snorkel_obj_mutex	Creates a mutual exclusion object	
snorkel_obj_event	Creates an event synchronization object	
snorkel_obj_stream	Creates a connection with a remote host using a specified non-HTTP protocol	*(char *)hostname, (char *)protocol, (int)port, (int)connection_timeout*
snorkel_obj_log	Enables logging and debugging	(char *)filename

Table 1, snorkel object types

To create a server object, the function **snorkel_obj_create** takes two parameters: the number of handler-threads to create, and the location of the HTML index file “index.html”. Handler-threads are threads that process incoming HTTP requests. Figure 5 illustrates how handler threads process requests.

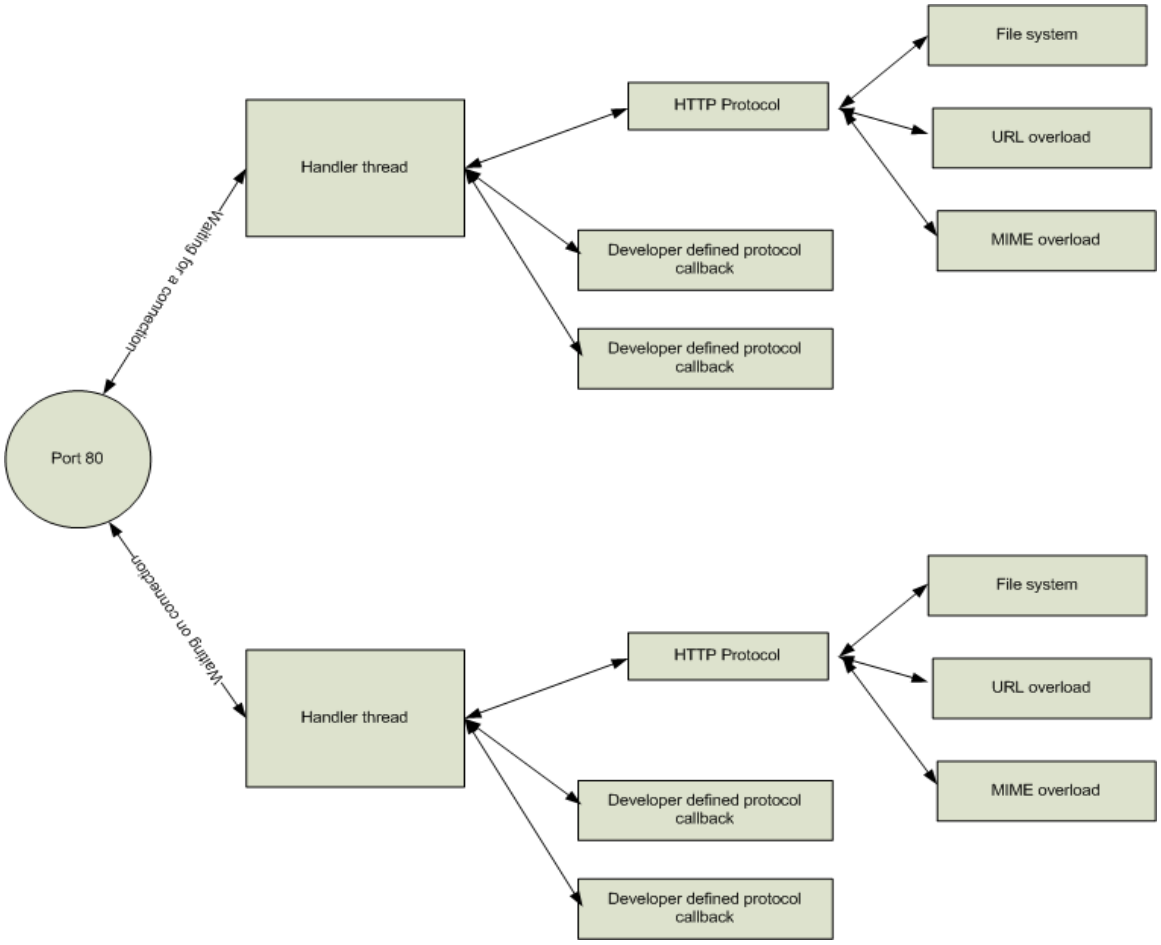


Figure 4, message-processing overview

In this example, we defined a server that has two handler-threads. As requests come into an embedded-server, the operating system delegates each request to the first available handler-thread for processing. Handler-threads are protocol agnostic and can process HTTP as well as developer-defined protocols. The server in this example can handle two user requests concurrently if the deployment system has at least two CPUs. A thread-governor within the runtime limits the number of threads to either the value provided (a suggestion) or the number of CPUs on a system – the lesser value of the two. For example, specifying a thread count of five on a three CPU system results in a thread count of three. In general, when determining a thread count it is better to use a high value, allowing the governor to correct it, or to allow an end-user to specify the value through a configuration method.

Before our server can take requests, it needs a port number. The default port number for this example is 80. An end user can override the default using the "-p" option at the command line. We use the **snorkel_obj_set** function with the attribute-type **snorkel_attrib_listener** to define the port number.

if (snorkel_obj_set (http, snorkel_attrib_listener ,port ,0) == SNORKEL_ERROR)

For the third parameter, the SSL flag we provide a zero (we will talk more about SSL later).

```
#include <snorkel.h>

int snorkel_obj_set (snorkel_obj_t server_obj , snorkel_attrib_t snorkel_attrib_listener, int port, int
want_ssl)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 5, defining a port listener

Even though we only used a single port in this example, Snorkel servers' can listen on multiple ports. Each additional port requires a separate call to **snorkel_obj_set**.

After defining a listener, we set the IP version for all incoming connections using the **snorkel_obj_set** function. The default setting for new connections supports both IPv4[2] and IPv6[3]. Unfortunatly, there are still some platforms that do not fully support IPv6. To insure portability we disable IPv6, by specifying IPv4 support only. As with the port number, the ability to configure IPv4 and or IPv6 protocols should be an end-user option.

All socket attributes such as IP version are listener attributes which operate on existing listeners. Setting a socket attribute prior to creating a listener will produce an error. In addition, calls to set socket attributes only apply to existing listeners; listeners created after a set operation do not inherit the setting.

Finally, we call the function **snorkel_obj_start**, which starts the server in a separate thread called the kernel-thread. The kernel-thread launches the handler-threads and returns procedural control to **main**.

```
if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
```

We call the function **fgets**, which waits for input, to prevent the program from exiting. When the end-user hits the enter key, to exit, we call the function **snorkel_obj_destroy** to gracefully shutdown the embedded server.

```
#include <snorkel.h>

int  snorkel_obj_start (snorkel_obj_t object)
```

[2] IPv4 (Internet Protocol version 4), referse to the fourth addition and first version of the Internet Protocol (IP) to be widely deployed.

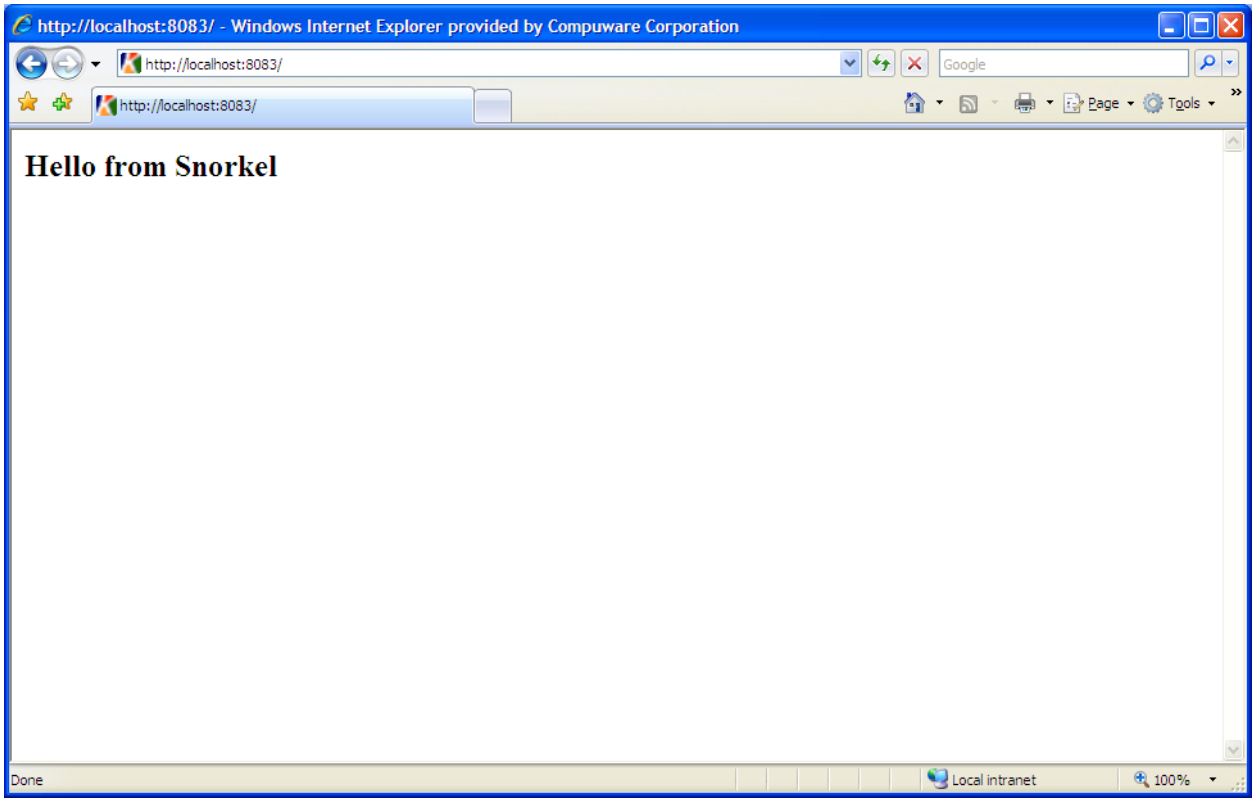
[3] IPv6 (Internet Protocol version 6), referse to the sixth addition of the Internet Protocol (IP) – the successor to IPv4.

```
#include <snorkel.h>

int  snorkel_obj_destory  (snorkel_obj_t object)
```

Figure 6, syntax for snorkel_obj_start and snorkel_obj_destroy

To compile the example on LINUX, link with the following libraries: pthread, nsl, m, rt and libsnorkel32.so. On Windows, specify /MD (multi-thread DLL option) for the C-runtime and link with snorkel32.lib. After launching the executable, enter the URL http://localhost from your browser to interact with the embedded server.

Example 1-1 Output



Play around with the example using different index files to get a feel for the capabilities and limitations of Snorkel embedded web servers. The code provided in the first example illustrates the basic framework/boiler-plate code for embedding Snorkel in any application.

Calls	**Description**
if(snorkel_init() != SNORKEL_SUCCESS) //handle error condition	Initialize API – snorkel_init

server = snorkel_obj_create(snorkel_obj_server,
NUM_THREADS,*INDEX_DIR*);
if(!**server**)
//handle error condition
	Create an embedded server – snorkel_obj_create

if(snorkel_obj_set(**server** ,snorkel_attrib_listener, $port_0$, *SSL_flag*) != SNORKEL_SUCCESS)	Set listener port numbers

//handle error condition
 	
if(snorkel_obj_set(**server** , snorkel_attrib_listener, *port*$_1$, *SSL_flag*) != SNORKEL_SUCCESS)
//handle error condition
•
•
 	
if(snorkel_obj_set(**server** , snorkel_attrib_listener, *port*$_n$, *SSL_flag*) != SNORKEL_SUCCESS)
//handle error condition	

if(snorkel_obj_set(**server**,*attrib*$_0$,…) != SNORKEL_SUCCESS)
// handle error condition
 	
if(snorkel_obj_set(**server**,*attrib*$_1$,…) != SNORKEL_SUCCESS)
// handle error condition
•
•
•
if(snorkel_obj_set(**server**,*attrib*$_n$,…) != SNORKEL_SUCCESS)
// handle error condition
	Set other server attributes – snorkel_obj_set

obj$_0$ = snorkel_obj_create(*object_type*$_0$,…);
 	
if(snorkel_obj_set(**obj$_0$**,*attrib*$_0$,…) != SNORKEL_SUCCESS)
// handle error condition
if(snorkel_obj_set(**obj$_0$**,*attrib*$_0$,…) != SNORKEL_SUCCESS)
// handle error condition
•
•
if(snorkel_obj_set(**obj$_0$**,*attrib*$_n$,…) != SNORKEL_SUCCESS)
// handle error condition
 	
obj$_1$ = snorkel_obj_create(*object_type*$_1$,…);
 	
•
•
	Create other Snorkel Objects and set attributes – snorkel_obj_create, snorkel_obj_set

if(snorkel_obj_start(**server)** != SNORKEL_SUCCESS) // handle error condition	Start embedded server – snorkel_obj_start

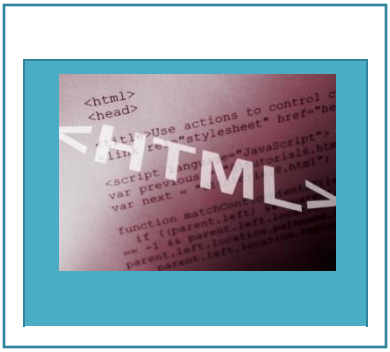
while(running)
{
•
	Do something

• }	

snorkel_obj_destroy(**obj$_0$**);
snorkel_obj_destroy(**obj$_1$**);
•
•
snorkel_obj_destroy(**obj$_n$**);
snorkel_obj_destroy(**server**);
	Destroy objects – snorkel_obj_destroy

Table 2, boilerplate code

2 – Dynamic Content, Overloading URIs



In this section, we are going to look at generating dynamic content. Dynamic content refers to web content generated by an application in real time. By default, the Snorkel –runtime looks to the file system to resolve URI requests. The procedure of providing a function that generates dynamic content and mapping the function to a URI, defines URI overloading. When a URI is overloaded, the runtime calls the function associated with the URI instead of attempting to resolve the URI using the file system.

In addition to overloading URIs with functions, you can also overload URIs by providing pointers to content stored within an applications memory space or a file to be loaded into an applications memory space at runtime. A "NULL" terminated string containing HTML or XML data is an example of memory based content.

Whether you are using memory based content or content generated by a function, Snorkel supports URI overloading for both the HTTP POST and GET methods.

Before we go further in our discussion on URI overloading, let us take a moment to clear up a common point of confusion regarding the definitions of a URI and URL. A URI (Universal Resource Identifier) defines a resource by name but does not identify the actual location of the resource. On the other hand, a URL (Universal Resource Locater) refers to the unique location of a specific resource. For example, let us say you wanted to contact an old friend that went by the name of John Doe. There are many John Doe's in the world; the name alone does not identify where to contact him or which John Doe we are referring. Think of the name John Doe as a URI. To contact the John Doe, the one we are referring to, we would need the URL form of his name, which would include his address and phone number. Think of URIs as a subset of URLs.

2.1 Overloading HTTP-GET

The HTTP request method GET, requests a representation of the source identified by a URI. When we overload the GET method, we associate the provided URI with a dynamically generated source. In the next example we create a routine to overload the URI "/index.html". The server, when accessed through a browser, displays the phrase "Dynamic Content".

Example 2-1

```c
#include <stdio.h>
#include <stdlib.h>
#include <snorkel.h>

callback_status_t
index_html (snorkel_obj_t http, /* read environment from this object */
            snorkel_obj_t outstream     /* write data to the output stream   */
  )
{
  if (snorkel_printf (outstream,
                      "<html><body><h2>Dynamic Content"
                      "</h2></body></html>\r\n") ==
      SNORKEL_ERROR)
    return HTTP_ERROR;

  return HTTP_SUCCESS;
}

void
syntax (char *pszProg)
{
  fprintf (stderr, "syntax error:\n");
  fprintf (stderr, "%s [-p <port>]\n", pszProg);
  exit (1);
}


void
main (int argc, char *argv[])
{
  int i = 1;
  int port = 80;
  snorkel_obj_t http = 0;
  char szExit[10];

  for (; i < argc; i++)
    {
      if (argv[i][0] == '-' || argv[i][0] == '/')
        {
          char carg = argv[i][1];

          switch (carg)
            {
            case 'p':          /* port number */
              port = atoi (argv[i + 1]);
              i++;
              break;
            default:
              syntax (argv[0]);
              break;
            }
        }

    }

  /*
```

```
57    *
58    * initialize API
59    *
60    */
61   if (snorkel_init () != SNORKEL_SUCCESS)
62     {
63       perror ("could not initialize snorkel\n");
64       exit (1);
65     }
66
67   /*
68    *
69    * create a server object
70    *
71    */
72   http = snorkel_obj_create (snorkel_obj_server, 2,     /* number of handler threads to create */
73                              NULL       /* directory containing index.html     */
74     );
75
76   if (!http)
77     {
78       perror ("could not create http server\n");
79       exit (1);
80     }
81
82   /*
83    *
84    * create a listener
85    *
86    */
87   if (snorkel_obj_set (http,    /* server object */
88                        snorkel_attrib_listener, /* attribute   */
89                        port,    /* port number */
90                        0 /* SSL support */ )
91       != SNORKEL_SUCCESS)
92     {
93       fprintf (stderr, "could not create listener\n");
94       snorkel_obj_destroy (http);
95       exit (1);
96     }
97
98   /*
99    *
100   * overload the URL index.html
101   *
102   */
103  if (snorkel_obj_set (http,    /* server object */
104                       snorkel_attrib_uri,      /* attribute type */
105                       GET,     /* method */
106                       "/index.html",   /* url */
107                       , encodingtype_text, index_html) != SNORKEL_SUCCESS)
108    {
109      perror ("could not overload index.html");
110      snorkel_obj_destroy (http);
111      exit (1);
112    }
```

```
113
114     if (snorkel_obj_set
115         (http, snorkel_attrib_ipvers, IPVERS_IPV4,
116          SOCK_SET) != SNORKEL_SUCCESS)
117       {
118         fprintf (stderr, "error could not set ip version\n");
119         exit (1);
120       }
121
122     /*
123      *
124      * start the server
125      *
126      */
127     fprintf (stderr, "\n\n[HTTP] starting embedded server\n");
128     if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
129       {
130         perror ("could not start server\n");
131         snorkel_obj_destroy (http);
132         exit (1);
133       }
134
135     /*
136      *
137      * do something while server runs
138      * as a separate thread
139      *
140      */
141     fprintf (stderr, "\n[HTTP] started.\n\n"
142              "--hit enter to terminate--\n");
143     fgets (szExit, sizeof (szExit), stdin);
144
145     fprintf (stderr, "[HTTP] bye\n");
146
147     /*
148      *
149      * graceful clean up
150      *
151      */
152     snorkel_obj_destroy (http);
153     exit (0);
154   }
```

In this example, we begin by defining the function **index_html**, our callback function. When a handler thread receives a request for a URI, it first checks thread local storage for a callback associated with the URI, a matching MIME, or a content buffer. When a callback association exists, the handler-thread passes the request along with the connection to the callback routine for it to provide the content portion of the HTTP response.

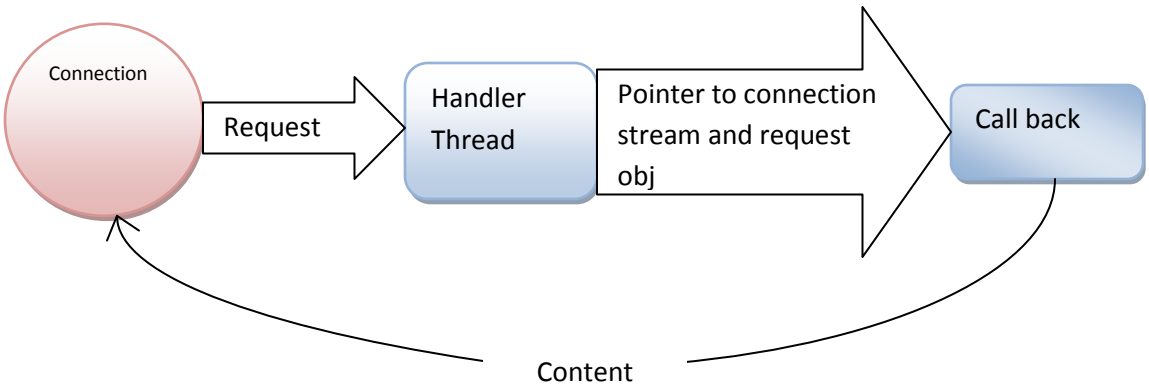


Figure 7, simplified version of how callbacks process content

In this example, a request for the URI "/index.html" equals a call to the function **index_html**. Figure eight, illustrates the syntax for URI callbacks.

```
#include <snorkel.h>

callback_status_t  (* snorkel_uri_callback_t)   (snorkel_obj_t http_request, snorkel_obj_t outputstream,
char *URI)

                                                    Return a value of one for success and zero on failure
```

Figure 8, callback prototype for URI overloading

In figure 8, *http_request* is a HTTP request object and *outstream* is an opened stream connected to the client- browser. Figure nine, illustrates the relationship between handler-threads and URI overloads.

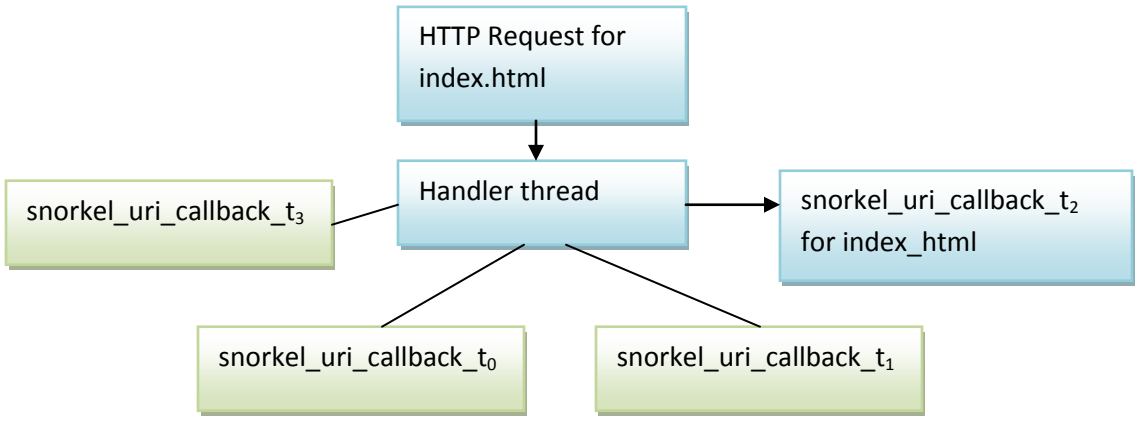


Figure 9, relationship between URI-overload and handler threads

In our callback **index_html,** we write our response/content to the output stream passed by the handler-thread using the function **snorkel_printf**.

> *if (snorkel_printf (outstream, "<html><body><h2>Dynamic Content</h2></body></html>\r\n") == SNORKEL_ERROR)*

The **snorkel_printf** function is syntactically similar to C's **fprintf** function and is ideal for sending text data. Figure 10, illustrates the syntax of the **snorkel_printf** function.

```
#include <snorkel.h>

int snorkel_printf  (snorkel_obj_t outstream,  char *pszformat, …)

                                    Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 10, the snorkel_printf function

We check the return value of **snorkel_printf** against the value SNORKEL_ERROR. If the **snorkel_printf** function fails, we return **HTTP_ERROR** to let the runtime know that an error has occurred. Returning **HTTP_ERROR** from a URI callback generates an error page containing a generic error in the client's browser. You can override the generic error message, by returning an error using the macro **ERROR_STRING**. The macro takes a pointer to a constant character string. If no errors occur in the callback, we return **HTTP_SUCCESS**.

To associate our function, **index_html**, with the URI "/index.html" we call the function **snorkel_obj_set** from **main**, lines 103-107.

> *snorkel_obj_set (server_obj, snorkel_attrib_uri, GET, "/index.html", encodingtype_text, index_html)*

In the call, we pass the server object, the HTTP method, the attribute that we want to set, the URI, the encoding type, and a pointer to the callback function *index_html*.

The HTTP method, in the call, identifies the method used by the client-browser to request the resource. The server will only call the function if both the method and URI match our association. As mentioned earlier, Snorkel supports both the HTTP-GET and HTTP-POST methods. When overriding a URI it is important not to confuse the two. In general, you overload a URI using POST if a form references the URI. Otherwise, you use GET.

In addition to identying the method, we must also identify the encoding. There are two types of encoding **encodingtype_text** and **encodingtype_binary**. The encoding type does not tell the runtime how to encode the data but how to send the data – as either binary or text. The method used for encoding text, as you will see later, is up to the application. For example, there is no encoding type uu-encode.

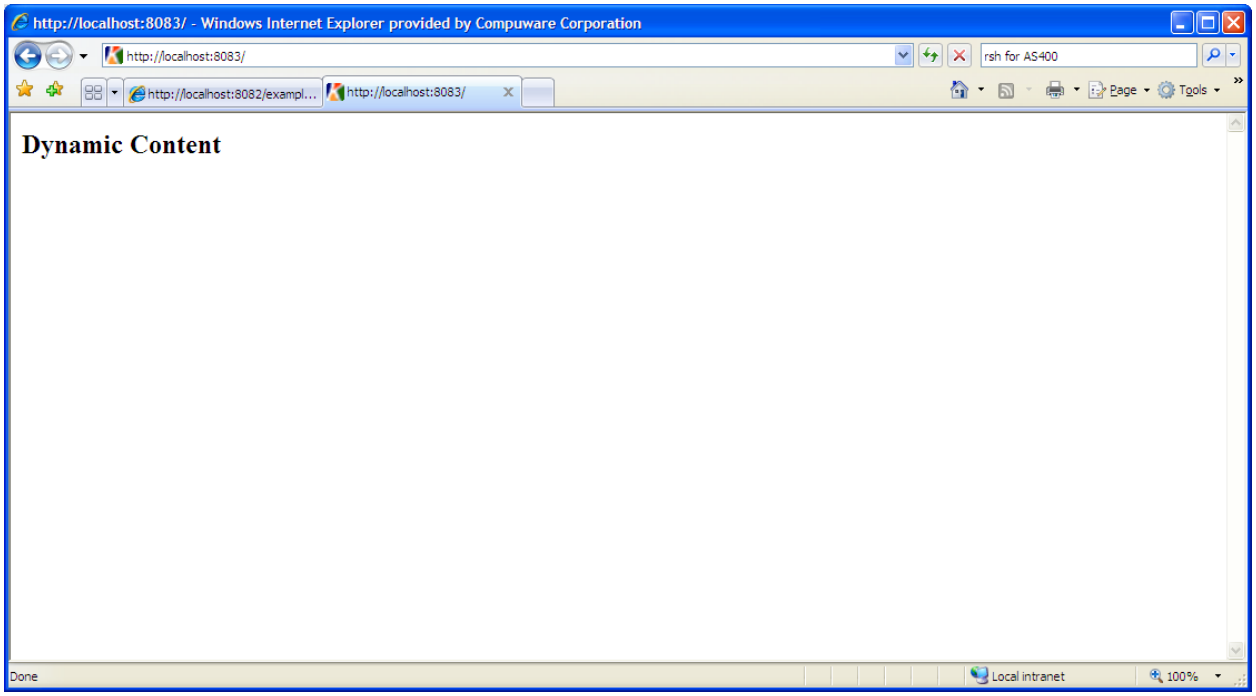
The **snorkel_obj_set** API that we used in this example, to associate URIs, acts as an interface for changing or amending any modifyable Snorkel objects' attributes. Both **snorkel_obj_create** and **snorkel_obj_set** are object-agnostic and argument lists are dependent on the object type requested or provided. As mentioned earlier, Snorkel objects contain both data and a description of the data for processing by the runtime. Most Snorkel functions are object-agnostic. This provides backwards compatibility for future releases of the API.

```
#include <snorkel.h>

int snorkel_obj_set  (snorkel_obj_t object,  snorkel_obj_attrib_t attrib, …)
```

Figure 11, general syntax of snorkel_obj_set

As with the previous examples, we block on user input to determine when to exit.

Example 2-1, Output



2.2 Overloading HTTP-POST

Overloading a URI request generated by an HTTP-POST is not much different from handling a HTTP-GET request. For the next example, we are going to create a HTML form that requests a user's first and last name. To respond to the HTTP-POST request we will add a callback that processes the information and presents it back to a client's browser.

Example 2-2

```
1  /*
2
3  example, overloads a form URL and its corresponding POST
4
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <snorkel.h>
9
10  /*
11   content buffer-containing form
12  */
```

```
13  char szForm[] =
14    { "<html>\r\n<body>\r\n"
15  "<form method=\"post\" action=\"postform.html\">\r\n"
16  "First name:\r\n"
17  "<input type=\"text\" name=\"firstname\">\r\n"
18  "<br>\r\n"
19  "Last name:\r\n"
20  "<input type=\"text\" name=\"lastname\">\r\n"
21  "<input type=\"submit\" value=\"submit\">\r\n"
22  "</form>\r\n" "</body>\r\n</html>\r\n"
23  };
24
25  call_status_t
26  postform (snorkel_obj_t http,   /* read environment from this object */
27            snorkel_obj_t outstream       /* write data to the output stream   */
28    )
29  {
30    char szlastname[80];
31    char szfirstname[80];
32
33    if (snorkel_obj_get
34        (http, snorkel_attrib_post, "firstname", szfirstname,
35         sizeof (szfirstname)) != SNORKEL_SUCCESS)
36      return HTTP_ERROR;
37    if (snorkel_obj_get
38        (http, snorkel_attrib_post, "lastname", szlastname,
39         sizeof (szlastname)) != SNORKEL_SUCCESS)
40      return HTTP_ERROR;
41
42    snorkel_printf (outstream,
43                    "<html><body>your first name is %s and your last name is %s</body></html>",
44                    szfirstname, szlastname);
45
46    return HTTP_SUCCESS;
47  }
48
49
50  void
51  syntax (char *pszProg)
52  {
53    fprintf (stderr, "syntax error:\n");
54    fprintf (stderr, "%s [-p <port>] [-l <log>]\n", pszProg);
55    exit (1);
56  }
57
58
59  void
60  main (int argc, char *argv[])
61  {
62    int i = 1;
63    int port = 80;
64    char *pszIndex = 0;
65    char *pszLog = 0;
66    char szExit[10];
67    snorkel_obj_t logobj = 0;
68    snorkel_obj_t http = 0;
```

```
69
70    for (; i < argc; i++)
71      {
72        if (argv[i][0] == '-' || argv[i][0] == '/')
73          {
74            char carg = argv[i][1];
75
76            switch (carg)
77              {
78              case 'p':
79                port = atoi (argv[i + 1]);
80                i++;
81                break;
82              case 'l':
83                pszLog = argv[i + 1];
84                i++;
85                break;
86              default:
87                syntax (argv[0]);
88                break;
89              }
90          }
91
92       }
93
94     /*
95      *
96      * create a log file object
97      * for logging
98      *
99      */
100    if (pszLog)
101      {
102        logobj = snorkel_obj_create (snorkel_obj_log, pszLog);
103        if (!logobj)
104          {
105            perror ("could not create log file\n");
106            exit (1);
107          }
108        snorkel_debug (1);
109      }
110
111
112     /*
113      *
114      * initialize the API
115      *
116      */
117    if (snorkel_init () != SNORKEL_SUCCESS)
118      {
119        perror ("could not initialize snorkel\n");
120        exit (1);
121      }
122
123     /*
124      *
```

```
125     * create our server object
126     *
127     */
128    http = snorkel_obj_create (snorkel_obj_server, 4, NULL);
129
130    if (!http)
131      {
132        perror ("could not create http server\n");
133        exit (1);
134      }
135
136    /*
137     *
138     * set up our listener
139     *
140     */
141    if (snorkel_obj_set (http,
142                         snorkel_attrib_listener,
143                         port, 0) != SNORKEL_SUCCESS)
144      {
145        perror ("could not establish listener\n");
146        snorkel_obj_destroy (http);
147        if (logobj)
148          snorkel_obj_destroy (logobj);
149        exit (1);
150      }
151
152    if (snorkel_obj_set
153        (http, snorkel_attrib_ipvers, IPVERS_IPV4,
154         SOCK_SET) != SNORKEL_SUCCESS)
155      {
156        fprintf (stderr, "error could not set ip version\n");
157        exit (1);
158      }
159    /*
160     *
161     * overload some URLs
162     *
163     */
164    if (snorkel_obj_set (http, snorkel_attrib_uri_content, GET,   /* method */
165                         "/index.html",   /* URL to overload */
166                         szForm,  /* pointer to content buffer */
167                         SNORKEL_STORE_AS_REF     /* do not duplicate */
168        ) != SNORKEL_SUCCESS)
169      {
170        perror ("could not overload index.html");
171        snorkel_obj_destroy (http);
172        if (logobj)
173          snorkel_obj_destroy (logobj);
174        exit (1);
175      }
176
177    if (snorkel_obj_set (http,
178                         snorkel_attrib_uri, POST,
179                         "/postform.html",
180                         encodingtype_text,
```

```
181                     postform) != SNORKEL_SUCCESS)
182       {
183         perror ("could not overload postform.html");
184         snorkel_obj_destroy (http);
185         if (logobj)
186           snorkel_obj_destroy (logobj);
187         exit (1);
188       }
189
190     fprintf (stderr,
191           "\n\n[HTTP] starting on port %d....", port);
192     if (pszLog)
193       {
194         fprintf (stderr,
195               "\n[HTTP] logging messages to %s\n", pszLog);
196       }
197     if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
198       {
199         perror ("could not start server\n");
200         snorkel_obj_destroy (http);
201         if (logobj)
202           snorkel_obj_destroy (logobj);
203         exit (1);
204       }
205
206     fprintf (stderr, "\n[HTTP] started.\n\n"
207           "--hit enter to terminate--\n");
208     fgets (szExit, sizeof (szExit), stdin);
209
210     fprintf (stderr, "[HTTP] bye\n");
211     snorkel_obj_destroy (http);
212     if (logobj)
213       snorkel_obj_destroy (logobj);
214     exit (0);
215   }
```

We begin with **main**. In this example, in addition to a port number the user can provide a log file name. To create the log we call the function **snorkel_obj_create**.

logobj = snorkel_obj_create (snorkel_obj_log ,pszLog)

```
#include <snorkel.h>

snorkel_obj_t  snorkel_obj_create  (snorkel_obj_type_t snorkel_obj_log, char *log_file_name)

                                   Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 12, syntax for creating a log file

The Snorkel runtime uses log files to provide information about runtime errors and runtime states. Applications can also write information to a log using the **snorkel_printf** function – passing in the log object as the stream parameter. For additional feedback, we enable debug logging using the **snorkel_debug** function.

snorkel_debug (1)

```
#include <snorkel.h>

int snorkel_debug (int enable)

                                    Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 13, providing a value of 1 enables debug logging and 0 disables it

```
[16/Jan/2010:14:17:00] - root directory set to .
[16/Jan/2010:14:17:00] - (snorkel_obj_start, 12775): checking for bubbles in
C:\Users\Compuware\Desktop\snorkel\release\bubbles...
[16/Jan/2010:14:17:00] - (snorkel_obj_start, 14379): loading bubble
C:\Users\Compuware\Desktop\snorkel\release\bubbles\csource.bbl
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14380):    reading registration data...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14608):    reading symbols...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14630):      symbol CSourceMime....
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14657):      creating property associations...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(c,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(c,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(cpp,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(h,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(hpp,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14665):        association: CSourceMime = MIME(java,text/html)...
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14714):      done
[16/Jan/2010:14:17:00] - (snorkel_bubble_load, 14718): done
[16/Jan/2010:14:17:00] - (snorkel_http_kernel, 12340): primary thread coming up...
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7141): attempting bind to port 80...
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7217):     -configuring AF_INET6 connection
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7334):     -sendbuffer: 8192, recievebuffer: 8192
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7283):     -configuring AF_INET connection
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7334):     -sendbuffer: 8192, recievebuffer: 8192
[16/Jan/2010:14:17:00] - (snorkel_socket_bind, 7366): port 80 ok
[16/Jan/2010:14:17:00] - (snorkel_socket_listen, 7048): attempting to listen on 80...
[16/Jan/2010:14:17:00] - (snorkel_socket_listen, 7091): port 80 ok
```

```
[16/Jan/2010:14:17:00] - (snorkel_http_kernel, 12358): starting handlers...
[16/Jan/2010:14:17:00] - (snorkel_http_handler, 12221): thread 0x14C8 waiting on connection...
[16/Jan/2010:14:17:00] - (snorkel_http_handler, 12221): thread 0x3E4 waiting on connection...
[16/Jan/2010:14:17:00] - (snorkel_http_kernel, 12385): all handlers started
[16/Jan/2010:14:17:00] - (snorkel_http_kernel, 12387): kernel entered primary loop
[16/Jan/2010:14:17:02] - (snorkel_thread_kill, 5347): thread did not respond, forcing down thread
[16/Jan/2010:14:17:03] - (snorkel_http_handler, 12311): thread 0x3E4 exited
[16/Jan/2010:14:17:03] - (snorkel_http_kernel, 12392): kernel signaled to exit
```

Figure 14, sample log listing

A synchronization object is associated with each log object created and only one thread can write to a log at a time. This is important to note since frequent writes to log objects from multiple threads can act as a bottleneck, degrading application performance. In general, keep log writes to a minimum.

Following log file creation we initialize the API by calling **snorkel_init**. If you have not noticed, we called **snorkel_obj_create** for our log object prior to calling **snorkel_init**. Creating a log object is one of the few API calls allowed prior to initializing the Snorkel runtime.

After initializing the runtime, we call the **snorkel_obj_create** function to create our server object. Unlike previous examples, we leave the index directory NULL. The index directory is not required for this example since the application generates all web content.

After defining the listener, we overload the URI "index.html". In this example, the index-URI is associated with a content buffer.

*If (snorkel_obj_set (*http*, snorkel_attrib_url_content, GET*,
"/index.html", szForm*, SNORKEL_STORE_AS_REF) != SNORKEL_SUCCESS)*

As mentioned earlier, content buffers are buffers that contain HTTP-content. In this example, we use a fixed content buffer, *szForm* that we define on line 13. To associate the URI with the content buffer we call the function **snorkel_obj_set** passing in our server object, the attribute type *snorkel_attrib_url_content*, the keyword **GET**, the URI, a pointer to our content buffer, and the keyword **SNORKEL_STORE_AS_REF**. We specified **SNORKEL_STORE_AS_REF** to instruct the runtime to store the address of the content buffer instead of a copy of its content. The difference between the two methods relates to performance. **SNORKEL_STORE_AS_DUP**, the alternative option, creates a separate copy of the content buffer for each threads local-storage and **SNORKEL_STORE_AS_REF** stores a reference to a global pointer in each threads local-storage. The **SNORKEL_STORE_AS_DUP** option takes advantage of NUMA topology, that is, the proximity of memory with respect to a thread's hosting CPU. This methodology insures the minimum distance of travel for thread memory access. When determining the attribute to use, it boils down to a choice between performance and memory consumption.

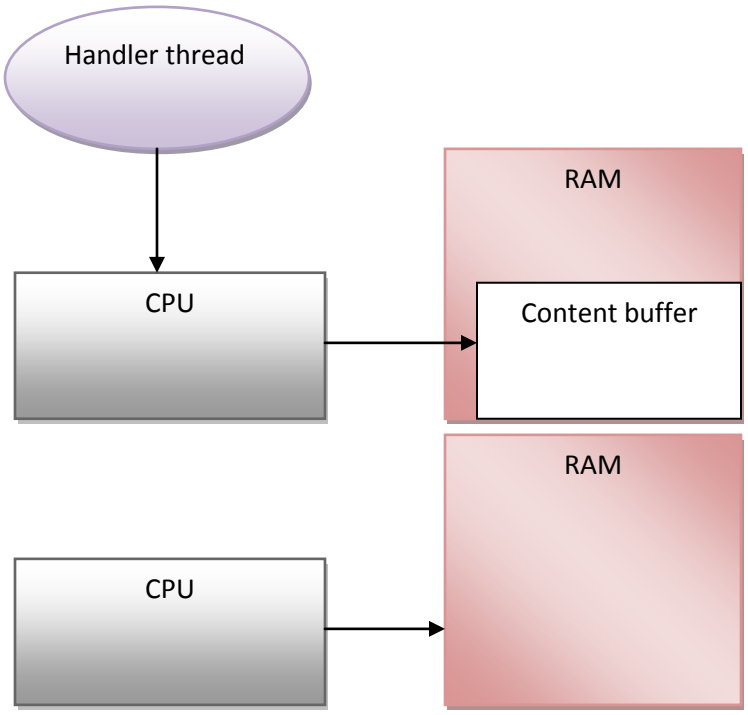


Figure 15, illustrates proximity for chipsets that support NUMA (SNORKEL_STORE_AS_DUP)

#include <snorkel.h>

int snorkel_obj_set(snorkel_obj_t *server_object*, snorkel_attrib_t ***snorkel_attrib_url_content***, int ***GET***, char **URI*, char **pointer_to_content*, int ***SNORKEL_STORE_AS_REF*** or ***SNORKEL_STORE_AS_DUP***)

Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error

Figure 16, syntax for setting a content buffer

Finally, we associate our callback *postform* with the URI "/postform.html" using **snorkel_obj_set** specifying **POST** as our HTTP method and identifying the file as text encoded.

*if (snorkel_obj_set (*http*, snorkel_attrib_uri, POST*, "/postform.html"*,*
encodingtype_text, postform*) != SNORKEL_SUCCESS)*

Unlike the *snorkel_attrib_uri_content,* which determines the encoding type based on the URI extension, you must provide the content type for *snorkel_attrib_uri.* This provides more flexability in URI naming. For example, we could have called our URI "/accept_button" which would have been a more descriptive name identifying the action performed by the client to request the URI.

In the function *postform*, as with all URI callbacks, the runtime passes both the HTTP request object and the output stream. We use the HTTP request object, *http*, in conjunction with the **snorkel_obj_get** function to acquire the user inputs for *firstname* and *lastname.*

> *if (snorkel_obj_get (http, snorkel_attrib_post, "firstname", szfirstname, sizeof (szfirstname)) != SNORKEL_SUCCESS)*
>
> *if (snorkel_obj_get (http, snorkel_attrib_post, "lastname", szlastname, sizeof (szlastname) != SNORKEL_SUCCESS)*

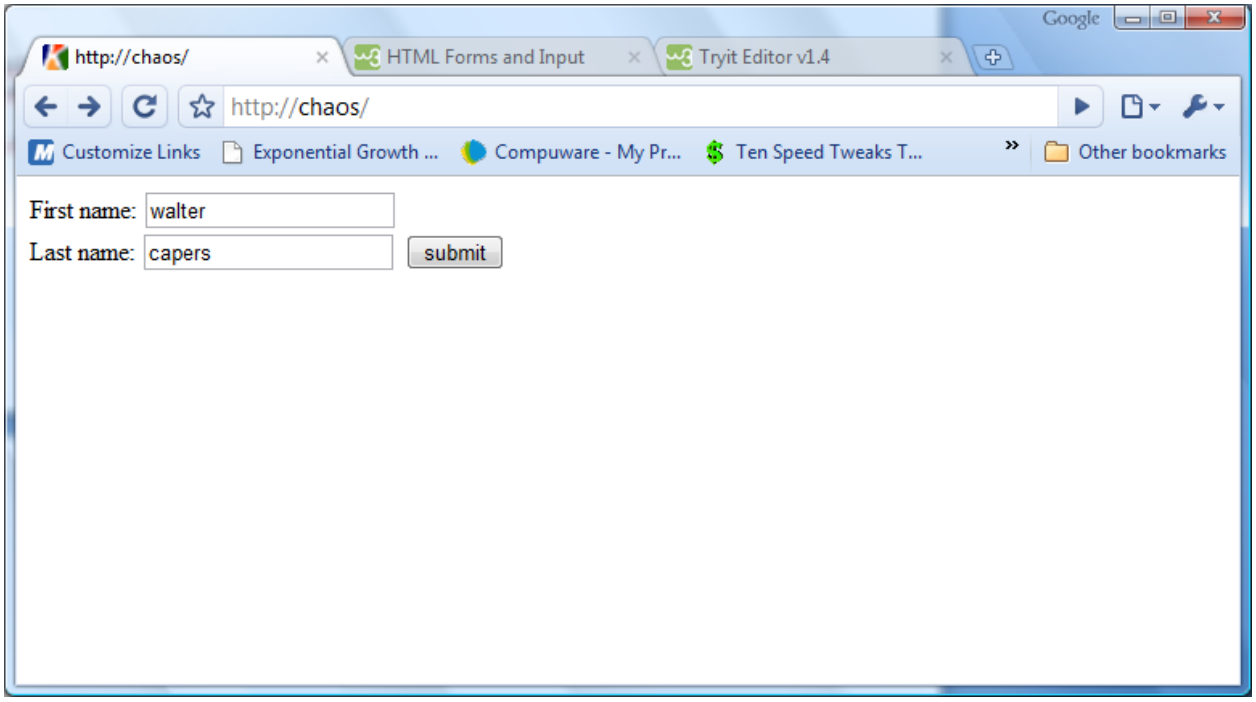
```
#include <snorkel.h>

int snorkel_obj_get (snorkel_obj_t http_request_object, snorkel_attrib_t snorkel_attrib_post,
char *variable_name, char *buffer_to_receive_value, size_t size_of_buffer)
```

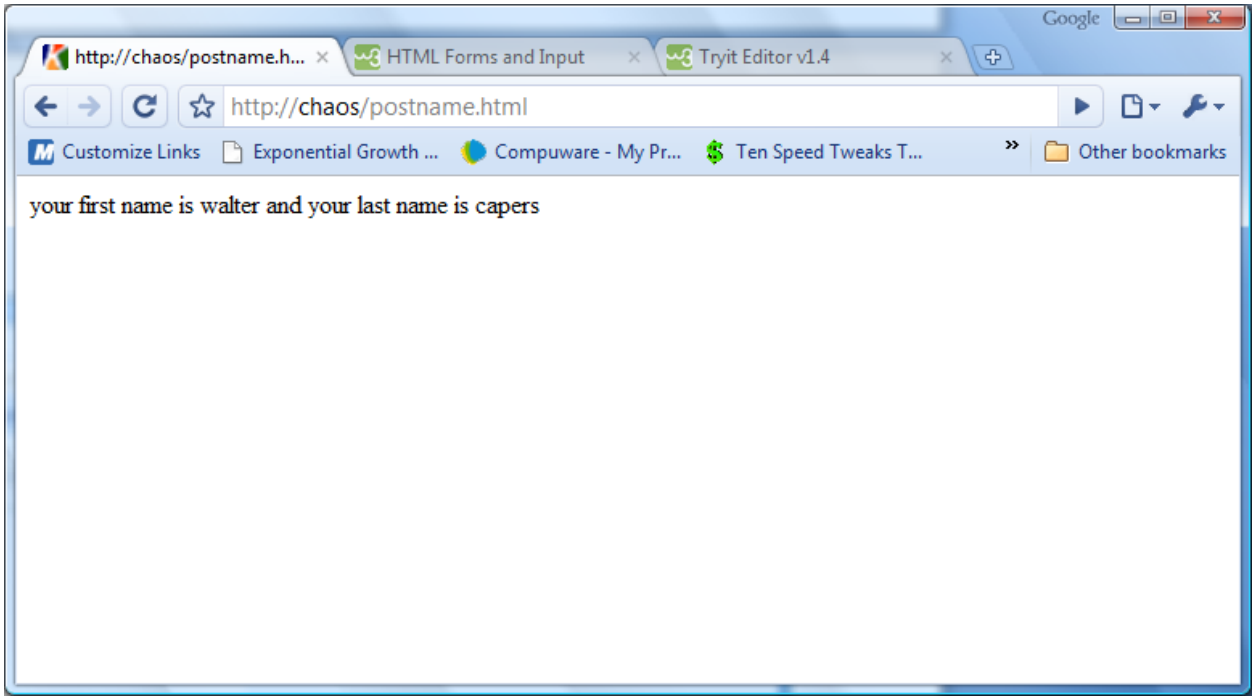
Figure 17, using snorkel_obj_get to read request variables

Prior to returning success from our function *postform*, we echo back the end user's entries using the **snorkel_printf** function, displaying them in the client's browser.

> *snorkel_printf (outstream, "<html><body>your first name is %s and your last name is %s</body></html>", szfirstname, szlastname)*

Example 2-2 Output





2.3 Content Caching

Content caching is another form of URI overloading. It involves associating an overloaded URI with a file. The file is loaded into memory at server startup. Caching content reduces I/O bottlenecks (caused by large usage volumes) by sacrificing memory. As with using content buffers, developers can choose to load copies of a file into each thread's local-storage or use a reference to content located in an application's global-storage.

```
#include <snorkel.h>

int snorkel_obj_set(snorkel_obj_t server_object, snorkel_attrib_t snorkel_attrib_url_cache, int GET,
char *URL, char *filename,  int SNORKEL_STORE_AS_REF or SNORKEL_STORE_AS_DUP)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 18, syntax for caching files

To cache a file, you call the function **snorkel_obj_set** with the attribute ***snorkel_attrib_uri_cache***. Only server objects that have the index/root directory defined (the second parameter provided when creating a server object, see page 11) can cache files. As with content buffers, the last parameter identifies the storage schema. The option **SNORKEL_STORE_AS_DUP** creates copies of the content for each handler thread, storing it in each handler-thread's local storage. The option **SNORKEL_STORE_AS_REF** provides each handler-thread with a reference to the content. Again, the storage schema used should be dependent on memory resources and the frequency of access for the resource. Caching files provides the ability to provide modifyable static content with the performance of memory based content.

2.4 Overloading MIMEs

While the ability to overload individual URIs is useful, there are cases in which you may want to handle all URIs of the same type the same way. Web-servers distinguish URI types by their extensions. For example, web-servers handle all files with the extensions *htm* or *html* as HTML data. The server uses a file's extension to determine its MIME type. MIME types identify the type of data referenced by a URI. In general, MIME overload-functions are data interpreters that convert files from a non-displayable browser format to a displayable one. Here are a few reasons you might want to overload a MIME:

- The extension of a file matches a known MIME type but the extension is not registered within the Snorkel API

- A MIME type is not supported by the Snorkel API
- To override an existing MIME behavior to facilitate file pre-processing

When you overload a MIME type with a function, the runtime calls the function for all URIs containing a matching extension.

In the next example, we are going to address a common problem, displaying C-source files in a browser. Many languages include characters that do not display as intended in a browser. For example, C interprets the characters "<" and ">" as less than or greater than or as delimiters surrounding an included header file. The browser interprets these characters as the beginning and ending of HTML/XML tags. To correct this issue, all "<" and ">" characters must be replaced with "<" and ">" respectively to properly display in a browser. In addition to "<" and ">" there are other special characters that the browser interprets differently than a C interpreter.

In this example, we are going to define an overloaded MIME type for URIs that have the extensions "c", "cpp", "h", "hpp", and java. The overload function will perform the following preprocessing during the transmission of a source file to a browser:

- Escape all special characters
- Provide line numbers for each line of source
- Apply colors to comments and reserved words

Example 2-3

```
1   /*
2
3   example, overloads a form URL and its corresponding POST
4
5   */
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <snorkel.h>
10
11   /*
12    content buffer containing index
13   */
14   char szIndex[] = {
15     "<html>\r\n"
16     "<body>\r\n"
17     "<a href=\"/mimeoverload.c\">example four source file</a>\r\n"
18     "</body>\r\n" "</html>\r\n"
19   };
20
21
22   char *reservedwords[] =
```

```
23    { "#include", "int", "float", "struct", "long", "true",
24    "bool", "false", "boolean", "double", "char", "unsigned",
25      "return", "#define", "break",
26    "defined", "#pragma", "union", "short", "#ifdef",
27      "#ifndef", "static", "#if", "#else",
28    "#endif", "endif", "if", "else", "switch", "else if",
29      "#elif", "const", "size_t",
30    "void", "enum", "typedef", "for", "case", "while", "do",
31      "public", "private", "signed",
32    "default", "goto", "volatile", "sizeof", "delete",
33      "class", "inline", "friend",
34    "namespace", "new", "operator", "private", "protected",
35      "public", "template",
36    "this", "throw", "try", "using", "abstract", "implements",
37      "throws",
38    "transient", "byte", "package", "synchronized", "extends",
39      "instanceof", "strictfp",
40    "catch", "final", "interface", "finally", "continue",
41      "super", "assert", "null", "NULL",
42    "exit", "extern", "public:", "private:", "protected:",
43      "private", "public", "protected",
44    "virtual", "false", "true", "register"
45  };
46
47  size_t chreservedwords =
48    sizeof (reservedwords) / sizeof (char *);
49  typedef char urlesc_t[10];
50
51  char *
52  char_esc (char c, urlesc_t escp)
53  {
54
55    escp[0] = 0;
56    if (c < 0 || c > 255)
57      return 0;
58
59    if (isalnum ((int) c) || c == '/' || c == '*' || c == '+'
60        || c == '-' || c == '#' || c == ' ' || c == '\\'
61        || c == '\n' || c == '\t' || c == '\r')
62      {
63        return 0;
64      }
65    sprintf_s (escp, sizeof (urlesc_t), "&#%d;", c);
66
67    return (char *) escp;
68  }
69
70
71
72
73  void
74  print_line (snorkel_obj_t obj,
75              int line,
76              int *pcomment,
77              char *pszin, char *pszout, size_t cb)
78  {
```

```
79    size_t i = 0;
80    char *psz = pszout;
81    int quot = 0;
82    int backslash = 0;
83    char green[] = "</font><font color=\"#008000\">";
84    char black[] = "</font><font color=\"#000000\">";
85    char blue[] = "</font><font color=\"#0000FF\">";
86    char red[] = "</font><font color=\"#C43100\">";
87    char brown[] = "</font><font color=\"#AA6600\">";
88    size_t lgreen, lblue, lblack, lred, lbrown;
89    urlesc_t escp;
90    char *pszstart = pszin;
91
92    memset (pszout, 0, cb);
93
94    if (strstr (pszin, "int") != 0)
95      {
96        int it = 0;
97        it = 1;
98      }
99
100   snorkel_printf (obj,
101                   "<font color=\"#000000\">% 6d</font>    ",
102                   line);
103   if (*pcomment)
104     snorkel_printf (obj, "<font color=\"#008000\">");
105   else
106     snorkel_printf (obj, "<font color=\"#000000\">");
107
108   lgreen = strlen (green);
109   lblue = strlen (blue);
110   lblack = strlen (black);
111   lred = strlen (red);
112   lbrown = strlen (brown);
113
114   for (; i < cb && *pszin != 0;)
115     {
116
117       if (strncmp (pszin, "/*", 2) == 0 && !*pcomment
118           && !quot)
119         {
120           *pcomment = 1;
121           if (i + lgreen + 2 > cb)
122             return;
123           strcat (psz, green);
124           strcat (psz, "/*");
125           psz += lgreen + 2;
126           pszin += 2;
127           i += lgreen + 2;
128           backslash = 0;
129         }
130       else if (strncmp (pszin, "//", 2) == 0 && !*pcomment
131                && !quot)
132         {
133           if (i + lgreen + 2 > cb)
134             return;
```

```
135           strcat (psz, green);
136           strcat (psz, "//");
137           psz += lgreen + 2;
138           pszin += 2;
139           i += lgreen + 2;
140           backslash = 0;
141         }
142       else if (*pszin == '\"')
143         {
144
145           if (*pcomment || backslash)
146             {
147               if (i + 5 > cb)
148                 return;
149               backslash = 0;
150               strcat (psz, char_esc (*pszin++, escp));
151               psz += 5;
152               i += 5;
153             }
154           else if (!quot)
155             {
156               if (i + 5 + lred > cb)
157                 return;
158               strcat (psz, red);
159               strcat (psz, char_esc (*pszin++, escp));
160               quot = 1;
161               psz += 5 + lred;
162               i += 5 + lred;
163             }
164           else
165             {
166               quot = 0;
167               if (i + 5 + lblack > cb)
168                 return;
169               strcat (psz, char_esc (*pszin++, escp));
170               strcat (psz, black);
171               psz += 5 + lblack;
172               i += 5 + lblack;
173             }
174
175         }
176       else if (strncmp (pszin, "*/", 2) == 0 && *pcomment)
177         {
178           backslash = 0;
179           *pcomment = 0;
180           if (i + lblack + 2 > cb)
181             return;
182           strcat (psz, "*/");
183           strcat (psz, black);
184           pszin += 2;
185           psz += lblack + 2;
186         }
187       else
188         {
189           size_t k = chreservedwords;
190           if (!*pcomment && !quot)
```

```
191             {
192
193
194               for (k = 0; k < chreservedwords; k++)
195                 {
196                   if ((strncmp (pszin, reservedwords[k],
197                                strlen (reservedwords[k]))
198                       == 0)
199                      && (pszin[strlen (reservedwords[k])]
200                          == 0
201                          ||
202                          pszin[strlen (reservedwords[k])]
203                          == '{'
204                          ||
205                          pszin[strlen (reservedwords[k])]
206                          == '\t'
207                          ||
208                          pszin[strlen (reservedwords[k])]
209                          == ' '
210                          ||
211                          pszin[strlen (reservedwords[k])]
212                          == '\n'
213                          ||
214                          pszin[strlen (reservedwords[k])]
215                          == '('
216                          ||
217                          pszin[strlen (reservedwords[k])]
218                          == ')'
219                          ||
220                          pszin[strlen (reservedwords[k])]
221                          == ':'
222                          ||
223                          pszin[strlen (reservedwords[k])]
224                          == ';'
225                          ||
226                          pszin[strlen (reservedwords[k])]
227                          == '<') && (pszin == pszstart
228                                      || (pszin - 1)[0] ==
229                                      ' '
230                                      || (pszin - 1)[0] ==
231                                      '\t'
232                                      || (pszin - 1)[0] ==
233                                      '('))
234                     break;
235                 }
236
237             }
238
239           if (k < chreservedwords)
240             {
241               size_t l = strlen (reservedwords[k]);
242               if (i + l + lblue + lblack > cb)
243                 return;
244               strcat (psz, blue);
245               strcat (psz, reservedwords[k]);
246               strcat (psz, black);
```

```
247
248               psz += l + lblue + lblack;
249               pszin += strlen (reservedwords[k]);
250               i += l + lblue + lblack;
251               backslash = 0;
252             }
253           else
254             {
255               if (char_esc (*pszin, escp))
256                 {
257                   size_t l = strlen ((char *) escp);
258                   if (i + l > cb)
259                     return;
260                   strcat (psz, escp);
261                   psz += l;
262                   i += l;
263                   pszin++;
264                 }
265               else
266                 {
267                   *psz = *pszin++;
268                   if (*psz == '\\')
269                     backslash = 1;
270                   else
271                     backslash = 0;
272                   psz++;
273                   i++;
274                 }
275             }
276         }
277     }
278   snorkel_printf (obj, "%s</font>\r\n", pszout);
279
280   return;
281 }
282
283 call_status_t
284 CSourceMime (snorkel_obj_t http_req,
285              snorkel_obj_t connection, char *pszurl)
286 {
287   FILE *fd = fopen (pszurl, "r");
288   char szbuffer[512];
289   char szencoded[2048];
290   int line = 0;
291   int comment = 0;
292
293   if (!fd)
294     return ERROR_STRING ("could not open source file!\r\n");
295
296   snorkel_printf (connection,
297                   "<html><body><a href=\"edit:%s\">Edit<a><hr><pre><code>\r\n");
298
299   while (!feof (fd))
300     {
301       line++;
302       if (fgets (szbuffer, sizeof (szbuffer), fd))
```

```
303         {
304           szbuffer[strlen (szbuffer) - 1] = 0;
305           print_line (connection,
306                       line, &comment, szbuffer, szencoded,
307                       sizeof (szencoded));
308         }
309       }
310     fclose (fd);
311     snorkel_printf (connection,
312                     "</code></pre></body></html>\r\n");
313     return HTTP_SUCCESS;
314   }
315
316   void
317   syntax (char *pszProg)
318   {
319     fprintf (stderr, "syntax error:\n");
320     fprintf (stderr, "%s [-p <port>] [-l <log>]\n", pszProg);
321     exit (1);
322   }
323
324
325   void
326   main (int argc, char *argv[])
327   {
328     int i = 1;
329     int port = 8080;
330     char *pszIndex = 0;
331     char *pszLog = 0;
332     char szExit[10];
333     snorkel_obj_t logobj = 0;
334     snorkel_obj_t http = 0;
335
336     for (; i < argc; i++)
337       {
338         if (argv[i][0] == '-' || argv[i][0] == '/')
339           {
340             char carg = argv[i][1];
341
342             switch (carg)
343               {
344               case 'p':
345                 port = atoi (argv[i + 1]);
346                 i++;
347                 break;
348               case 'l':
349                 pszLog = argv[i + 1];
350                 i++;
351                 break;
352               default:
353                 syntax (argv[0]);
354                 break;
355               }
356           }
357
358       }
```

```
359
360   if (pszLog)
361     {
362       logobj = snorkel_obj_create (snorkel_obj_log, pszLog);
363       if (!logobj)
364         {
365           perror ("could not create log file\n");
366           exit (1);
367         }
368       snorkel_debug (1);
369     }
370
371
372
373   if (snorkel_init () != SNORKEL_SUCCESS)
374     {
375       perror ("could not initialize snorkel\n");
376       exit (1);
377     }
378
379   http = snorkel_obj_create (snorkel_obj_server, 2, ".");
380
381   if (!http)
382     {
383       perror ("could not create http server\n");
384       exit (1);
385     }
386
387   if (snorkel_obj_set (http,
388                        snorkel_attrib_listener,
389                        port, 0) != SNORKEL_SUCCESS)
390     {
391       perror ("could not create listener\n");
392       snorkel_obj_destroy (http);
393       if (logobj)
394         snorkel_obj_destroy (logobj);
395       exit (0);
396     }
397
398
399   /*
400    *
401    * overload a URL with a content buffer
402    *
403    */
404   if (snorkel_obj_set (http, snorkel_attrib_uri_content,        /* using a content buffer */
405                        GET,     /* method is GET */
406                        "/index.html",   /* URL */
407                        szIndex, /* content buffer */
408                        SNORKEL_STORE_AS_REF)    /* store as reference */
409       != SNORKEL_SUCCESS)
410     {
411       perror ("could not overload index.html");
412       snorkel_obj_destroy (http);
413       if (logobj)
414         snorkel_obj_destroy (logobj);
```

```
415
416       exit (1);
417     }
418
419   /*
420    *
421    * overload a mime
422    *
423    */
424   snorkel_obj_set (http, snorkel_attrib_mime, "c",      /* extension */
425                    "text/html", /* browser interpretation */
426                    encodingtype_text,   /* encoded as text */
427                    CSourceMime);        /* callback */
428
429   snorkel_obj_set (http, snorkel_attrib_mime, "cpp",    /* extension */
430                    "text/html", /* browser interpretation */
431                    encodingtype_text,   /* encoded as text */
432                    CSourceMime);        /* callback */
433
434   snorkel_obj_set (http, snorkel_attrib_mime, "hpp",    /* extension */
435                    "text/html", /* browser interpretation */
436                    encodingtype_text,   /* encoded as text */
437                    CSourceMime);        /* callback */
438
439   snorkel_obj_set (http, snorkel_attrib_mime, "h",      /* extension */
440                    "text/html", /* browser interpretation */
441                    encodingtype_text, CSourceMime);     /* callback */
442
443   snorkel_obj_set (http, snorkel_attrib_mime, "java",   /* extension */
444                    "text/html", /* browser interpretation */
445                    encodingtype_text, CSourceMime);     /* callback */
446
447
448   fprintf (stderr, "\n\n[HTTP] starting on port %d....",
449            port);
450   if (pszLog)
451     {
452       fprintf (stderr, "\n[HTTP] logging messages to %s\n",
453                pszLog);
454     }
455   if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
456     {
457       perror ("could not start server\n");
458       snorkel_obj_destroy (http);
459       if (logobj)
460         snorkel_obj_destroy (logobj);
461       exit (1);
462     }
463
464   fprintf (stderr, "\n[HTTP] started.\n\n"
465            "--hit enter to terminate--\n");
466   fgets (szExit, sizeof (szExit), stdin);
467
468   fprintf (stderr, "[HTTP] bye\n");
469   snorkel_obj_destroy (http);
470   if (logobj)
```

```
471       snorkel_obj_destroy (logobj);
472     exit (0);
473   }
```

We begin with the callback *CSourceMime,* lines 283-322. When a handler-thread calls a mime-callback it passes an HTTP request object, an output stream, and the fully qualified path to the file referenced by the URL.

> *callback_status_t CSourceMime (snorkel_obj_t http_req, snorkel_obj_t connection, char *pszurl)*

In the callback *CSourceMime,* we first stream the beginning of our HTML response using the **snorkel_printf** function. Next, we open the URL and read each line into a buffer passing the buffer to our *print_line* function. The *print_line* function converts the line to HTML and streams it back to the browser.

```
#include <snorkel.h>

callback_status_t  (* snorkel_mime_callback_t)  (snorkel_obj_t http_request, snorkel_obj_t
outputstream, char *path_to_url)

                                                    Return value of 1 on success and zero on failure
```

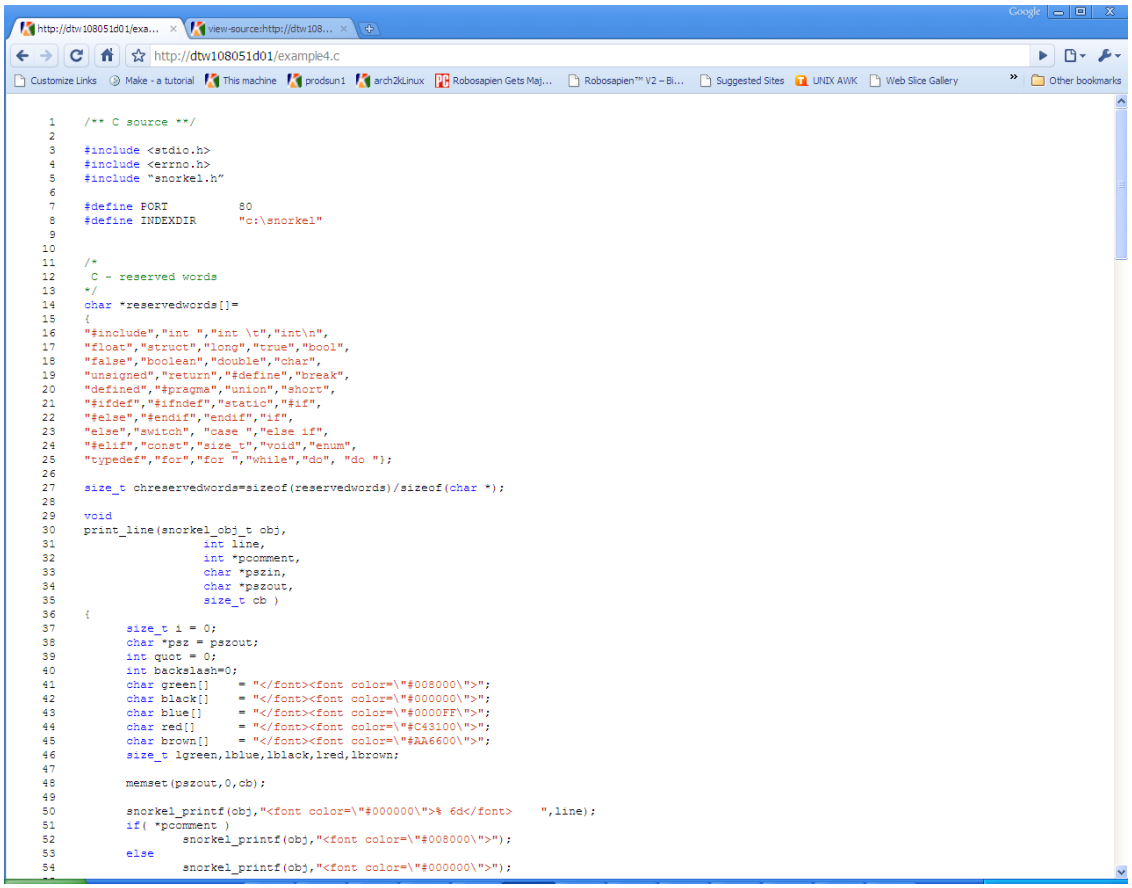
Figure 19, prototype for MIME callback

We register *CSourceMime* in **main** using calls to **snorkel_obj_set**, one call per extension (lines 424-446).

```
#include <snorkel.h>

int  snorkel_obj_set ( snorkel_obj_t server_object, snorkel_attrib_t snorkel_attrib_mime, char
*extension, char * http_description, encodingtype_t encoding_type, snorkel_mime_callback_t
callback_function)

                                    Returns a value of SNORKEL_SUCCESS on success and SNOKEL_ERROR on error
```

Figure 20, syntax for setting a MIME overload

Example 2-3 Output



There are many advantages to overloading MIMEs, but there will be cases in which you may want to create exceptions. An exception is a URI that you want to exclude from a defined MIME overload. To create an exception to an overloaded MIME, overload the URI using the INGORE_MIME keyword. For example,

snorkel_obj_set (server_obj, snorkel_attrib_url, GET, "/nomime.c", IGNORE_MIME)

ignores the URI "/nomime.c". You can also create an exception by creating a URI overload that shares the MIME extension since overloaded URIs take precedence over overloaded MIME types.

2.5 URI Overloading and Wildcards

The Snorkel runtime provides limited wildcard support for URI overloading. Wildcards are useful when specifying groups of files that share comminalities in their names. The following table illustrates the types of wildcard combinations supported by the API.

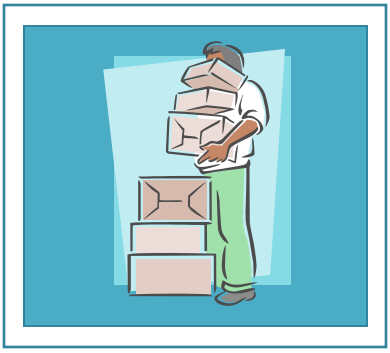
Wildcard Combinations
URI_resource_name.
*URI_resource_name
URI_path/URI_resource_name.*
URI_path *.ext
*URI_resource_name.ext
URI_path*

Table 3, supported wildcard combinations

To understand the deference between wildcarded URIs and non-wildcarded URIs look at the following two code snipts:

```
snorkel_obj_set  (server,
                  snorkel_attrib_uri, GET,
                 "*makefile", encodingtype_text,
                  proc_file);

snorkel_obj_set  (server,
                  snorkel_attrib_uri, GET,
                 "/makefile", encodingtype_text,
                  proc_file);
```

In the first snipet, the function *proc_file* processes any URI containing name *makefile* while the second associates a specific URI-path, */makefile, with proc_file*.



3 – Overloading HTTP, Protocol Stacking

Writing proprietary protocols

Data exchanges between clients and servers do not always have a human element, that is, there may be cases in which automated processes drive data exchanges and/or system interactions. Encapsulating data within the HTTP protocol or XML wastes bandwidth when a user component is not required to interpret the data. Snorkel handler threads are protocol agnostic and are not limited to processing HTTP requests. Overloading HTTP, which we will refer to as Protocol-Stacking, allows a single server instance to process multiple protocols. Protocol stacking eliminates the complexities of tunneling and the need for multiple port openings in a firewall to support a single web-enabled application and its supporting components. In this section, we explore proprietary protocol development.

In our next example, we write a web-server that supports two protocols:

- A simple file transport protocol
- A web page that lets us know if the server is up

The following table illustrates our transport protocol.

Client	Server
1. Sends filename and transmission method	2. Recieves requested filename and transmission method
4. Recieves file open status	3. Sends file open status
6. Recieves requested file	5. Streams requested file using requested transmission method
8. Closes connection	7. Closes connection

Table 4, file get protocol

Example 3-1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <errno.h>
5  #include <sys/stat.h>
6  #include <snorkel.h>
```

```
 7  #include <time.h>
 8
 9
10  int g_sendfile_method = 0;
11
12
13  /*
14   *
15   * server protocol identifier
16   *
17   */
18  #define FGET_PROTOCOL "fileget"
19
20  char index_html[]=
21   "<html><body><H2>File server is up</H2></body></html>";
22
23
24  void
25  path_to_local_path (char *psz)
26  {
27    if (!psz)
28      return;
29
30    for (; *psz!=0; psz++)
31       {
32  #ifdef WIN32
33         if (*psz == '/')
34           *psz = '\\';
35  #else
36         if (*psz == '\\')
37           *psz = '/';
38  #endif
39       }
40  }
41
42
43  call_status_t
44  fileget (snorkel_obj_t outstream)
45  {
46    char *psz_local_file = 0;
47    FILE *fd = 0;
48    struct stat stf;
49    int use_send_file = 0;
50
51
52    if (snorkel_unmarshal (outstream, "zi", &psz_local_file,
53                           &use_send_file) != SNORKEL_SUCCESS)
54      return PROTOCOL_ERROR;
55
56    path_to_local_path (psz_local_file);
57
58    if (stat (psz_local_file, &stf) != 0)
59      {
60        if (snorkel_marshal (outstream, "i", -1) !=
61            SNORKEL_SUCCESS)
62          return PROTOCOL_ERROR;
```

```
63      }
64
65    if (snorkel_marshal (outstream, "i", (int) stf.st_size) !=
66        SNORKEL_SUCCESS)
67      return PROTOCOL_ERROR;
68
69    if (use_send_file && g_sendfile_method == 0)
70      g_sendfile_method =
71        SNORKEL_USE_SENDFILE | SNORKEL_FILE_SEND;
72
73    if (snorkel_file_stream
74        (outstream, psz_local_file,
75         stf.st_size, g_sendfile_method) != SNORKEL_SUCCESS)
76      return PROTOCOL_ERROR;
77
78    return PROTOCOL_SUCCESS;
79  }
80
81  void
82  syntax (char *pszsrv)
83  {
84
85    fprintf (stderr, "syntax error:\n");
86    fprintf (stderr, "%s [-p port] [-s]\n", pszsrv);
87    fprintf (stderr,
88             "-p: specifies port number, default is 8083\n");
89    fprintf (stderr,
90             "-s: use sendfile on UNIX and TransmitFile on Windows\n");
91    exit (1);
92  }
93
94  int
95  main (int argc, char *argv[])
96  {
97
98    int port = 8083;
99    char *pszport = 0;
100     snorkel_obj_t server = 0;
101     char szsignal_exit[10];
102     int i = 0;
103
104
105     for (i = 1; i < argc; i++)
106       {
107         if (*argv[i] == '-')
108           {
109             switch (argv[i][1])
110               {
111               case 'p':         /* port number */
112                 pszport = argv[i + 1];
113                 i++;
114                 break;
115               case 's':         /* sendfile flag */
116                 g_sendfile_method |= SNORKEL_USE_SENDFILE;
117                 break;
118               default:
```

```
119               syntax (argv[0]);
120               break;
121             }
122         }
123     }
124   if (pszport)
125     port = atoi (pszport);
126
127   snorkel_init ();
128
129
130   if (snorkel_init () != SNORKEL_SUCCESS)
131     {
132       perror ("could not initialize server\n");
133       exit (1);
134     }
135
136   server = snorkel_obj_create (snorkel_obj_server, 2, NULL);
137
138   if (!server)
139     {
140       perror ("could not create server, check log\n");
141       exit (1);
142     }
143
144
145   if (snorkel_obj_set (server,
146                        snorkel_attrib_listener,
147                        port, 0) != SNORKEL_SUCCESS)
148     {
149       perror ("could not create listener, check log\n");
150       snorkel_obj_destroy (server);
151       exit (1);
152     }
153
154
155   if (!snorkel_obj_create (snorkel_obj_protocol,
156                            FGET_PROTOCOL, fileget))
157     {
158       perror ("could not implement protocol\n");
159       snorkel_obj_destroy (server);
160       exit (1);
161     }
162
163   if (snorkel_obj_set
164       (server, snorkel_attrib_ipvers, IPVERS_IPV4,
165        SOCK_SET) != SNORKEL_SUCCESS)
166     {
167       perror ("could not set ip version\n");
168       snorkel_obj_destroy (server);
169       exit (1);
170     }
171
172   if (snorkel_obj_set
173       (server, snorkel_attrib_uri_content,
174        GET, "/index.html",
```

```
175         index_html,
176         SNORKEL_STORE_AS_REF) != SNORKEL_SUCCESS)
177       {
178         perror ("could not set index page\n");
179         snorkel_obj_destroy (server);
180         exit (1);
181       }
182
183     fprintf (stderr, "\n\nstarting server");
184
185
186     if (snorkel_obj_start (server) != SNORKEL_SUCCESS)
187       {
188         perror ("could not start server\n");
189         snorkel_obj_destroy (server);
190         exit (1);
191       }
192
193     fprintf (stderr, "\nstarted\n\n");
194     fprintf (stderr, "--hit enter to terminate--\n");
195     fgets (szsignal_exit, sizeof (szsignal_exit), stdin);
196
197     snorkel_obj_destroy (server);
198
199     exit (0);
200   }
```

Beginning with main, we define our file transport protocol using the **snorkel_obj_create** function. To distinquish requests for our protocol from other protocols, the call to **snorkel_obj_create** takes a string, the third parameter, which identifies the protocol by name. If you have not noticed, the call to **snorkel_obj_create** does not return an object. This is because protocol objects are not modifiable and only the runtime can access them.

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create(snorkel_obj_type_t snorkel_obj_protocol, char *pszprotocol_name,
snorkel_proto_callback_t callback)

                                Returns a value of (snorkel_obj_t)1 on success and (snorkel_obj_t)0 on failure
```

Figure 21, syntax for creating a protocol object

The *fileget* function, associated with the protocol string “fileget”, processes all incoming requests for the protocol. On the first executable line of the routine, the server reads the filename and an integer defining the method for transmitting the file back to the client.

```
call_status_t  (*snorkel_proto_callback_t)  (snorkel_obj_t)
```

Figure 22, protocol callback syntax

If the method, the second parameter, is one the server sends the file using the *send file* method; otherwise, it is streams the file using a conventional read write method. The *send file* method uses the **sendfile** system call on UNIX and **TransmitFile** system call on Windows to stream the file. Using the *send file* method takes advantage of Zero-Copy, which provides better throughput and lower CPU utilization in comparison to conventional means of file transport.

To read the input parameters we use the function **snorkel_unmarshal**. **Snorkel_unmarshal** is the binary equivalent of **snorkel_printf**. We could have used **snorkel_printf**; however, **snorkel_printf** converts data to a contiguous string and sending a string consumes more bandwidth and is less efficient than sending data as binary.

Sending binary data across a network can be tricky especially when the system on the receiving end is of a different architecture. This is because, some platforms store numbers significant byte first and others do not. There can also be differences in the storage of floating-point numbers between platforms. Normally, the operation of translating data types between different platform types is a developer task. Snorkel frees the developer from this task by providing a couple of functions that perform the translations behind the scenes. The functions **snorkel_unmarshal** and **snorkel_marshal** perform binary data translations between platforms by exchanging data in a neutral IEEE format. On the receiving end, the runtime converts the data from IEEE to a platform specific format. Like the **snorkel_printf** function, the **snorkel_marshal** function takes a format string followed by a list of variables that correspond with elements in the format string. Unlike the **snorkel_printf**, the format string parameter for the **snorkel_unmarshal** and **snorkel_marshal** functions can only contain format directives. For these functions, a format directive is a single character that acts as both a placeholder and a data type identifier. Format directives do not include a leading '%' and the relationship between the characters in the format string and the additional arguments in the argument list is one to one. There can be no spaces or other separators within the format string. The following two tables illustrate the format delimiters and their corresponding parameters for both **snorkel_unmarshal** and **snorkel_marshal.**

```
#include <snorkel.h>

int snorkel_unmarshal(snorkel_obj_t stream, char *format,…)
```

Directive	Variable input type
f	float *
d	double *

l	long *
i	int *
s	short *
y	byte_t *
z	char **

Figure 23, snorkel_unmarshal

#include <snorkel.h>

int snorkel_marshal(snorkel_obj_t stream, char *_format_,…)

Returns SNORKEL_SUCCESS on SUCCESS and SNORKEL_ERROR on failure

Directive	Variable output type
f	float
d	double
l	long
i	int
s	short
y	byte_t
z	char *

Figure 24, snorkel_marshal

Following the call to **snorkel_unmarshal** we check to see if the file exists. We return the status of the open, a minus one for failure and the file size for success, to notify the client whether or not data will follow. Finally, we stream the file using the **snorkel_file_stream** function.

#include <snorkel.h>

int snorkel_file_stream (snorkel_obj_t output_stream, char *psz_local_file_path, size_t bytes_to_send, int method)

bytes_to_send: can be zero, if zero entire file is streamed
method: SNORKEL_USE_SENDFILE or SNORKEL_FILE_SEND

Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error

Table 5, using snorkel_file_stream to send a file

Dependent on the success or failure of the protocol invocation we return either PROTOCOL_ERROR, which generates an error in the log file, or PROTOCOL SUCCESS.

To use our newly created protocol we write the client.

Example 3-1 Continued

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/stat.h>
6  #include <snorkel.h>
7  #include <errno.h>
8  #include <time.h>
9
10  #define FGET_PROTOCOL "fileget"
11
12
13  void
14  syntax (char *pszprog)
15  {
16    fprintf (stderr, "syntax error:\n");
17    fprintf (stderr,
18            "%s port@host:/remote_path local_path [-s]\n",
19            pszprog);
20    exit (1);
21  }
22
23
24  int
25  main (int argc, char *argv[])
26  {
27    FILE *fdlocal = 0;
28    char *psz_remote_host = 0;
29    char *psz_remote_path = 0;
30    char *psz_local_path = 0;
31    char *psz_port = 0;
32    int port = 0;
33    byte_t *pybuf;              /* set buffer to size of send/recieve buffers */
34    int cbbuf = 0;
35    snorkel_obj_t server = 0;
36    snorkel_obj_t logobj = 0;
37    struct stat stf;
38    int cbsource = 0;
39    int use_send_file = 0;
40    char *pszlocalarg = 0;
41    char *pszremotearg = 0;
42    int i = 0;
43    int save_file = 1;
44
45    for (i = 1; i < argc; i++)
```

```
46      {
47        if (argv[i][0] == '-')
48          {
49            switch (argv[i][1])
50              {
51               case 'b':
52                if (i + 1 >= argc)
53                  syntax (argv[0]);
54                cbbuf = atoi (argv[i + 1]);
55                i++;
56                break;
57              case 's':
58                use_send_file = 1;
59                break;
60              default:
61                syntax (argv[0]);
62                break;
63              }
64          }
65        else if (strchr (argv[i], '@') != 0 && !pszremotearg)
66          pszremotearg = argv[i];
67        else if (!pszlocalarg)
68          pszlocalarg = argv[i];
69        else
70          syntax (argv[0]);
71      }
72    if (!pszlocalarg || !pszremotearg)
73      syntax (argv[0]);
74
75    psz_port = pszremotearg;
76
77    if (!(psz_remote_host = strchr (pszremotearg, '@')) ||
78        !(psz_remote_path = strchr (pszremotearg, ':')))
79      syntax (argv[0]);
80
81    psz_local_path = pszlocalarg;
82
83    *psz_remote_host = 0;
84    *psz_remote_path = 0;
85
86    psz_remote_host++;
87    psz_remote_path++;
88
89    fprintf (stderr, "\nhost\t\t: %s\n", psz_remote_host);
90    fprintf (stderr, "port\t\t: %s\n", psz_port);
91    fprintf (stderr, "remote path\t: %s\n", psz_remote_path);
92    fprintf (stderr, "local path\t: %s\n", psz_local_path);
93    fprintf (stderr, "send file\t: %s\n",
94             (use_send_file) ? "enabled" : "disabled");
95
96
97    if (stat (psz_local_path, &stf) == 0)
98      {
99        int answ = 0;
100
101       fprintf (stderr,
```

```
102               "\nThe file %s already exists, is it ok to overwrite? <n> ",
103               psz_local_path);
104       answ = fgetc (stdin);
105
106       if (answ == '\n' || answ == EOF || answ == 'n'
107           || answ == 'N')
108         exit (0);
109     }
110
111
112   snorkel_init ();
113
114 #if defined(DEBUG) || defined(_DEBUG)
115   remove ("log.txt");
116   logobj = snorkel_obj_create (snorkel_obj_log, "log.txt");
117
118   if (logobj)
119     snorkel_debug (1);
120 #endif
121
122   fprintf (stderr,
123            "\n\nconnecting to %s@%s; please wait,...",
124            psz_port, psz_remote_host);
125
126
127   server = snorkel_obj_create (snorkel_obj_stream,      /* hostname */
128                                psz_remote_host, FGET_PROTOCOL, /* protocol */
129                                atoi (psz_port), 30);    /* connection timeout */
130
131
132   if (!server)
133     {
134       fprintf (stderr,
135                "\na connection could not be established with %s@%s\n"
136                "check log file for errors\n",
137                psz_port, psz_remote_host);
138       exit (1);
139     }
140
141   fprintf (stderr, "connected\n");
142
143
144   if (snorkel_marshal
145       (server, "zi", psz_remote_path,
146        use_send_file) != SNORKEL_SUCCESS)
147     {
148       perror ("request failed check log file\n");
149       snorkel_obj_destroy (server);
150
151       exit (1);
152     }
153
154   if (snorkel_unmarshal (server, "i", &cbsource) !=
155       SNORKEL_SUCCESS)
156     {
157       perror ("request failed, check log file\n");
```

```
158       snorkel_obj_destroy (server);
159
160       exit (1);
161     }
162   if (cbsource < 0)
163     {
164       perror
165         ("request failed, could not open remote file or file size zero\n");
166       snorkel_obj_destroy (server);
167       exit (1);
168     }
169
170   printf ("\nreceiving %d bytes; please wait,...\n",cbsource);
171   if (snorkel_file_stream (server, psz_local_path,
172                            (size_t) cbsource,
173                            SNORKEL_FILE_RECEIVE) !=
174       SNORKEL_SUCCESS)
175     {
176       perror ("request failed on receive\n");
177       snorkel_obj_destroy (server);
178       exit (1);
179     }
180
181   snorkel_obj_destroy (server);
182   exit (0);                 /* no errors */
183 }
```

In the client source, as with the server, we first initialize the runtime by calling the function **snorkel_init**. Using the hostname and port number passed in from the command line, we establish a connection with the server using the **snorkel_obj_create** function. In the same call, we request the protocol and specify the duration that we are willing to wait for a connection.

server = snorkel_obj_create (snorkel_obj_stream, psz_remote_host, "fileget", port, 10)

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create (snorkel_obj_stream, char *pszhostname, char *pszprotocol, int
port, int timeout_in_seconds)

                                                    Returns a server object on success or (snorkel_obj_t) 0 on error.
```

Figure 25, syntax for creating a connection

After establishing a connection, we send the fully qualified path to the file along with the *send file* option. Finally, we stream back the file using a variation of the **snorkel_file_stream** function shown in table 5.

#include <snorkel.h> int snorkel_file_stream (snorkel_obj_t server_object, char *psz_local_path, size_t bytes_to_receive, int *SNORKEL_FILE_RECEIVE*) Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error

Table 6, syntax for using snorkel_file_stream to receive a file

The ability to use a single port for processing both HTTP requests and other protocols simplifies the development process for complex web-applications. In addition, the API provided by Snorkel for protocol development, greatly simplifies the process of developing proprietary protocols by reducing the interactions between client and server to its simplest form – reading and writing data to a connected stream.

4 – Building Your First Bubble

A bubble is a Snorkel plug-in that encapsulates one or more functions that interact with a Snorkel-embedded server. Bubbles allow the addition of new functionality to an existing server without modifications to the server's source-code. A Bubble can contain URI overloads, MIME overloads and proprietary protocols. Concisely, bubbles are server side shared objects. Snorkel embedded servers do not load bubbles by default; it is up to a server's developer to determine whether they are supported. In this section, we will explore Bubbles and their implementation.

In example 2-3, we overloaded the MIME type for C-source files. Encapsulating the functionality into a bubble would allow us to share our functionality with other Snorkel enabled web-servers or our own embedded server solutions. In this next example, we will take the code from example 2-3 and encapsulate it into a bubble.

A bubble registers its functionality with an embedded server through the exported function **bubble_main** or through a digital tag or both.

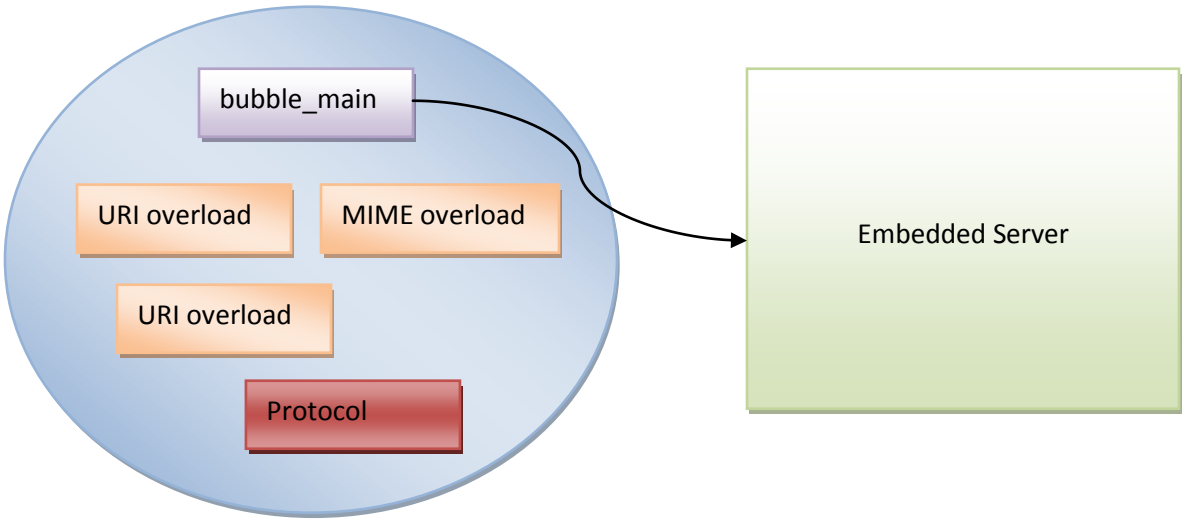


Figure 26, the function main registering functions with an embedded server

The procedure for building a bubble is simple:

- Write a shared object with exported functions of the type *snorkel_http_callback_t*, *snorkel_proto_callback_t*, and *snorkel_mime_callback_t*.
- Register the shared object as a bubble using the **snplugin** program (included in the SDK) to digitally tag the shared object for registration or register the exported functions through the exported function **bubble_main**. You can also register a bubble using a combination of both registration methods.

Example 4-1

```
1  /*
2
3  example, overloads a form URL and its corresponding POST
4
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <snorkel.h>
10
11  /*
12   content buffer containing index
13  */
14  char szIndex[] = {
15    "<html>\r\n"
16    "<body>\r\n"
17    "<a href=\"/mimeoverload.c\">example four source file</a>\r\n"
18    "</body>\r\n" "</html>\r\n"
19  };
20
21
22  char *reservedwords[] =
23    { "#include", "int", "float", "struct", "long", "true",
24    "bool", "false", "boolean", "double", "char", "unsigned",
25      "return", "#define", "break",
26    "defined", "#pragma", "union", "short", "#ifdef",
27      "#ifndef", "static", "#if", "#else",
28    "#endif", "endif", "if", "else", "switch", "else if",
29      "#elif", "const", "size_t",
30    "void", "enum", "typedef", "for", "case", "while", "do",
31      "public", "private", "signed",
32    "default", "goto", "volatile", "sizeof", "delete",
33      "class", "inline", "friend",
34    "namespace", "new", "operator", "private", "protected",
35      "public", "template",
36    "this", "throw", "try", "using", "abstract", "implements",
37      "throws",
38    "transient", "byte", "package", "synchronized", "extends",
39      "instanceof", "strictfp",
40    "catch", "final", "interface", "finally", "continue",
41      "super", "assert", "null", "NULL",
42    "exit", "extern", "public:", "private:", "protected:",
```

```
43      "private", "public", "protected",
44    "virtual", "false", "true", "register"
45  };
46
47  size_t chreservedwords =
48    sizeof (reservedwords) / sizeof (char *);
49  typedef char urlesc_t[10];
50
51  char *
52  char_esc (char c, urlesc_t escp)
53  {
54
55    escp[0] = 0;
56    if (c < 0 || c > 255)
57      return 0;
58
59    if (isalnum ((int) c) || c == '/' || c == '*' || c == '+'
60        || c == '-' || c == '#' || c == ' ' || c == '\\'
61        || c == '\n' || c == '\t' || c == '\r')
62      {
63        return 0;
64      }
65    sprintf_s (escp, sizeof (urlesc_t), "&#%d;", c);
66
67    return (char *) escp;
68  }
69
70
71
72
73  void
74  print_line (snorkel_obj_t obj,
75              int line,
76              int *pcomment,
77              char *pszin, char *pszout, size_t cb)
78  {
79    size_t i = 0;
80    char *psz = pszout;
81    int quot = 0;
82    int backslash = 0;
83    char green[] = "</font><font color=\"#008000\">";
84    char black[] = "</font><font color=\"#000000\">";
85    char blue[] = "</font><font color=\"#0000FF\">";
86    char red[] = "</font><font color=\"#C43100\">";
87    char brown[] = "</font><font color=\"#AA6600\">";
88    size_t lgreen, lblue, lblack, lred, lbrown;
89    urlesc_t escp;
90    char *pszstart = pszin;
91
92    memset (pszout, 0, cb);
93
94    if (strstr (pszin, "int") != 0)
95      {
96        int it = 0;
97        it = 1;
98      }
```

```
 99
100     snorkel_printf (obj,
101                     "<font color=\"#000000\">% 6d</font>    ",
102                     line);
103     if (*pcomment)
104       snorkel_printf (obj, "<font color=\"#008000\">");
105     else
106       snorkel_printf (obj, "<font color=\"#000000\">");
107
108     lgreen = strlen (green);
109     lblue = strlen (blue);
110     lblack = strlen (black);
111     lred = strlen (red);
112     lbrown = strlen (brown);
113
114     for (; i < cb && *pszin != 0;)
115       {
116
117         if (strncmp (pszin, "/*", 2) == 0 && !*pcomment
118             && !quot)
119           {
120             *pcomment = 1;
121             if (i + lgreen + 2 > cb)
122               return;
123             strcat (psz, green);
124             strcat (psz, "/*");
125             psz += lgreen + 2;
126             pszin += 2;
127             i += lgreen + 2;
128             backslash = 0;
129           }
130         else if (strncmp (pszin, "//", 2) == 0 && !*pcomment
131                  && !quot)
132           {
133             if (i + lgreen + 2 > cb)
134               return;
135             strcat (psz, green);
136             strcat (psz, "//");
137             psz += lgreen + 2;
138             pszin += 2;
139             i += lgreen + 2;
140             backslash = 0;
141           }
142         else if (*pszin == '\"')
143           {
144
145             if (*pcomment || backslash)
146               {
147                 if (i + 5 > cb)
148                   return;
149                 backslash = 0;
150                 strcat (psz, char_esc (*pszin++, escp));
151                 psz += 5;
152                 i += 5;
153               }
154             else if (!quot)
```

```
155             {
156               if (i + 5 + lred > cb)
157                 return;
158               strcat (psz, red);
159               strcat (psz, char_esc (*pszin++, escp));
160               quot = 1;
161               psz += 5 + lred;
162               i += 5 + lred;
163             }
164           else
165             {
166               quot = 0;
167               if (i + 5 + lblack > cb)
168                 return;
169               strcat (psz, char_esc (*pszin++, escp));
170               strcat (psz, black);
171               psz += 5 + lblack;
172               i += 5 + lblack;
173             }
174
175         }
176       else if (strncmp (pszin, "*/", 2) == 0 && *pcomment)
177         {
178           backslash = 0;
179           *pcomment = 0;
180           if (i + lblack + 2 > cb)
181             return;
182           strcat (psz, "*/");
183           strcat (psz, black);
184           pszin += 2;
185           psz += lblack + 2;
186         }
187       else
188         {
189           size_t k = chreservedwords;
190           if (!*pcomment && !quot)
191             {
192
193
194               for (k = 0; k < chreservedwords; k++)
195                 {
196                   if ((strncmp (pszin, reservedwords[k],
197                                 strlen (reservedwords[k]))
198                        == 0)
199                       && (pszin[strlen (reservedwords[k])]
200                           == 0
201                           ||
202                           pszin[strlen (reservedwords[k])]
203                           == '{'
204                           ||
205                           pszin[strlen (reservedwords[k])]
206                           == '\t'
207                           ||
208                           pszin[strlen (reservedwords[k])]
209                           == ' '
210                           ||
```

```
211                     pszin[strlen (reservedwords[k])]
212                     == '\n'
213                     ||
214                     pszin[strlen (reservedwords[k])]
215                     == '('
216                     ||
217                     pszin[strlen (reservedwords[k])]
218                     == ')'
219                     ||
220                     pszin[strlen (reservedwords[k])]
221                     == ':'
222                     ||
223                     pszin[strlen (reservedwords[k])]
224                     == ';'
225                     ||
226                     pszin[strlen (reservedwords[k])]
227                     == '<') && (pszin == pszstart
228                             || (pszin - 1)[0] ==
229                             ' '
230                             || (pszin - 1)[0] ==
231                             '\t'
232                             || (pszin - 1)[0] ==
233                             '('))
234                 break;
235               }
236
237           }
238
239         if (k < chreservedwords)
240           {
241             size_t l = strlen (reservedwords[k]);
242             if (i + l + lblue + lblack > cb)
243               return;
244             strcat (psz, blue);
245             strcat (psz, reservedwords[k]);
246             strcat (psz, black);
247
248             psz += l + lblue + lblack;
249             pszin += strlen (reservedwords[k]);
250             i += l + lblue + lblack;
251             backslash = 0;
252           }
253         else
254           {
255             if (char_esc (*pszin, escp))
256               {
257                 size_t l = strlen ((char *) escp);
258                 if (i + l > cb)
259                   return;
260                 strcat (psz, escp);
261                 psz += l;
262                 i += l;
263                 pszin++;
264               }
265             else
266               {
```

```
267                 *psz = *pszin++;
268                 if (*psz == '\\')
269                   backslash = 1;
270                 else
271                   backslash = 0;
272                 psz++;
273                 i++;
274               }
275             }
276         }
277       }
278   snorkel_printf (obj, "%s</font>\r\n", pszout);
279
280   return;
281 }
282
283 call_status_t SNORKEL_EXPORT
284 CSourceMime (snorkel_obj_t http_req,
285             snorkel_obj_t connection, char *pszurl)
286 {
287   FILE *fd = fopen (pszurl, "r");
288   char szbuffer[512];
289   char szencoded[2048];
290   int line = 0;
291   int comment = 0;
292
293   if (!fd)
294     return ERROR_STRING ("could not open source file!\r\n");
295
296   snorkel_printf (connection,
297                   "<html><body><a href=\"edit:%s\">Edit<a><hr><pre><code>\r\n");
298
299   while (!feof (fd))
300     {
301       line++;
302       if (fgets (szbuffer, sizeof (szbuffer), fd))
303         {
304           szbuffer[strlen (szbuffer) - 1] = 0;
305           print_line (connection,
306                       line, &comment, szbuffer, szencoded,
307                       sizeof (szencoded));
308         }
309     }
310   fclose (fd);
311   snorkel_printf (connection,
312                   "</code></pre></body></html>\r\n");
313   return HTTP_SUCCESS;
314 }
315
316 byte_t SNORKEL_EXPORT
317 bubble_main(snorkel_obj_t http)
318 {
319
320   /*
321    *
322    * overload a URL with a content buffer
```

```
323     *
324     */
325    if (snorkel_obj_set (http, snorkel_attrib_uri_content,        /* using a content buffer */
326                         GET,     /* method is GET */
327                         "/index.html",   /* URL */
328                         szIndex, /* content buffer */
329                         SNORKEL_STORE_AS_REF);    /* store as reference */
330
331    /*
332     *
333     * overload a mime
334     *
335     */
336    snorkel_obj_set (http, snorkel_attrib_mime, "c",      /* extension */
337                     "text/html", /* browser interpretation */
338                     encodingtype_text,   /* encoded as text */
339                     CSourceMime);        /* callback */
340
341    snorkel_obj_set (http, snorkel_attrib_mime, "cpp",    /* extension */
342                     "text/html", /* browser interpretation */
343                     encodingtype_text,   /* encoded as text */
344                     CSourceMime);        /* callback */
345
346    snorkel_obj_set (http, snorkel_attrib_mime, "hpp",    /* extension */
347                     "text/html", /* browser interpretation */
348                     encodingtype_text,   /* encoded as text */
349                     CSourceMime);        /* callback */
350
351    snorkel_obj_set (http, snorkel_attrib_mime, "h",      /* extension */
352                     "text/html", /* browser interpretation */
353                     encodingtype_text, CSourceMime);     /* callback */
354
355    snorkel_obj_set (http, snorkel_attrib_mime, "java",   /* extension */
356                     "text/html", /* browser interpretation */
357                     encodingtype_text, CSourceMime);     /* callback */
358
359    return 1;
360   }
```

We begin by adding the macro SNORKEL_EXPORT to our CSourceMime declaration to export the source interpreter. Next, we convert main to the exported function **bubble_main**.

For the embedded server we strip out all of the functionality supporting CSourceMime, leaving the functions syntax and main.

```c
/*

example, overloads a form URL and its corresponding POST

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <snorkel.h>


 void
 syntax (char *pszProg)
 {
  fprintf (stderr, "syntax error:\n");
  fprintf (stderr, "%s [-p <port>] [-l <log>]\n", pszProg);
  exit (1);
 }


 void
 main (int argc, char *argv[])
 {
  int i = 1;
  int port = 8080;
  char *pszIndex = 0;
  char *pszLog = 0;
  char szExit[10];
  snorkel_obj_t logobj = 0;
  snorkel_obj_t http = 0;

  for (; i < argc; i++)
   {
    if (argv[i][0] == '-' || argv[i][0] == '/')
     {
      char carg = argv[i][1];

      switch (carg)
       {
       case 'p':
        port = atoi (argv[i + 1]);
        i++;
        break;
       case 'l':
        pszLog = argv[i + 1];
        i++;
        break;
       default:
        syntax (argv[0]);
        break;
       }
     }
```

```
53
54      }
55
56    if (pszLog)
57      {
58        logobj = snorkel_obj_create (snorkel_obj_log, pszLog);
59        if (!logobj)
60          {
61            perror ("could not create log file\n");
62            exit (1);
63          }
64        snorkel_debug (1);
65      }
66
67
68
69    if (snorkel_init () != SNORKEL_SUCCESS)
70      {
71        perror ("could not initialize snorkel\n");
72        exit (1);
73      }
74
75    http = snorkel_obj_create (snorkel_obj_server, 2, ".");
76
77    if (!http)
78      {
79        perror ("could not create http server\n");
80        exit (1);
81      }
82
83    /*
84     *
85     * identify plugin directory
86     *
87     *
88
89     note: since path to bubble directory is NULL, the (current
90     directory)/bubbles directory is used */
91     if (snorkel_obj_set (http, snorkel_attrib_bubbles,NULL)
92         != SNORKEL_SUCCESS)
93       {
94         fprintf (stderr,
95                  "error encountered setting bubbles directory!\n");
96         exit (1);
97       }
98
99    if (snorkel_obj_set (http,
100                         snorkel_attrib_listener,
101                         port, 0) != SNORKEL_SUCCESS)
102     {
103       perror ("could not create listener\n");
104       snorkel_obj_destroy (http);
105       if (logobj)
106         snorkel_obj_destroy (logobj);
107       exit (0);
108     }
```

```
109
110
111     fprintf (stderr, "\n\n[HTTP] starting on port %d....",
112             port);
113     if (pszLog)
114       {
115         fprintf (stderr, "\n[HTTP] logging messages to %s\n",
116                 pszLog);
117       }
118     if (snorkel_obj_start (http) != SNORKEL_SUCCESS)
119       {
120         perror ("could not start server\n");
121         snorkel_obj_destroy (http);
122         if (logobj)
123           snorkel_obj_destroy (logobj);
124         exit (1);
125       }
126
127     fprintf (stderr, "\n[HTTP] started.\n\n"
128             "--hit enter to terminate--\n");
129     fgets (szExit, sizeof (szExit), stdin);
130
131     fprintf (stderr, "[HTTP] bye\n");
132     snorkel_obj_destroy (http);
133     if (logobj)
134       snorkel_obj_destroy (logobj);
135     exit (0);
136   }
```

In main, we enable bubble support by calling the function **snorkel_obj_set** (lines 91-92).

if (snorkel_obj_set (http,snorkel_attrib_bubbles,NULL) != SNORKEL_SUCCESS)

We pass the function the attribute *snorkel_attrib_bubbles* followed by a NULL pointer. The NULL pointer instructs the runtime to use the default bubble directory, the directory "*current_working_directory/*bubbles". Once enabled, the server automatically loads any file with the extension "bbl", located in the bubble directory, at start up.

```
#include <snorkel.h>

int snorkel_obj_set( snorkel_obj_t server_object, snorkel_attrib_t snorkel_attrib_bubbles, char
*fully_qualified_path_to_bubbles_directory)

        Returns SNORKEL_SUCCES if directory exists and SNORKEL_ERROR if directory does not exist or
        is not accessible
```

Figure 27, syntax for defining a bubble directory

To load bubbles manually, use the **snorkel_bubble_load** function.

```
#include <snorkel.h>


int snorkel_bubble_load (snorkel_obj_t server_object, char *psz_bubble_path)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 28, syntax for snorkel_bubble_load function

4.2 Registering Functions with an Existing Bubble

Extensions and URIs can change over time. For example, looking at our last example, we may want to add additional extensions to files that closely resemble the C or C++ file formats in the future. The **snplugin** application allows the registration of additional MIME extensions or URI associations following the build process. The application can be used to perform all registration or in addition to providing a **bubble_main** in a bubble's source.

In the next short example, we are going to associate the extension cxx with the exported CSourceMime function.

We begin by providing the shared object name to the application **snplugin**.

```
snplugin csource.bbl
```

```
J:\snorkel\release>snplugin csource.dll

snplugin - Copyright (C) 2009, Walter E. Capers
--------------------------------------------------

enter exported property/function name:<exit>  CSourceMime

select property type to register
1 - MIME
2 - URL
3 - PROTOCOL
--------------------------------
<exit>: 1
```

```
enter one or more extensions separated by
a comma extension:<return> cxx

select mime type
0 - text/html
1 - text/plain
2 - text/richtext
3 - text/rtf
4 - text/xml
5 - image/jpeg
6 - image/gif
7 - image/png
8 - image/tiff
9 - audio/mpeg
10 - audio/midi
11 - audio/x-wav
12 - video/mpeg
13 - video/quicktime
14 - application/vnd.ms-excel
15 - application/pdf
16 - application/postscript
17 - application/mspowerpoint
18 - application/zip
19 - model/vrml
20 - text/css
21 - image/psd
22 - image/vnd.microsoft.icon
23 - application/octet-stream
24 - user defined
---------------------------
:<return> 0


file: csource.bbl
property callback: view_uri
extension: cxx
type: text/html
encoding: text
---------------------------------

is the information listed above correct <y|n>? <n> y
```

```
registering property...
MIME view_uri has been successfully registered


enter exported property/function name:<exit>
bye
```

Figure 29, creating a bubble using the snplugin program… User input is in yellow.

We can review the results of our work by issueing the command:

snplugin -r csource.bbl

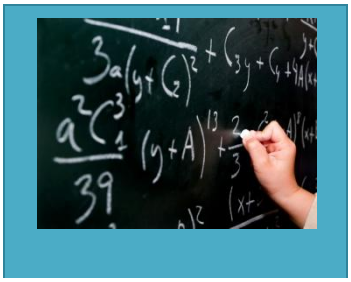
```
snplugin - Copyright (C) 2009, Walter E. Capers
--------------------------------------------------


registered properties:
================================================
property: CSourceMime
type: MIME
extension: cxx
description: text/html
encoding: text
```

Figure 30, output from snplugin -r

```
snplugin [-h]|[-r <filename>]| [-d] <filename>

h: help
r: read and display associations
d: shared object
```

Figure 31, syntax of snplugin program

5 – Advanced Topics



By now, you should have a pretty good understanding of how to write simple embedded web-servers using the Snorkel API. In the next few sections, we are going to explore advanced topics in embedded web-server development such as thread-currency, threading, performance, memory management and state-maintenance.

5.1 Mutual Exclusion

Synchronization Objects and thread currency

Mutual exclusion prevents multiple threads from accessing the same resource at the same time. As mentioned earlier, Snorkel handles requests asynchronously. When you provide callbacks for URLs, MIMEs, and proprietary protocols, the runtime invokes them in parallel. If you plan to access shared data from within these callbacks, you need to deploy mutual exclusion. The Snorkel API supports two types of mutual exclusion objects, Mutexes and Events.

5.1.1 Mutexes

A mutex is an advisory lock used collaboratively between threads where by each thread attempts to acquire the lock before accessing a shared resource. When a thread attempts to acquire a lock previously locked by another, it is blocked and placed into a wait state until the current owner of the lock releases it. In cases where multiple threads access a shared resource, mutexes used properly prevent gridlock and race conditions.

To create a mutex, you use the **snorkel_obj_create** function with the parameter *snorkel_obj_mutex*.

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create(snorkel_obj_type_t ***snorkel_obj_mutex***)

                                    Returns a pointer to a mutex object on success and NULL on failure
```

Figure 32, syntax for creating a mutex

The **snorkel_mutex_lock** function locks a mutex if is available and blocks if it is not. The **snorkel_mutex_unlock** function releases a previously acquired mutex. It is important to note, that only

the thread that acquires a mutex can release it. The runtime will produce an error for violations of this rule.

```
#include <snorkel.h>

// acquires a mutex
int snorkel_lock(snorkel_obj_t mutex_object)
// releases a mutex
int snorkel_unlock (snorkel_obj_t mutex_object)

                                    Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on failure
```

Figure 33, syntax of mutex functions

In the following code snippet, we use a mutex to prevent multiple threads from accessing the value of a global string concurrently.

```
1   #include <snorkel.h>
2
3   char *g_psz_global_string = 0;
4   snorkel_obj_t  g_mu_global_string = 0;
5
6   int
7   set_string(char *psz)
8   {
9     int ret = 0;
10    if(!psz || !!g_mu_global_string) return;
11
12    /*
13     *
14     * acquire mutex or wait until it
15     * is free
16     *
17     */
18    if(snorkel_lock(g_mu_global_string)
19       == SNORKEL_SUCCESS )
20    {
21       if(g_psz_global_string)
22        free(g_psz_global_string);
23
24        g_psz_global_string=strdup(psz);
25      ret = (g_psz_global_string)?1:0;
26
27       /*
28         * once we have changed the value
29         * release the value so other threads
```

```
30          * can view it
31          *
32          */
33        snorkel_unlock(g_mu_global_string);
34       }
35      return ret;
36  }
37
38
39  int
40  get_string(char *psz_buff,
41             int chbuff)
42  {
43       int ret = 0;
44
45       if(snorkel_lock(g_mu_global_string)
46            == SNORKEL_SUCCESS )
47          {
48           if( g_psz_global_string )
49           {
50                       ret = 1;
51                 strncat(psz_buff,g_psz_global_string, chbuff);
52            }
53           snorkel_unlock(g_mu_global_string);
54           }
55
56         return ret;
57  }
58
59
60  void
61  main(int argc,
62           char *argv[])
63  {
64         .
65         .
66         .
67    g_mu_global_string = snorkel_obj_create(snorkel_obj_mutex);
68         .
69     snorkel_obj_destroy(g_mu_global_string);
```

Keep your use of mutexes to a minimum; locking a mutex is a costly system level operation. There are many techniques for avoiding the use of mutexes documented online.

5.1.2 Events

In the Snorkel runtime, events are synchronization objects that provide thread notification. They are a cross between UNIX semaphores and Windows thread-events. To create an event object you use the **snorkel_obj_create** function with the object type **snorkel_obj_event**.

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create(snorkel_obj_type_t ***snorkel_obj_event***)

                                                    Returns an event object on success and NULL on failure
```

Figure 34, syntax for creating an event object

For setting an event, you use the **snorkel_event_set** function.

```
#include <snorkel.h>

int snorkel_event_set(snorkel_obj_t *event_object*)

                                                Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on failure
```

Figure 35, syntax for setting an event

A thread waits on an event using either the **snorkel_event_wait** function or the **snorkel_event_waittimed** function. The **snorkel_event_wait** function blocks the calling thread until an event occurs. An event occurs when another thread signals an event using the **snorkel_event_set** function. The **snorkel_event_waittimed** blocks the calling thread until an event occurs or a specified period in seconds elapses.

```
#include <snorkel.h>

int snorkel_event_wait(snorkel_obj_t *event_obect*)

        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error

int snorkel_event_waittimed(snorkel_obj_t *event_object*, int *time_in_seconds*)
```

Returns SNORKEL_ERROR if an error occurs, a value of 1 if an event occurs and 0 if one has not

Figure 36, event wait functions

5.2 Snorkel Threads

Writing your own threads should not be necessary since the Snorkel runtime handles threading for message processing internally. In addition, the runtime's use of threading has been highly optimized to perform well on multi-core systems. Writing your own threads might degrade these performance gains. Never the less we did not want to limit developers to platform specific threading APIs. Some optimization is better than no optimization at all.

5.2.1 Supported Thread Implementations

The Snorkel API supports two common implementations of threading: worker threads and run- and-forget threads. Worker threads are threads that run for the life of the process, processing events on demand. Run-and-forget threads are threads that perform a single task and exit at task completion.

5.2.2 Worker threads

Worker threads are the least costly of the two thread implementations. This is because CPU cycles are not lost to frequent thread construction and tear down.

To create a worker thread object as with other Snorkel objects, you use the **snorkel_obj_create** function.

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create(snorkel_obj_type_t snorkel_obj_worker_thread, int
thread_heap_size,  int event_queue_size)

                                        Returns a worker thread object on success and NULL on failure
```

Figure 37, syntax for creating a worker thread

The *thread_heap_size* parameter, which is in bytes, defines the threads internal heap size also known as thread local storage. Memory allocation from a threads local storage insures locality, which reduces

page faulting and boosts overall performance. We will talk more on memory management with threading later. The *event_queue_size* parameter defines the maximum number of events that a thread can receive in its queue. Worker threads use the First Come First Server model for handling events.

To place a task into a threads queue you use the function **snorkel_worker_task**.

```
#include <snorkel.h>

typedef struct int (*snorkel_task_t) (void *);

int snorkel_worker_ task( snorkel_obj_t worker_thread_obj, snorkel_task_t task, void *data, size_t cb)
```

Returns SNORKEL_SUCCESS on success, SNORKEL_ERROR on error, and SNORKEL_BUSY if the thread is too busy to receive additional tasks
Note: data can be null. If data is not NULL and cb=0 the data is passed by reference to the task otherwise it is passed by value.

Figure 38, syntax for tasking a worker thread

The task callback function provided to the **snorkel_worker_task** should always return a value of one on success and zero on failure. In the example below, we create a worker thread and task it with printing the phrase “hello from a task”.

```
1   #include <stdio.h>
2   #include <snorkel.h>
3
4   #define QUEUE_SIZE 5
5
6   int
7   task(void *lpv)
8   {
9         char *psz = (char *)lpv;
10
11        printf("%s", psz);
12
13        return 1;
14  }
15
16  void
17  main(int argc,
18           char *argv[])
19  {
20        snorkel_obj_t worker=0;
21        int i = 0;
22        char szmessage[]="hello, from a task\n";
```

```
23
24      snorkel_init();
25
26      worker = snorkel_obj_create(snorkel_obj_worker,
27                                              256000,/* 1 meg heap */
28                                              QUEUE_SIZE);   /* queue size */
29
30      if( !worker )
31      {
32              fprintf(stderr,"could not create worker thread!\n");
33              exit(1);
34      }
35
36      for( i=0; i< QUEUE_SIZE; i++ )
37
38              snorkel_worker_task(worker,szmessage,strlen(szmessage));
39
40      snorkel_thread_sleep(500);
41
42      snorkel_obj_destroy(worker);
43
44      exit(0);
45  }
46
47
48
```

On line 26, we create our worker thread specifying a heap size of 256k and a queue size of five. 256k is quite large for this example, but for normal implementations, we recommend a heap size of at least 256k. After tasking the thread, we call the thread safe function **snorkel_thread_sleep** to give the worker thread a chance to complete its tasks.

```
#include <snorkel.h>

void snorkel_thread_sleep(size_t milliseconds)
```

Figure 39, snorkel_thread_sleep function

As with other Snorkel objects we call **snorkel_obj_destroy** to terminate the worker-thread and free the resources allocated to it when we exit.

Even though the example above may seem trivial, the amount of effort required to develop a platform independent queue based worker thread using system level APIs would require quite a bit more effort.

5.2.3 Run-and-forget Threads

A Run-and-forget thread is a thread that performs a single operation and exits. It is the simplest thread type to implement but suffers in performance when used frequently due to thread construction and tear down.

As with other Snorkel objects, you use the **snorkel_obj_create** function to create the thread object.

```
#include <snorkel.h>

snorkel_obj_t snorkel_obj_create(snokel_obj_type_t ***snorkel_obj_thread***, snorkel_task_t *function,* void *argument)

                                                    Returns a thread object on success and NULL on failure
```

Figure 40, using snorkel_obj_create to create a thread

The functions **snorkel_thread_wait** and **snorkel_thread_waittimed** allow you to wait on a thread for its completion.

```
#include <snorkel.h>

int snorkel_thread_wait(snorkel_obj_t thread_object)

int snorkel_thread_waittimed(snorkel_obj_t thread_object, size_t milliseconds)

                    Both functions return SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 41, the snorkel_thread_wait functions

You use the **snorkel_obj_destroy** function to terminate a thread created by the **snorkel_obj_create** function.

5.3 Memory Management

Roughly, 90% of all memory requested by server processes are of varying sizes and are short-term. This behavior over time leads to excessive external heap fragmentation. This is a weakness of some allocators. This form of fragmentation occurs when a process allocates and deallocates regions of memory that are of differing sizes. The allocator responds by leaving the allocated and deallocated regions interspersed. As memory with in the process heap becomes unavailable, the operating system grows the heap to continue satisfying requests. Overtime the result can be that although free storage is available, the fragmented chunks are too small to satisfy a request. To solve this problem the Snorkel runtime provides two optimized functions, **snorkel_mem_alloc** and **snorkel_mem_free** for the allocation and deallocation of short-term memory. These functions are useful when allocating resources from within a Snorkel callback.

Snorkel_mem_alloc will always attempt to allocate memory from a calling thread's local/heap storage. If memory in the thread heap is not available, it allocates it from the process heap. Do not attempt to reference memory allocated by **snorkel_mem_alloc** globally, threads periodically clear or free this memory between connections to conserve resources.

For short-term memory allocations, use these functions instead of the system level memory API.

```
#include <snorkel.h>

void * snorkel_mem_alloc(size_t cb)

void snorkel_mem_free(void *lpv)
```

Figure 42, syntax for the Snorkel memory management APIs

These functions attempt to insure locality for memory access, providing improved performance over conventional memory allocation schemas.

5.4 Server Optimization

By default, Snorkel's performance settings are set based on a balance between performance and memory consumption. You should consider changing or exposing these settings for applications designed to run on high-end systems and networks with large bandwidths. In this section, we will take a brief look at ways to optimize Snorkel performance.

5.4.1 Handler Thread Heap Storage

By default, handler threads have a fixed heap size of 256K. In cases were large amounts of memory is allocated and deallocated this might not be enough. When a thread's heap storage is exhausted, the runtime does not perform garbage collection or attempt to increase it. Instead, it resorts to using the system memory allocation APIs until large enough chunks of the thread-heap become available to handle new memory requests. Accessing memory outside of a thread's heap voids locality and can degrade overall performance. To prevent this from occurring you can override the default heapsize by requesting much larger heaps using the **snorkel_obj_set** function with the *snorkel_attrib_threadheap_size*.

```
#include <snorkel.h>

int snorkel_obj_set(snorkel_obj_t system_object,snorkel_attrib_t snorkel_attrib_threadheap_size, int
size_in_bytes)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 43, syntax for increasing thread heap size

The use of this function requires a pointer to the system object. To get the system object use the function **snorkel_get_sys**.

```
#include <snorkel.h>

snorkel_obj_t snorkel_get_sys()

                                        Returns a pointer to the Snorkel system object
```

Figure 44, use the system object to set global attributes

Thread heap size must be set prior to calling **snorkel_obj_start**. The example below sets the thread heap size to one megabyte.

snorkel_obj_set (snorkel_get_sys, snorkel_attrib_threadheap_size, 1024000)

One way to take advantage of larger thread heaps it to use the **SNORKEL_STORE_AS_DUP** option when overloading URIs with content buffers. Ideally, you should cache images that appear as part of web-content when ever possible.

5.4.2 Using Larger TCP Windows

The send/recieve buffer size also known as the TCP window size, defines the maximum amount of data sent per transmission between a client and server. Eventhough we recommend at least 66560 we use

the system default. For networks with high bandwidth, this can greatly improve server throughput. This is because no matter how fast your CPU is, the socket layer, which involves the transport of data across a network, is a bottle kneck. By sending data in larger chunks, the amount of calls to the socket layer becomes less frequent, reducing the bottle kneck affect. To set the send/receive buffer size use the **snorkel_obj_set** function with the *snorkel_attrib_tcpbuffsize* attribute.

```
#include <snorkel.h>

int snorkel_obj_set(snorkel_obj_t server_object, snorkel_attrib_t snorkel_attrib_tcpbuffsize, int
buffer_size_in_bytes)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 45, syntax for setting the send/receive buffer size

6 – Miscellaneous Topics

There are a few topics that we did not feel belonged in the other sections such as Cookies and user authentication.

6.1 Cookies

Cookies can be set in URI overload callbacks by calling the **snorkel_obj_set** function with the *snorkel_attrib_cookie* attribute.

```
#include <snorkel.h>

int snorkel_obj_set( snorkel_obj_t http_request_object, snorkel_attrib_t snorkel_attrib_cookie, char
*cookie_name, char *cookie_value, char *domain, int secure, int duration)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 46, syntax for setting a cookie

The parameters *domain* and *secure* restrict the deployment of cookies. If the *domain* parameter is not NULL a cookie will only be set on systems with in the same domain. The *secure* parameter, if one, will only allow a cookie to be set over a HTTPS connection.

The *duration* parameter specifies life expectancy of a cookie in seconds. If this parameter is zero the cookie never expires. For example,

snorkel_obj_set (http_req,snorkel_attrib_cookie,"order","1234",0,0,0)

stores the cookie *order* with a value of "1234".

To retrieve the value of a cookie use the **snorkel_obj_get** function with the *snorkel_attrib_cookie* attribute.

```
#include <snorkel.h>

int snorkel_obj_get (snorkel_obj_t http_request_object,snorkel_attrib_t, char * cookie, char
*cookie_value, int size_of_cookie_value)

                                        Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on error
```

Figure 47, syntax for retrieving a cookie

For example,

snorkel_obj_get (http_req, "order",sz_order_no,cb_order_no)

retrieves the value of the cookie "order", placing it into the string *sz_order_no*.

6.2 User Authentication

User authentication is incomplete in this release, the Snorkel runtime only supports Basic authentication and passwords in password files are not encrypted. Once enabled, users are prompted with a password dialog when entering restricted sites.

6.2.1 Restricting User Access

By User

- Create a file named *.htaccess* similar to the one below.

```
AuthUserFile c:\path\.htpasswd
AuthGroupFile \dev\null
require valid-user
```

Figure 48, .htaccess file for user authentication

- Copy the *.htaccess* file to directories that you wish to restrict.
- Create the password file referenced by the group access file similar to below. Each record has the form username password.

```
John:biggipper
Jack:123456
Mark:ab12c67d
```

Figure 49, .htpasswd file

By Group

- Create a file named *.htaccess* similar to the one below.

```
AuthUserFile c:\path\.htpasswd
AuthGroupFile c:\path\.htgroup
require group developers
require group users
```

Figure 50, .htaccess file for user authentication

- Copy the *.htaccess* file to directories that you wish to restrict.

- Create the password file referenced by the group access file similar to below. Each record has the form username password.

```
John:biggipper
Jack:123456
Mark:ab12c67d
```

Figure 51, .htpasswd file

- Create a group file similar to below.

developers:John Jack

users:Mark

Allow by IP address

- Create a file named *.htaccess* similar to the one below.

```
allow from 192.12.4.8
allow from 192.12.4.12
allow from 193.13.3
```

Figure 52, .htaccess file for user authentication

In the example above the IP-addresses 192.12.4.8 and 192.12.4.12 in addition to connections from the domain 193.13.3 allowed.

- Copy the *.htaccess* file to directories that you wish to restrict.

Deny by IP address

- Create a file named *.htaccess* similar to the one below.

```
deny from 192.12.4.8
deny from 192.12.4.12
deny  from 193.13.3
```

Figure 53, .htaccess file for user authentication

The example above blocks IP-addresses 192.12.4.8 and 192.12.4.12 in addition to connections from the domain 193.13.3 allowed.

- Copy the *.htaccess* file to directories that you wish to restrict.

You can also mix the various methods of HTTP security.

6.3 Queries and HTTP Header Values

URIs proceded by a "?" and query string define a query. To acquire a query string from a URI the **snorkel_obj_get** function with the *snorkel_attrib_header* parameter. For example,

```
snorkel_obj_get (http, snorkel_attrib_header, "QUERY", szquery, (int) sizeof (szquery))
```

sets the value of the query string in szquery. The call above returns SNORKEL_SUCCESS if a query string exists and SNORKEL_ERROR if it does not. You can also acquire other header values using the *snorkel_attrib_header* parameter for **snorkel_obj_get**.

Roadmap

The Snorkel project is an ongoing project with new enhancements and platforms added regularly. Over the next year there are several enhancements planed for the next release of the Snorkel runtime, version 1.0. These include:

- Improved error messaging
  - User definable multi-lingual error messages
  - More granual error messaging
- SSL support for non-HTTP protocols
- AES-encryption for non-HTTP protocols when SSL is not available
- Windows Mobil 7 support
- Android support (tentative)
- MAC OS (dependent on platform availability)
- IPhone support (tentative)
- Linux specific optimizations to further extend multi-core support
- Expose API function to disable thread governer
- Expose currently untested APIs

Other projects based on the Snorkel core

Development work on Aqua, a user interface libarary based on Windows dialogs for browser interface development, begins in early to late summer. Beginning in the spring and slated for release in mid to late summer, the Kwatee project, a C++ implementation of snorkel API.

snorkel_init

Initializes the Snorkel API

int snorkel ()

Parameters
none

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

Remarks

Call this routine prior to calling any Snorkel API functions except for creating a log.

snorkel_obj_create

Creates Snorkel objects

snorkel_obj_t snorkel_obj_create (**snorkel_obj_type_t** type,…)

Parameters

type: identifies the object type and syntax

snorkel_obj_server	Allocate storage for a HTTP server object. **Parameters** ***int*** – the number of connection handler threads ***char **** – null or a pointer to a null terminated character string containing the directory which contains the index.html file.
snorkel_obj_protocol	Define a protocol (HTTP-overload) **Parameters** ***char **** – string identifying protocol name ***call_status_t (*callback) (snorkel_obj_t)*** – pointer to a protocol callback function.
snorkel_obj_event	Allocate storage for an event object. **Parameters** ***none***
snorkel_obj_thread	Create a Run-and-Forget thread **Parameters** ***snorkel_thread_func_t*** – pointer to a task callback function ***void **** **-** argument for callback function or NULL

snorkel_obj_worker_thread	Create a worker thread **Parameters** ***int*** – thread heap size in bytes ***int*** – event queue size
snorkel_obj_stream	Create a connection with a sever using the specified protocol. **Parameters** ***char *-*** pointer to null terminated string containing the hostname or address of the server system. ***char **** - pointer to null terminated stringcontaining the protocol name descriptor. ***int*** – server port number. ***int*** – timeout in seconds to wait for a connection.
snorkel_obj_log	Create a runtime log file **Parameters** ***char **** – pointer to a null terminated string containing the fully qualified path to a log file.

Return

For all object types excluding *snorkel_obj_protocol*, this function returns a pointer to the allocated object on success, NULL on failure. A value of (snorkel_obj_t *)1 is returned on success and (snorkel_obj_t *)0 on failure for protocol objects.

ERRNO VALUES:

EINVAL – missing or incorrect arguments passed

ENOMEM -- there was not enough resources to allocate memory

Remarks

snorkel_get_sys

Get runtime system object.

snorkel_obj_tsnorkel_get_sys()

Parameters
none

Return

Returns NULL on failure.

Remarks

snorkel_mutex_lock

Lock a mutual exclusion object.

int snorkel_mutex_lock(**snorkel_obj_t** mutex)

Parameters

mutex: mutex object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno*. If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

Blocking occurs if another thread holds the lock.

snorkel_mutex_unlock

Release a mutex lock.

int snorkel_mutex_unlock(**snorkel_obj_t** mutex)

Parameters

mutex: mutex object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno*. If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter, invalid parameter type, or the calling thread does not own the lock.

Remarks

Only the thread that locks a mutex can release it.

snorkel_marshal/snorkel_unmarshal

Write and read binary data from a stream.

int snorkel_marshal(snorkel_obj_t stream, **char** ***format,…**)

int snorkel_unmarshal(snorkel_obj_t stream, **char** ***format,…**)

stream – pointer to snorkel stream object
format – a list containing one or more format directives which have associated arguments in the argument list.

snorkel_marshal directives

Directive	Variable output type
f	float
d	double
l	long
i	int
s	short
y	byte_t
z	char *

snorkel_unmarshal directives

Directive	Variable output type
f	float *
d	double *
l	long *
i	int *
s	short *
y	byte_t *
z	char **

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

Remarks

snorkel_event_set

Signal an event object.

int snorkel_event_set (**snorkel_obj_t** event)

Parameters

event: event object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

snorkel_event_wait

Wait on an event object signal.

int snorkel_event_wait (**snorkel_obj_t** event)

Parameters

event: event object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

Blocks until the provided event is signaled.

snorkel_event_waittimed

Wait on an event signal for the specified period in milliseconds.

int snorkel_event_waittimed(**snorkel_obj_t** event, **long** milliseconds)

Parameters

event: event object created using the ***snorkel_obj_create*** function.
milliseconds: the maximum amount of time to wait on an event.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

Blocking occurs if another thread holds the lock.

snorkel_obj_destroy

Releases resources allocated to a Snorkel object.

int snorkel_obj_destroy (**snorkel_obj_t** obj)

Parameters

obj: object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

In the case of an HTTP server object, terminates all server threads prior to releasing resources. This function should not be called against an HTTP connection except by the API. When applied against a connection, the call tears down the connection.

snorkel_obj_start

Start a server thread

int snorkel_obj_start (**snorkel_obj_t** server_object)

Parameters

server_object: http server object created using the ***snorkel_obj_create*** function.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

snorkel_printf, snorkel_put

Write data to a Snorkel object

int snorkel_printf(**snorkel_obj_t** obj, **char** *pszformat, …)
int snorkel_uprintf(**snorkel_obj_t** obj, **char** *pszformat, …) UUEncode version of snorkel_printf
int snorkel_put(**snorkel_obj_t** obj, **void** *data, **size_t** cb)

Parameters

obj: must be a object that supports output such as a connection or buffer object.
pszformat: null terminated string containing format descriptors, see C's printf.
data: pointer to data to be sent
cb: size of data in bytes

snorkel_uprintf converts incoming data to UUEncoded form.

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

snorkel_mem_alloc

Allocate storage from thread local storage.

void * snorkel_mem_alloc (size_t cb)

Parameters

cb: size in bytes of requested storage.

Return

This function returns a pointer to the newly allocated storage or null on failure.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type
ENOMEM: insufficient memory

Remarks

Only use this function for temporary storage… Thread local storage is periodically cleared or freed between connections and is not permanent. To free memory allocated by snorkel_mem_alloc use snorkel_mem_free.

snorkel_mem_free

Free memory allocated by snorkel_mem_alloc

void snorkel_mem_free(**void *** lpv)

Parameters

lpv: pointer to previously allocated memory

Remarks

snorkel_obj_set

Set object attributes

snorkel_obj_t snorkel_obj_set(**snorkel_obj_t** obj, **snorkel_attrib_t** attrib, …)

Parameters

When obj is of type server

attribute:

snorkel_attrib_uri	Overload a URI **Parameters** ***int*** –the value GET or POST ***char **** – URI ***encodingtype_t –*** encodingtype_text or encodingtype_binary ***Function pointer*** – *callstatus_t (*snorkel_uri_callback_t)(snorkel_obj_t* http_object*, snorkel_obj_t* output_stream*)* or the reserved word IGNORE_MIME
snorkel_attrib_uri_content	Overload a URI using a content buffer **Parameters** ***int –*** must be GET ***char * --*** URI ***char * --*** pointer to null terminated content ***int --*** flag SNORKEL_STORE_AS_DUP: duplicates content within each threads local stoage. SNORKEL_STORE_AS_REF: stores a pointer to content within each threads local storage.
snorkel_attrib_mime	Allocate storage for an event object.

	Parameters
	char** * -- extension* ***char** *-- mime type* ***encoding_type_t – *identifies the encoding type: encodingtype_binary or encodingtype_text* ***function pointer –*** *call_status_t (*snorkel_mime_callback_t)(snorkel_obj_t http_object, snorkel_obj_t output_stream, char *URL)*
snorkel_attrib_bubbles	Enables bubble recognition
	Parameters
	***char** * -- bubble directory or NULL. If NULL provided, defaults to default bubble directory.*
snorkel_attrib_listener	Creates a port listener in the server object
	Parameters
	int – port ***int*** – SSL flag. If one enables SSL over HTTP if zero disables SSL over HTTP.
snorkel_attrib_uri_cache	Overloads a URL with a content buffer derived from a file
	Parameters
	int – request method. Must be get **GET** for this version. ***char** * - null terminated string containing URI* ***char** * - null terminated string containing path to file relative to the index directory.* ***int*** – duplication option. SNORKEL_STORE_AS_DUP: duplicates content within each threads local stoage. SNORKEL_STORE_AS_REF: stores a pointer to content within each threads local storage.
snorkel_attrib_tcpbuffersize	Sets the send/receive buffer size for the socket layer
	Parameters
	int -- size in bytes.

snorkel_attrib_index_file	Sets the name of the index file which defaults to index.html **Parameters** ***char** **** -- pointer to null terminated string containing the name of the index file.
snorkel_attrib_show_dir	Enables/disables the display of directories as HTML pages when an index is not present or the user specifies a URI that maps to a directory. **Parameters** ***int*** **–** a value of one enables the display of directories and zero disables it.

When obj is of type http request

attribute:

snorkel_attrib_cookie	Sets the value of a cookie **Parameters** ***char** **** – null terminated string containing the name of the cookie to set. ***char** **** – value of cookie ***char** **** – NULL or a null terminated string containing the cookies domain restriction. ***int*** – if one the cookie is only sent over a secure/https connection. ***int*** **–** the duration of the cookie in seconds
snorkel_attrib_header	Set a HTTP header value **Parameters** ***char** **** – null terminated string containing the name of the header entry. ***char** **** – value of header entry

snorkel_attrib_pagespan	Set the life span of returned content. **Parameters** ***int** ** – life expectancy in seconds

When obj is of type system

attribute:

snorkel_attrib_tcpbuffsize	Sets the TCP/IP window size. **Parameters** ***int*** – size in bytes.
snorkel_attrib_threadheapsize	Sets the default local storage/heap size for handler-threads. **Parameters** ***int*** – size in bytes.

Return

Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on fault

ERRNO VALUES:

EINVAL – missing or incorrect arguments passed
ENOMEM -- there was not enough resources to allocate memory

snorkel_thread_sleep

Pause a thread or process

void snorkel_thread_sleep(**long** milliseconds)

Parameters

milliseconds: time to sleep in milliseconds

snorkel_obj_get

Set object attributes

snorkel_obj_t snorkel_obj_get(**snorkel_obj_t** obj, **snorkel_attrib_t** attrib, …)

Parameters

When obj is of type http request

attribute:

snorkel_attrib_post	Retrieve the value of a post variable. **Parameters** ***char** ** -- pointer to a null terminated string containing the variable name. ***char** ** – pointer to a buffer to receive the variable value ***int*** *– the size of the buffer in bytes.*
snorkel_attrib_header	Retrieve the value of a HTTP header variable. **Parameters** ***char** ** -- pointer to a null terminated string containing the variable name. ***char** ** – pointer to a buffer to receive the variable value ***int*** *– the size of the buffer in bytes.*

snorkel_attrib_uri	Retrieve the URI of the requested page. **Parameters** ***char** **** – pointer to a buffer to receive the URI ***int*** *– the size of the buffer in bytes.*
snorkel_attrib_local_url	Retrieve the URL of the requested page. **Parameters** ***char** **** – pointer to a buffer to receive the URL ***int*** *– the size of the buffer in bytes.*
snorkel_attrib_cookie	Retrieve the value of a cookie. **Parameters** ***char** **** -- pointer to a null terminated string containing the cookie name. ***char** **** – pointer to a buffer to receive the cookie value ***int*** *– the size of the buffer in bytes.*

Return

Returns SNORKEL_SUCCESS on success and SNORKEL_ERROR on fault

ERRNO VALUES:

EINVAL – missing or incorrect arguments passed
ENOMEM -- there was not enough resources to allocate memory

snorkel_worker_task

Assign a task to a worker thread

int snorkel_worker_task(**snorkel_obj_t** worker_thread, **snorkel_task_t** task, **void** *data, **size_t** cd)

Parameters

worker_thread: worker thread object created using the ***snorkel_obj_create*** function.
task: pointer to a task callback
data: pointer to data to be passed to task or NULL
cb: the size of the data pointed to by data or 0 if data is NULL

Return

Returns SNORKEL_ERROR on failure and sets *errno.* If no errors occur, SNORKEL_SUCCESS returned.

ERRNO VALUES

EINVAL: missing parameter or invalid parameter type.

Remarks

Blocking occurs if another thread holds the lock.