The powerful IWEB API

Abstract

This article gives a brief describes in about how to use IWEB and related APIs like IWebResp, IWebOpt and IWebUtil that were first introduced in the 1.1 SDK. It also touches upon this API's capabilities and how this API interfaces with other APIs like IHTMLVIEWER.

Basic Functioning

The main purpose of the IWEB API is to make HTTP requests using a BREW application. In HTTP one can request for files using the HTTP "GET" method or upload data using the HTTP "POST" method. IWEB supports both these methods. IWEB manages its own connections and sockets thereby making it much easier to do web transactions.

The IWEB_GetResponse function is used to kick off a web transaction.

```
void IWEB_GetResponse(
               IWeb * pIWeb,
               IWebResp * * ppiwresp,
               AEECallback * pcb,
               const char * cpszUrl,
               ...
               )
```

where:

`pIWeb`	A valid pointer to an instantiated IWeb object
`ppiwresp`	A valid pointer to a pointer IWebResp.
`Pcb`	A valid pointer to an instantiated AEECallBack
`cpszUrl`	Pointer to the URL.
`...`	This is a variable list of WebOpt id/value pairs, terminated with WEBOPT_END.

Remember: The IWebResp pointer should not be created on the stack. Ideally this pointer should be in your applet structure so that it is valid through the lifetime of your Callback

The URL should not contain any reserved characters. If it does, then it needs to be encoded accordingly. Refer to the "Using reserved characters in the URL" section of this document for details.

In its simplest form, a file can be requested using IWEB as follows:

```
   // Create IWeb instance
   ISHELL_CreateInstance(pMe->a.m_pIShell, AEECLSID_WEB, (void **)&pMe-
   >m_pIWeb);
   // Initialize the callback to WebReadCB
   CALLBACK_Init(&pMe->m_Callback, WebReadCB, pMe);
   // Request IWEB to fetch the
   IWEB_GetResponse(pMe->m_pIWeb,
            (pMe->m_pIWeb, &pMe->m_pIWebResp, &pMe->m_Callback,
            “http://www.qualcomm.com", WEBOPT_END));
```

Note the atypical manner in which parameters are passed to this function.

IWEB creates a network connection with the server hosting the URL “www.qualcomm.com “and requests the file. When the IWEB engine gets a response from the server it calls the applications AEECallBack passed as pcb. If the method to be used to get the file is not specified, IWEB uses “GET” by default. For details on how to specify the method and how to specify other options please refer to the “*Adding options (WebOpts) to your IWEB_GetResponse request***”** section of this document.

The application can extract the response in the callback using IWEBRESP_GetInfo. This function populates the WebRespInfo structure which essentially contains the body of the response and other information about the response like error code, length, type, etc. For more details on how to interpret the error codes, please refer to the section “*How to interpret WebRespInfo Error Codes?*”:

```
// Callback
static void WebReadCB(void* cxt)
{
      …
      // Get information about the response
      WebRespInfo* pWebRespInfo = IWEBRESP_GetInfo(pMe->m_pIWebResp);
      // the body of the response is contained in the ISOURCE within
      ISource* pISource = pWebRespInfo->pisMessage;
}
```

Adding options (WebOpts) to your IWEB_GetResponse request

Now that you are familiar with how to kick off a simple HTTP transaction using IWEB, let’s explore how to tweak this API. This can be done using WebOpts (Web Options).

A WebOpt holds an ID/value pair. These need to be added to the IWEB_GetResponse call to configure the request being sent out. WebOpts can be added in 2 ways:

Using IWEB_AddOpt

The options added using this method is permanent. The options are valid as long as the IWEb object is not released.

```
int    i = 0;
WebOpt awo[10];

// Add handler data
awo[i].nId  = WEBOPT_HANDLERDATA;
awo[i].pVal = (void *)pMe;
i++;

// Add the handler function for status callbacks
awo[i].nId  = WEBOPT_STATUSHANDLER;
awo[i].pVal = (void *) WebStatusNotification;
i++;

// Add the option stating which method
awo[i].nId  = WEBOPT_METHOD;
awo[i].pVal = (void *) “GET”;
i++;

 // Marks the end of the array of WebOpts
awo[i].nId  = WEBOPT_END;

// Add Options
IWEB_AddOpt(pApp->m_pIWeb,awo);

IWEB_GetResponse(pMe->m_pIWeb,
            (pMe->m_pIWeb, &pMe->m_pIWebResp, &pMe->m_Callback,
            "http://webber.qualcomm.com",
            WEBOPT_END));
```

Directly

The WebOpt ID and the value are specified directly in the IWEB_GetResponse call as follows:

```
IWEB_GetResponse(pMe->m_pIWeb,
            (pMe->m_pIWeb, &pMe->m_pIWebResp, &pMe->m_Callback,
               "http://webber.qualcomm.com",
                     WEBOPT_HANDLERDATA, pMe,
                     WEBOPT_STATUSHANDLER, WebStatusNotification,
               WEBOPT_METHOD, "GET",
               WEBOPT_END));
```

The options added using this method apply only to the IWEB_GetResponse call in which they are used.

Remember: The values for the options added to a request, irrespective of which of the above methods was used, should be valid for the life time of the callback. They are not copied unless WEBOPT_COPYOPTS is included in the list of options.

Commonly used WebOptions and their explanations are given below. Sometimes the comments in the AEEWeb.h file might have more detail about the webopts than the API reference:

WEBOPT_METHOD: Used to specify which method “GET” or “POST” to use.
e.g values: “GET”, “POST”
WEBOPT_BODY: Used for the body of a POST message

WEBOPT_CONNECTTIMEOUT: The time for which IWEB should wait while attempting a successful connection before reporting an error. Its units are in milli-seconds and defaults to AEENet connect timeout. 0 means system default, -1 means infinite. This is strictly for delays in being able to make a successful connection. If after making a successful connection, the server takes a long time to respond, IWEB will NOT report a connection timeout. The application could implement its own timer using ISHELL_SetTimer
WEBOPT_IDLECONNTIMEOUT: The time for which an idle connection is kept open before being torn down. The default is equal to the INetMgr PPP linger time, passing -1 sets it to infinite.
WEBOPT_HEADER: This is used to specify a particular HTTP header. Its value consists of CRLF-separated, name-value pairs like this: "Name1: v1\r\nName2: v2\r\n".
e.g. values: "X-Method: POST\r\n"
WEBOPT_PROXYSPEC: Used to specify the Proxy. For details please refer to the BREW Web Proxy Spec Knowledge Base. >
WEBOPT_FLAGS: This influences the behavior of the underlying protocol engine. These flags give more control over how connections are used. The flags which have been implemented are:

- WEBREQUEST_NOKEEPALIVE: IWEB will not try to keep the connection alive (persistent). It will disconnect soon after data transfer. By default IWEB tries to use Keep Alives.
- WEBREQUEST_FORCENEWCONN: IWEB will create a new connection for each transaction even if a connection to the same server already exists. Here you run the risk of running out of sockets on the handset and the transaction failing. By default IWeb tries to re-use the existing connection.
- WEBREQUEST_NOWAITCONN: Creates a new connection if no existing connection is available. If a connection is available but the connection is not idle, IWEB would normally wait till the current transaction completes before starting another. If this flag is set, IWEB will create a new connection immediately.

For details on the other options, please refer to AEEWeb.h

How to interpret WebRespInfo Error Codes?

I've sent a request to the server, I have received the response, but what do I make of it? As explained above, the callback contains a pointer to the IWebResp object. This object can be used to get information about the response in the form of IWebRespInfo as follows:

```
// Callback
static void WebReadCB(void* cxt)
{
      …
      // Get information about the response
      WebRespInfo* pWebRespInfo = IWEBRESP_GetInfo(pMe->m_pIWebResp);

      …
}
```

pWebRespInfo->nCode contains the status of the response, whether it was successful or not. A positive number is one that was returned by the server while negative error codes are system errors present in AEEError.h. To interpret the value in nCode, the following macros should be used:

WEB_ERROR_SUCCEEDED(nCode): This macro returns TRUE if transaction succeeded
WEB_ERROR_MAP(nCode): This returns WEB_ERROR_PROTOCOL if the error is an HTTP error. Refer to AEEWeb.h or RFC2626 to interpret what the error with value nCode means.

Inspecting the Status and Headers of the Response

An application can easily inspect the individual headers returned in the response from the server if need be. The WEBOPT_HEADERHANDLER and WEBOPT_STATUSHANDLER options can be set in the response. These options register callback functions for the headers and the status respectively.

Usage:

```
IWEB_GetResponse(pApp->m_pIWeb,
                   (pApp->m_pIWeb, &pwa->piWResp, &pwa->cb, pszUrl,
                     WEBOPT_HEADERHANDLER, WebAction_Header,
                     WEBOPT_STATUSHANDLER, WebAction_Status,
                     WEBOPT_END));
```

WebAction_Header is of type PFNWEBHEADER and WebAction_Status is of type PFNWEBSTATUS. For more details please refer to the API reference document.

Making HTTP POST requests

The HTTP POST request involves POST'ing data to the server. To do this IWEB needs a stream of data to be sent to the server.

```
   // set up the callback
   CALLBACK_Init(&pMe->m_Callback, WebReadCB, pMe);
   // Create a Source Util object which will create an ISource object
   // from a buffer, file, socket, etc.
   ISHELL_CreateInstance(pMe->a.m_pIShell, AEECLSID_SOURCEUTIL, (void
   **)&pMe->m_pISourceUtil);


   // Create ISOURCE object
   ret = ISOURCEUTIL_SourceFromMemory(pMe->m_pISourceUtil,
                       pMe->m_szData, // data in buffer
                       pMe->m_nContentLength, // length of data
                       NULL,// No callback
                       NULL,
                       &pMe->m_pISource //object to be created
                       );
   // Kick off the transaction
   IWEB_GetResponse(pMe->m_pIWeb,
               (pMe->m_pIWeb,
               &pMe->m_pIWebResp,
               &pMe->m_Callback,
               pMe->m_pURL,
               WEBOPT_METHOD, "POST", // Set method to POST
               WEBOPT_BODY, pMe->m_pISource, // Set body of message
                                             // to ISOURCE object
               WEBOPT_CONTENTLENGTH, pMe->m_dwContentLength,
               WEBOPT_END));
```

Just like with a "GET" request, the callback will be called with the appropriate codes and data.

Note: HTTP POST is known to have issues in BREW 1.x and 2.0 when the body of the POST is big.

Using reserved characters in the URL:

While making an HTTP request it is very important that the URL not contain any reserved characters. If any reserved characters need to be sent then URl needs to be encoded into an acceptable format before it can be sent in the request. This can be done using IWEBUTIL_UrlEncode.

```
    char * IWEBUTIL_UrlEncode
    (
    IWebUtil * pIWebUtil,
    const char * cpcIn,
    int * pnInLen,
    char * pcOut,
    int * pnOutLen
    )
```

This method encodes characters in a string for inclusion in a URL, according to RFC2936.

Remember: Only the suffix of the URL string which contains the reserved characters should be encoded and NOT the whole URL.

Say the URL was something like: "http://www.qualcomm.com/test?param1=x
Encode only "test?param1=x" part of the URL. A sample usage of this is as follows:

```
// Determine what the length of the output buffer needs to be
IWEBUTIL_UrlEncode(pMe->m_pIWebUtil,
            (const char *)(buf),    // buffer containing the suffix
                                 // of URL with reserved chars
            &nDataLen,              // number of characters to be
            encoded
            NULL,                         // if NULL, method calculates
                                 //size of buffer needed
            &pMe->m_dwContentLength // contains the length
                                     // of the buffer required
                  );
// Allocate space to hold the encoded string
if(pMe->m_dwContentLength)
      pMe->m_szData = (char*) MALLOC(pMe->m_dwContentLength);

// Encode the string.
IWEBUTIL_UrlEncode(pMe->m_pIWebUtil, (const char *)(buf), &nDataLen,
                  pMe->m_szData, &pMe->m_dwContentLength);
```

The encoded string would look like “test%3Fparam1%3Dx%0”.
m_szData contains the suffix of the URL in encoded form. This needs to be concatenated with the prefix of the URL and then passed to IWEB_GetResponse.

Keeping a connection alive and Connection caching

HTTP 1.1 introduces the concept of persistent connections using Keep-Alives. IWEB makes use of this feature to keep connections alive. IWEB uses Keep-Alives by default. This can be disabled by using the WEBREQUEST_NOKEEPALIVE WEBOPT_FLAG in the request.
If an app makes a request to a server “www.qualcomm.com/index.html“ IWEB opens a connection this server. After the request has been serviced, IWEB does not close the connection. It caches the connection. If any further requests are made like “www.qualcomm.com/developer.html” which needs a connection to the same server, it tries to reuse the same connection. This again can be turned off using WEBOPT_FLAGS.

However, there still are a few limitations of persistent connections. Keepalives are dependent on the Content-Length headers. You will only see a "Connection: Keep-Alive" when there is a valid "Content-Length: xxx" header. For the remote side to know when to stop reading the stream either the content length is required or the connection needs to be brought down.

So, most servers will fall back on "Connection: Close" when serving up CGI or other dynamic content, as it is harder/not possible to provide a valid "Content-Length: xxx" header on the fly.

Even if there is a valid Content Length header, the server is free to close the connection at any time. This is part of the HTTP standard. There is no way to guarantee a persistent connection.

Using IWEBUTIL_ParseFormFields to make life easier.

This utility method proves to be quite a boon when an application needs a user to provide some information which then has to be sent to a server.
The BREW application can make use of the IHTMLViewer API to display an HTML form on a handset. This form can be used to prompt the user to enter necessary information. This information may include data to be sent to the server and also information that could be used by the application to operate in a manner that the user desires.

For e.g: It could provide a text box for the user to enter the name of a ringer. This information could be sent to the server to retrieve the ringer file. In the same form, it could provide a volume control checkbox, for the user to select the volume of sound played back. Although this information is not sent to the server it is useful to the application to modify its behavior accordingly.

When the user clicks the submit button on the form, the HVN_SUBMIT code is sent to the HTMLViewer's callback function. Along with this code the URL is also passed. This URL has a suffix that contains the form field and their values embedded in x-www-urlencoded form. i.e.
name1=value1&name2=value2&name3=value3& ...
or:
name1=value1;name2=value2;name3=value3; ....

This suffix can be fed to IWEBUTIL_ParseFormFields to get the individual field and their respective values as follows:

Say the URL passed to the callback function was:
"request:ringer?ringer=Sacrifice&volume=50"

Using IWebUtil to get the individual fields:

```
// Find the suffix
pszIter = STRCHREND(pszIter, '?');
if (*pszIter)
    ++pszIter;

{
    IWebUtil     *piwu;
    WebFormField  wff[4]; // Create an array of WebFormField structs

    ISHELL_CreateInstance(pApp->a.m_pIShell, AEECLSID_WEBUTIL,
                          (void **)&piwu)) ;

// This populates the WebFormField structs with a maximum of 4 fields
    IWEBUTIL_ParseFormFields(piwu,&pszIter,wff,4,0);
    IWEBUTIL_Release(piwu);
    piwu = 0;
}
```

The WebFormFields now contain the name of the field and their respective values.