HTTP / HTTPS Library for Palm OS

Version 1.3.2

PDADevelopers.com

HTTP/HTTPS Library for Palm OS
Table of Contents

Introduction

This library is an implementation of a subset of W3C specs for HTTP protocol. It's packed as a library file with C++ headers, so you can include it in your own programs simply linking it into them. It runs on ANY Palm OS device running Palm OS version 3.0 and higher (not only on Palm VII).

Features

- Provides HTTP/HTTPS client protocol on top of the standard Palm socket library.
- Easy to use, extensible C++ classes: functions as simple as connect(), execute request(), get response(), disconnect(), etc.
- Provides control for getting HTTP headers and cookies management.

General Notes

You can use the library for requesting information from existing web servers, and then parse the results. You can even use it for integrate your programs with existing Web Services.

An HTTP server (usually called Web Server) hosts resources (i.e. files, like HTML files, images, etc.) that can be accessed by an URL (Uniform Resource Locator). The HTTP library can be used to convert your program into an HTTP client that can retrieve those resources from HTTP servers into a local buffer. Then your program can manage the resources contained in your buffer to do whatever is necessary to be done (save to a file, parse a text, show information to the user, etc.)

Tutorial

Step By Step example

Step 1: Create an HTTP Connection Object

The HTTPConnection object holds de socket connection with your web server. Depending on the headers setting and/or the web server behavior, the socket connection is maintained or released after retrieving the resource from the server.

```
HTTPConnection *cnn = new HTTPConnection(40960,10240);
```

The value 40960 indicates the buffer size in bytes, this is the max resource/file size can be retrieved in a single request. This value can range between 1 byte to slightly less than 64K.
The value 10240 indicates the output buffer size in bytes.

Step 2: Create an HTTP Request

After you established the connection, you can create an HTTP Request based on that connection. The HTTPRequest object will carry the full URL for getting the resource, request headers, cookies, etc.

```
HTTPRequest *req = new HTTPRequest(cnn);
```

Step 3: Fill the HTTPRequest object and executing the Request

There are two ways for filling the values in an HTTPRequest object. The first approach, a simpler one, enables you to leave all properties to the default value and execute the request in the same step. The following code illustrates this situation:

```
responseCode = req->execute(httpGet, “www.palmone.com”, NULL);
```

In this single step, the routine:

1) Assumes the domain and resource name (path + file name) by parsing the URL.
2) Specifies GET as the verb to be used.
3) Effectively executes the request to the server.

The other way to fill the HTTPRequest object and execute the request is to manually fill the verb, domain, path and headers. This gives you a better control on how the request will be executed. This is definitively the way to go when you are passing parameters in the URL, like when invoking FORM result, calling WebService, etc. The following code illustrates this:

```
// Set the Verb
req->verb = httpGet;

// Set the domain and path of the resource
req->domain.set("www.palmone.com");
req->path.set("/pda/getImage.asp?ID=101");

// Modify the default headers (optional)
req->getHeaders()->remove("Connection");
req->getHeaders()->remove("Accept");
req->getHeaders()->remove("User-Agent");
```

```
// Execute the Request
responseCode = req->execute();
```

Step 4: Process the HTTP Response

Once the HTTPRequest has been executed, you will receive the reply from the web server or an error condition. The error can be checked by calling `req->getLastError()`. A zero value indicates no error.
Your program can retrieve the Response information by accessing the HTTPResponse object contained in the executes HTTPRequest:

```
// Get resource content
content = req->getResponse()->getContent();

//Get content-length header (Not available if transfer-coding: chunked)
contentL = req->getResponse()->getContentLength();
```

Please note that the library can calculate the Content Length value in two different ways:

1) If the Response includes a *Content-Length* header, then this value is returned.
2) If no *Content-Length* header is specified in the Response, then the value is calculated from the size of the Content itself.

Step 5: Release the objects

Finally, don't forget to release the object. You can release the HTTPRequest and maintain the HTTPConnection, if the server keeps it alive, or release both:

```
delete req;
delete cnn;
```

Establishing a Secure Connection

HTTP works fine in most uses, but there is a drawback: data is sent in clear text. Using any sniffer available in the market you can read the information contained in the request and response message flow. In order to improve security, HTTPS protocol was developed.

HTTPS protocol is a version of HTTP that runs on top of Secure Socket Layer (SSL). SSL was implemented in the core of Palm OS after OS version 5.0. Using this layer the HTTP connection can be established over a secure channel. For further information on SSL, check RFC 2246.

The steps to use HTTPS instead of HTTP are EXTREMELLY simple. You just need to use the *HTTPSConnection* (with 'S') object instead of *HTTPConnection*. That is, assuming the example shown in the previous chapter, replace step 1 with this:

```
HTTPSConnection *cnn = new HTTPSConnection(40960, 10240);
```

All other management of the SSL connection, in addition to certificates management et al, are automatically handled by Palm operating system.
Please note that HTTPS can only be used with Palm OS 5.0 and above.

Frequently Asked Questions (FAQ)

Default Headers

In order to ease the creation of the HTTPRequest we include five default headers that are used in mostly all typical requests. Here is the list of default headers:

```
headers->setValue("Accept", "text/html, text/xml, */*");
headers->setValue("User-Agent", USER_AGENT);
headers->setValue("Connection","Keep-Alive");
headers->setValue("Content-Type", "text/html; charset=ISO-8859-4");

headers->setValue("Content-Length", IToA(StrLen(m_request->data)));
```

In case your application does not need these headers, you can remove or replace them after creating the object and before executing the request.

Maintaining open the connection with the web server

Closing the socket connection after a response to a GET command is sent is the usual behavior on most web servers. Some web servers admit a special header to keep the connection open, improving performance. In order to maintain the connection opened with your server, you must set the HTTPRequest header *"Connection=Keep-Alive"*:

```
req->getHeaders()->setValue("Connection", " Keep-alive");
```

Automatic Redirection

Automatic Redirection is a special feature present in most Web Browsers that allows redirecting one resource to another. It is managed with a special Return Code = 302. As this is application-specific behavior (not all applications may like to get redirected automatically) it must be handled by the application itself: the library does not handle it automatically.

Implemented Verbs

GET, POST, PUT, DELETE, HEADER.

Simulating Specific Browsers Headers

You can simulate a specific browser by adding the "User-Agent" header to your request. Here's an example:

```
req->getHeaders()->setValue("User-Agent",
    " Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)");
```

Search Google for other browser's strings

Managing timeouts

When handling large responses on slow web servers or under heavy traffic, you may need to increase the receive timeout. You can do that by calling the *setTimeOut* function:

```
HTTPConnection::setTimeOut(int timeOut)
```

timeOut = Maximum timeout in system ticks; -1 means wait forever.

Library Size

The library size is about 100Kb.

Reference

HTTPConnection Object

HTTPConnection::HTTPConnection(
unsigned short inBufferSize = INPUT_BUFFER_SIZE,
unsigned short outBufferSize = OUTPUT_BUFFER_SIZE);
Creates a new HTTPConnection object. Prepares buffer and internal HTTPRequest / HTTPResponse objects. Enables connection for an upcoming request.

HTTPRequest* HTTPConnection::getRequest();
void HTTPConnection::setRequest(HTTPRequest* req);
Gets/sets the HTTPRequest object. Typically you only need to retrieve the internal HTTPRequest object, previously from executing the request.

void HTTPConnection::HTTPResponse* getResponse();
void HTTPConnection::setResponse(HTTPResponse *resp);
Gets/sets the HTTPResponse object. Typically you only need to retrieve the internal HTTPResponse object, once the request has been executed.

short HTTPConnection::getTimeOut();
void HTTPConnection::setTimeOut(unsigned short timeOut);
Gets/sets the current request retrieval timeout, expressed in system ticks. This timeout applies to the initial connection and to every request to the server. If the HTTP server does not send any data to the client for more than timeOut ticks, then a timeout occurs.

HTTPCookies* HTTPConnection::getCookies();
Retrieves the cookies collection. This collection is maintained during the connection lifetime.

bool HTTPConnection::parseURL(char* url,
CMKString* domainName,
short *port,
CMKString* path,
CMKString* fileName,
CMKString* queryString);
Prepares the internal HTTPRequest object with this information.

short HTTPConnection::executeRequest(CMKString* pLastStringSend = NULL);
Executes the current request. If pLastStringSend is present it returns the last string send to server.

bool HTTPConnection::connected();
Indicates if the connection established with the server is alive.

bool HTTPConnection::connect();
bool HTTPConnection::connect(char *domain, unsigned short port);

void HTTPConnection::disconnect();
Manually connects to/disconnects from the web server.

HTTPRequest Object

HTTPRequest::HTTPRequest();
HTTPRequest::HTTPRequest(HTTPConnection *connection);
Creates a new object, with or without HTTPConnection associated.

HTTPRequestType HTTPRequest::verb;
CMKString HTTPRequest::domain;
short HTTPRequest::port;
CMKString HTTPRequest::path;
CMKString HTTPRequest::fileName;
CMKString HTTPRequest::queryString;
CMKString HTTPRequest::data;
Assessors to verb, domain, port path, filename, query string and additional data.

HTTPHeaders * HTTPRequest::getHeaders();
Retrieve Headers collection.

short HTTPRequest::execute();
Execute a previously defined HTTPRequest. Must have an associated HTTPConnection object.

short HTTPRequest::execute(HTTPRequestType verb,
char* url,
char *data = NULL);
Creates a new HTTPRequest, doing a simple parser on the URL, and executes it. It must have an associated HTTPConnection object.

void HTTPRequest::setAuthorization(char* user, char* pass);
Sets user/password for the request, if requested by the server.

void HTTPRequest::GetLastStringSendToServer(CMKString& rMKString)
Returns the last string sent to the server.

HTTPResponse Object

HTTPResponse::HTTPResponse(HTTPRequest* request);
Creates a new response object, initializing the pairing request.

unsigned short HTTPResponse::responseCode;
Returns the return code from the previous request.

unsigned short HTTPResponse::getContentLength();
Gets the content length received from the previous request. Content Length value can be calculated in two different ways:

1. If the Response includes a Content-Length header, then this value is returned.
2. If no Content-Length header is specified in the Response, then the value is calculated from the size of the Content itself.

char* HTTPResponse::getContent();
Returns a pointer to the received data from the previous request execution.

void HTTPResponse::setResponseBuffer(char* buffer);
Sets the response buffer pointer.

void HTTPResponse::clearResponseObject();
Clears the object.

TABLES

Response Codes

RESPONSE CODE	Value
rcContinue	100
rcOK	200
rcCreated	201
rcAccepted	202
rcNoContent	204
rcMultiple	300
rcMovedPermanently	301
rcMovedTemporarily	302
rcNotModified	304
rcBadRequest	400
rcUnauthorized	401
rcForbidden	403
rcNotFound	404
rcInternalServerError	500
rcNotImplemented	501
rcBadGateway	502
rcServiceUnavailable	503

VERBS

```
enum HTTPRequestType {
        httpGet,
        httpPost,
        httpPut,
        httpDelete,
        httpHeader
};
```

ERROR Codes

Error Code	Value
errSocketNotSet	1000
errSocketAlreadyConnected	1001
errSocketNoConnected	1002
errSocketError	1003

Purchasing the source code

If you are interested in purchasing the library source code, please contact **sales@pdadevelopers.com** for more information.

Revision History

Version 1.0
Initial Version.

Version 1.0.4
Fixed: duplicated error code in HTTPConnection.h. Other minor issues.

Versión 1.0.5
Improved timeout support. Other code enhancements.

Version 1.1
First version including HTTPS.

Version 1.1.2
Header files improved in order to allow compiling separated project (secure and not secure).

Version 1.2
Solved problem regarding socket reuse when reaching 16 connections.

Version 1.2.1
Improved documentation (this manual).

Version 1.2.5
New version allowing user to set output buffer size.

Version 1.2.6
Improved end of reception detection: header "Accept-Encoding:identity" included by default.

Version 1.2.7
New function in class HTTPResponse: HTTPHeaders* getHeaders().
New Targets: Expanded Mode (A4/A5-relative data) and Expanded with A5-based Jumptable.

Version 1.2.8
Fixed: parser error in URL with parameters in function: "short execute(HTTPRequestType verb, char* url, char *data = NULL)" (Example: www.alerts.com/sky.asp?root=125).

Version 1.2.9
Fixed: error detected when using one connection and executing multiple requests.

Version 1.3.0
Fixed: Time Out error when using secure connections (HTTPS), it didn't wait for server's answer.

Version 1.3.1
Fixed: error reported in HTTPS when using one connection and executing multiple requests.

Version 1.3.2
New Targets: 4 byte "int".

Check our website **http://www.pdadevelopers.com** for more information on our products.