Winwap Technologies Oy

Client WAP Stack Library

Application Programming Interface

WAP stack version 2.6
WAP specification version: 2.0
Document dated: 8-Dec-2005

Notice of Confidentiality

Revision history

Date	Author	Description
14-Oct-2004	S Markelov	Initial version of the newly designed WAP StackAPI specification.
9-Nov-2004	S Markelov	New WAP HTTP Stack event: "progress".
24-Nov-2004	Maria Sandell	English spell checked.
26-Nov-2004	S Markelov	New events: func_id_cert, func_id_cert_req. New function: wps_set_prm_ptr 3.3.2.
15-Dec-2004	S Markelov	Issuer certificate loading, server certificate verification (wps_set_prm_ptr function 3.3.2 and func_id_cert event).
27-Jan-2005	S Markelov	Timeout tuning for synchronous API functions wps_get 3.7.1 and wps_post 3.7.2 in the WAP HTTP Client Stack.
12-May-2005	M Shkirja	wps_get_prm 3.3.1, wps_set_prm 3.3.1, wps_get_prm_ptr 3.3.2, wps_set_prm_ptr 3.3.2: WSP Capability and WTP timer intervals management.
2-Sep-2005	S Markelov	Size of sent data in progress_id_send 3.4.3 event.
20-Sep-2005	S Markelov	Allow partial HTTP data sending: wps_set_prm 3.3.1.
8-Dec-2005	S Markelov	Appendix with the list of certificate validation codes.

Preamble

The reader of this document should be familiar with all or some of the following in order to fully understand and evaluate the information in this document:

- Basic knowledge in programming techniques.
- Basic understanding of networking connections and client/server architecture where user-agents retrieve and render information (e.g. Internet browsers and servers with services).
- The interaction between a WAP user-agent, a WAP Gateway and a WEB Server.

Contents

1 Definitions

In this document the following definitions have been used:

WAP Stack An Object that is created by the function wps_open 3.5.1.

WAP Stack client An Application that creates the WAP Stack and uses its service(s) through the Application Programming Interface (API).

2 Normative references

RFC-2616 "Hypertext Transfer Protocol – HTTP/1.1", ftp://ftp.isi.edu/in-notes/rfc2616.txt.

WP-HTTP "Wireless Profiled HTTP", http://www.openmobilealliance.org/.

WSP "Wireless Session Protocol", http://www.openmobilealliance.org/.

WTLS "Wireless Transport Layer Security Protocol", http://www.openmobilealliance.org/.

WTP "Wireless Transaction Protocol Specification", http://www.openmobilealliance.org/.

3 API specification

3.1 Declarations

All functional and type declarations are available in the following C-headers files:

wps_e2.h — General API functions and type definitions.
wps_utils.h — API functions not belonging to data transfer operations.
wps_errno.h — Named error constants.
wps_types.h — Type definitions. These types are used as types for functional parameters and as WAP Stack internal types.

3.1.1 Obsolete and deprecated declarations

wpsutils.h — Obsolete name of the file wps_utils.h.
wps_e.h — Deprecated general API of the WAP Stack. Provided only for backward compatibility.

If you wish to use the deprecated API in the source code of a client application, the macros WPS_DEPRECATED_API must be defined. The WAP Stack Library still exports the deprecated API, so that older applications using the deprecated API do not need to be updated. It is strongly recommended, especially for recently developed applications, not to use the deprecated API.

3.2 Library initialization and release

3.2.1 wps_init, wps_fini

NAME

wps_init — initialize the WAP Stack Library
wps_fini — release the WAP Stack Library

SYNOPSIS

```
#include "wps_e2.h"

int wps_init(void);
int wps_fini(void);
```

DESCRIPTION

The wps_init function is used for WAP Stack Library initialization. This function is mandatory to call after loading of the WAP Stack Library and prior to using any other API functions.

The wps_fini function is an opposite of the wps_init function. It is used for releasing the WAP Stack Library and it destroys all allocated WAP Stack Library internal objects. This function shall be called before unloading the WAP Stack Library and after using any other API functions.

RETURN VALUE

On success, zero is returned. On failure, the returned value is an error code.

ERRORS

The wps_init function can return the following error codes:

wps_error_no_memory — Not enough memory for internal object allocations.

wps_error_network — Applicable for Win32 platforms only. Can not initialize Windows Sockets library.

The wps_fini function can return the following error code:

wps_error_network — Applicable for Win32 platforms only. Can not initialize Windows Sockets library.

NOTES

The function wps_fini should not be called in any callback function as it may cause a deadlock situation or hang the application. Neither should the wps_fini be called while any WAP Stack instance exists.

3.2.2 Logging

NAME

wps_init_log — set logger parameters.

SYNOPSIS

```
#include "wps_e2.h"

typedef struct t_log_prm
{
    const char *path;
    t_log_level level;
    size_t split;
} t_log_prm;

int wps_init_log(t_log_prm *prm);
```

DESCRIPTION

The wps_init_log function is used for setting logger parameters. It can also be used for switching the logger on or off. The logger is a global internal object, which is the only one available for all WAP Stack instances. After WAP Stack Library initialization the logger is by default switched off.

The `prm` parameter is a pointer to the t_log_prm structure and sets the logger parameters. The members of the structure are:

`path`	path to the logging file;
`level`	level of logging: how much information to be logged;
`split`	approximate maximum size of the logging file.

The value of the level structure member can be one of the following values:

LOG_DISABLED	switch logging off;
LOG_FATAL	log only critical messages;
LOG_WARNING	log important messages;
LOG_INFORMATION	log yet more messages;
LOG_ALL	log all messages.

The value `split` & 0x7FFFFFFF — sets the maximum size of the log file; the highest bit of the `split` value requires the current log file to be moved and start a new log file immediately. The logging data will be appended to the current log file in case the highest bit is 0. When the size of the log file named "logfile" becomes greater than `split` & 0x7FFFFFFF value, the logged file "logfile" is renamed to file "logfile.1", i.e. ".1" is appended to the filename. If the file "logfile.1" already exists it will be removed. Logging will then continue with the new file "logfile". The `split` value does not guarantee that the new log file will be used immediately when the size of the active log file reaches the `split` & 0x7FFFFFFF value. The logger checks the file size only during client activities (inside of wps_get 3.7.1, wps_post 3.7.2, wps_connect 3.6.1 or wps_disconnect 3.6.2 functions) or when the stack receives data.

The wps_init_log function can be called at any time. Received and sent packages are logged with the level LOG_INFORMATION. Please be careful with this level as all data sent and received on any of the stack layers WSP, WTP and WTLS will be logged. This includes both data before encryption (i.e. encrypted data) and the same data after encryption (i.e. decrypted plain data).

RETURN VALUE

On success, zero is returned. On failure the returned value 1 indicates that the parameter has an incorrect value. Any other value is an error code.

ERRORS

wps_error_system — System error occurred.

3.3 Tuning

3.3.1 wps_get_prm, wps_set_prm

NAME

wps_get_prm — retrieve a numerical parameter value.
wps_set_prm — set a new numerical parameter value.

SYNOPSIS

```
#include "wps_e2.h"

int wps_get_prm(t_hwps hwps, enum t_wps_layer layer, int prm_id,
        int *val);

int wps_set_prm(t_hwps hwps, enum t_wps_layer layer, int prm_id,
        int val);
```

DESCRIPTION

The wps_get_prm and wps_set_prm function calls manipulate the values of numerical parameters associated with different protocol levels of the WAP Stack.

The parameter hwps sets the WAP Stack handle, which is returned by the wps_open 3.5.1 function call.

The parameter layer sets the WAP WSP protocol level. The present implementation of the WAP Stack includes five protocol levels, each associated with numerical constants as described below:

layer_wdp	WDP	— Wireless Datagram Protocol;
layer_wtls	WTLS	— Wireless Transport Layer Security;
layer_wtp	WTP	— Wireless Transaction Protocol;
layer_wsp	WSP	— Wireless Session Protocol;
layer_http	HTTP	— Wireless Profiled HTTP;
layer_ssl	SSL	— Secure Socket Layer Protocol.

The parameter prm_id identifies the parameter of the given protocol level. The following constants, depending on the protocol level, can be used as values for the prm_id parameter:

WDP:

buf_wmax	Maximum size of the WDP buffer for sending.
buf_rmax	Maximum size of the WDP buffer for reading.

WTP:

sar_enable	Switcher "enable/disable SAR".
sar_first_grp_fcnt	Count of frames of first SAR group.
sar_grp_fcnt	Count of frames of other SAR groups.
sar_fsize	Size of SAR frame.
tit_base_retry	Base retry timer interval in seconds.
tit_base_acknow	Base acknowledgment timer interval in seconds.
tit_wait_timeout	Wait timeout interval in seconds.

WSP:

cap_client_sdu_size	Client SDU Size.
cap_server_sdu_size	Server SDU Size.
cap_proto_options	Protocol Options.
cap_method_mor	Maximum Outstanding Method Requests.
cap_push_mor	Maximum Outstanding Push Requests.
cap_client_msgsize	Client Message Size.
cap_server_msgsize	Server Message Size.
wsp_he_ver	Header encoding version: 0x12 — version 1.2 0x13 — version 1.3 0x14 — version 1.4

HTTP/HTTPS:

http_send_timeout	HTTP request sending timeout.
http_wait_timeout	HTTP server reply timeout.
http_split_size	The Maximum size of a partial message that should be sent in one call of TCP socket send operation.

In case the function call set_wps_prm is used, the parameter `value` sets the value of the given parameter. In case the function call get_wps_prm is used the parameter points to the location where the current value of the requested parameter shall be placed.

RETURN VALUE

On success, zero is returned. On failure the returned value ranging from 1 to 4 indicates the parameter number with incorrect value. Other values are error codes.

ERRORS

wps_error_not_allowed — Attempt to set a parameter value although not allowed.

NOTES

The size of the WDP buffers can be tuned only once before the function wps_bind is called. By default the size of the WDP buffers are set to the maximum values supported by the operating system (OS). 1024 bytes of the WDP buffers are reserved for the WTLS layer. WDP buffers for reading and writing can therefore not be greater than the buffers provided by the OS minus 1024 bytes. The wps_set_prm function call returns no errors in case of very large buffers. After the wps_set_prm function is called, the wps_get_prm function can be used to determine the actual set sizes of the WDP buffers.

The switcher "enable/disable SAR" value can be set either to 0 or 1. The 0 value disables SAR and the 1 value enables SAR. Please note that when several WSP sessions within one WAP Stack instance are active, the switcher is applied to the whole WTP layer (and thus to the whole WAP Stack instance and all WSP sessions). In order to avoid problems with enabling and disabling SAR it is recommended to open sessions with WTP SAR enabled and disabled in different instances of the WAP Stack. In case of one session only you may simply enable/disable SAR without any problems.

The number of frames of SAR groups can range from 1 to 255. Since the first group is sent without knowing the status of the receiver, the number of frames should not be too large. The default values are two frames for the first group and five frames for the other groups.

The SAR frame size can be changed at any time. When SAR is enabled, requests are split in frames of this size. When SAR is disabled, all requests are sent in one frame with the maximum size of the WDP buffer for sending. The SAR frame size can not be smaller than 16 bytes or greater than the size of the WDP buffer for sending. In case the value parameter is not passed, the default value is used. The default SAR frame size is 1400 bytes.

There are three kinds of timers used by the WTP layer: acknowledgment, retry and wait timeouts. Different values of timers are used depending on whether user acknowledgments are used. To modify or retrieve timer values used with user acknowledgment set the highest bit of the `val` parameter. For more details about timer intervals see the WTP protocol specification.

There are number of capabilities defined in the WSP protocol specification. It is possible to access and modify them. But in order to set Client SDU Size and Server SDU Size modify buf_wmax and buf_rmax from the WDP layer. For more information about capabilities see WSP protocol specification.

Header encoding version shall be encoded as defined for the Encoding-Version field in the WAP WSP specification. Currently the WAP Stack accepts the following hexadecimal values:

0x12 — Version 1.2;
0x13 — Version 1.3;
0x14 — Version 1.4.

The default header encoding version is 1.3.

The wps_get_prm and wps_set_prm functions allow to determine or set timeout values for HTTP request sending and HTTP response waiting operations. The timeout values are in milliseconds. These values can be changed by the WAP Stack client at any moment, but the new values do not take effect on currently active requests.

If a HTTP request can not be sent during the time (in milliseconds) set to http_send_timeout option, then wps_get 3.7.1 and wps_post 3.7.2 functions return wps_error_timedout error code. If either 0 or big enough number of milliseconds is assigned to the http_send_timeout option then the wps_get 3.7.1 and wps_post 3.7.2 functions will try to send a HTTP request until TCP timeout occurs.

The default sending timeout is 0 milliseconds, i.e. operations will wait for TCP timeout.

If a HTTP response can't be received during the time (in milliseconds) assigned to http_wait_timeout option, then the func_id_reply 3.4.1 event occurs with 0x48 (HTTP 408 Request Timeout) status code.

The default waiting timeout for HTTP server response is 60 000 milliseconds (60 seconds).

The wps_get_prm and wps_set_prm functions allow to determine or set the maximum size of a partial message. If the value is set and greater than 0 the message to be sent will be split in several parts of the given size and each partial message will be send in the one TCP send operation. So the whole HTTP request will be sent as a set of several messages and the event func_id_progress 3.4.3 with the indicated progress_id_send action will allow to determine in run-time the count of bytes that are already sent as a request to a HTTP Server.

The value 0 switches the HTTP client to its default behaviour that is depended on the used platform:

Win32: the whole request will be always sent in one atomic TCP send operation for both HTTP and HTTPS modes.

Linux: same as in the Win32, but if http_send_timeout is set for HTTP mode then the whole messages will be sent as a set of partial messages with the size determined by the system.

Please note: if the value http_split_size is greater than 0 it will be ignored in HTTPS mode. A partial sending is performed with the size of a message block, which is 16kB for SSLv3/TLSv1.

This parameter may reduce the efficiency of the network operations.

3.3.2 wps_get_prm_ptr, wps_set_prm_ptr

NAME

wps_get_prm_ptr — retrieve not numerical parameter value.

wps_set_prm_ptr — set not numerical parameter value.

SYNOPSIS

```
#include "wps_e2.h"

int wps_get_prm_ptr(t_hwps hwps, enum t_wps_layer layer, int prm_id,
        void *ptr, size_t *size);

int wps_set_prm_ptr(t_hwps hwps, enum t_wps_layer layer, int prm_id,
        const void *ptr, size_t size);
```

DESCRIPTION

The wps_get_prm_ptr and wps_set_prm_ptr function calls manipulate the values of not numerical parameters associated with different protocol levels of the WAP Stack.

The hwps parameter sets the WAP Stack handle, which is returned by wps_open 3.5.1.

The parameter layer sets the protocol level. The present implementation of the WAP Stack includes five protocol levels, each associated with the following numerical constants:

layer_wdp	WDP — Wireless Datagram Protocol;
layer_wtls	WTLS — Wireless Transport Layer Security;
layer_wtp	WTP — Wireless Transaction Protocol;
layer_wsp	WSP — Wireless Session Protocol;
layer_http	HTTP — Wireless Profiled HTTP.
layer_ssl	SSL — Secure Socket Layer Protocol.

The prm_id parameter identifies the parameter of the given protocol level. The following constants, depending on the protocol level, can be used as values for the prm_id parameter:

WSP:

cap_ext_method	Extended Methods.
cap_hcodepage	Header Code Pages.
cap_alias	Aliases.

WTLS:

ke_avail_list	list of available key exchange suits (read-only);
be_avail_list	list of available bulk encryption algorithms (read-only);
mac_avail_list	list of available keyed MAC algorithms (read-only).

SSL (HTTPS):

ssl_cert	client's certificate and private key (write-only);
ssl_ca_cert	trusted issuer certificate that will be used for verification (write-only).

In case of the wps_get_prm_ptr function the ptr parameter points to the allocated memory buffer where the requested data shall be placed. The returned data is a byte array of values and the size of the array is returned through the size pointer.

The size parameter points to the location with the size of the allocated memory buffer placed. The size argument must not be NULL. The size of the placed data is stored in the location pointed by the size argument.

In case of the wps_set_prm_ptr function the ptr parameter points to the allocated memory buffer where the data to be assigned is placed.

The size parameter contains the size of the data to be assigned. The size is counted in bytes. Please see section "NOTE" for data format description.

RETURN VALUE

On success, zero is returned. On failure the returned value ranging from 1 to 4 indicates the parameter number with incorrect value. Other values are error codes.

ERRORS

The wps_get_prm_ptr function can return only one error code:

wps_error_no_space — Not enough space for the returned array. size parameter contains needed size.

The wps_set_prm_ptr function can return only one error code:

wps_error_no_memory — Not enough memory for new value.

NOTES

The wps_set_prm_ptr function guarantees that the old value will not be changed in case of error.

To assign new private key and client certificate the WAP Stack client should call wps_set_prm_ptr function for layer_ssl and parameter ssl_cert. The passed data is the filled structure defined as follows:

```
struct t_cert_cli_data
{
    enum t_cert_type cert_type;
    const void *cert;
    size_t cert_size;
    enum t_pkey_type pkey_type;
    const void *pkey;
    size_t pkey_size;
};
```

The members of the structure are:

cert_type	certificate type: cert_type_x509 — X509 certificate;
cert	pointer to the first byte of the DER-encoded[1]1 certificate;
cert_size	size of the DER-encoded certificate;
pkey_type	private key type: pkey_type_rsa — RSA private key, pkey_type_dsa — DSA private key;
pkey	pointer to the first byte of the DER-encoded private key;
pkey_size	size of the DER-encoded private key.

The size parameter must be set to sizeof(t_cert_cli_data).

The WAP Stack client should not contain a private key in its executable file. A violator can extract the private key from the executable file.

A UNIX user shall remove a dumped core file if an application that uses the WAP Stack Library has crashed. A violator can extract the private key from the core file.

To add a new issuer certificate the WAP Stack client should call wps_set_prm_ptr function for layer_ssl and parameter ssl_ca_cert. The passed data is a structure defined as follows:

```
struct t_cert_ca_data
{
    enum t_cert_type cert_type;
    const void *cert;
    size_t cert_size;
};
```

The members of the structure are:

cert_type	certificate type: cert_type_x509 — X509 certificate;
cert	pointer to the first byte of the DER-encoded certificate;
cert_size	size of the DER-encoded certificate.

The size parameter must be set to sizeof(t_cert_ca_data).

The WAP Stack can only add an issuer certificate to the list of trusted certificates. To clear the list of trusted certificates and assign new issuer certificates the WAP Stack client shall close a WAP Stack instance using wps_close 3.5.2 and open a new WAP Stack instance using wps_open 3.5.1.

To set or get "Extended Methods", "Header Code Pages" or "Aliases" capabilities call wps_set_prm_ptr or wps_get_prm_ptr function for layer_wsp and corresponding capability in third parameter. The ptr parameter shall point to next structures or array of those structures.

```
struct uint8_and_str
{
    unsigned char *num;
```

[1]The DER format is "Distinguished Encoding Rules" and is the most commonly used binary form for ASN.1-specified objects.

```
    const char *str;
};

typedef struct uint8_and_str t_ext_method;
typedef struct uint8_and_str t_hcodepage;

struct t_address
{
    unsigned char bearer;
    unsigned short port;
    const char *address;
    size_t addrlen;
};

typedef struct t_address t_alias;
```

The members of the structures are:

num	either PDU Type for method or code for header page;
str	either name of method or name of header code page;
bearer	type of bearer network to use. 0xFF indicates no bearer;
port	port number to use. 0 indicate no port number;
address	bearer address to use;
addrlen	length of the address field.

3.3.3 wps_set_wtlsclient_id

NAME

wps_set_wtlsclient_id — assign WTLS client identifier.

SYNOPSIS

```
#include "wps_e2.h"

int wps_set_wtlsclient_id(unsigned char type,
        const unsigned char *id, unsigned char size,
        unsigned short charset);
```

DESCRIPTION

The wps_set_wtlsclient_id function is used for filling the WTLS structure Identifier, which is global for all WAP Stack instances.

The parameter `type` is an identifier which can be assigned to the following values:

0 — null
1 — text
2 — binary

The parameter `id` is a pointer to an identifier of the given type. In case the type is null, it is ignored. If the parameter `id` is NULL and `type` is not null, the default identifier will be set. The default identifier is a value of binary type: 0x00000000A5B5C5D5E50000.

The parameter `size` is an identifier size in bytes, which is ignored for null type.

The parameter `charset` sets the character set of the text identifier. In case `type` differs from text, it is ignored. `charset` is a unique MIBenum value that can be used to identify coded character sets.

RETURN VALUE

On success, zero is returned. On failure the returned value ranging from 1 to 3 indicates the parameter number with incorrect value.

NOTES

The function wps_set_wtlsclient_id is not thread safe.

3.4 Register event handlers

Starting from version 2.4 of the WAP Stack, a new API for manipulating event handlers is provided. Now all callback functions have the same syntax and they can be assigned to the WAP Stack at any time.

Due to the evolution of the WAP Stack 3 old callback functions are still available. However it is recommended to use the new callback API.

3.4.1 wps_reg_callback_func

NAME

wps_reg_callback_func — register callback function that will process events.

SYNOPSIS

```
#include "wps_e2.h"

int wps_reg_callback_func(t_hwps hwps, t_func_id func_id,
        f_callback *func, f_callback **func_old);
```

DESCRIPTION

The wps_reg_callback_func function registers a given function as event handler.

The WAP Stack is identified by the parameter `hwps`.

The `func_id` parameter identifies an event, which will be processed by the given callback function. The present implementation of the WAP Stack can generate five events, each associated with numerical constants as described below:

func_id_connect	WAP WSP Stack connection;
func_id_reply	WAP WSP/HTTP Stack received requested data;
func_id_push	WAP WSP Stack received pushed data;
func_id_wtp_frame	WAP WSP Stack low-level transfer operations;
func_id_request	WAP HTTP Server: received request;
func_id_progress	WAP HTTP Stack: activity indication;
func_id_cert	WAP HTTP/SSL Stack: server certificate
func_id_cert_req	WAP HTTP/SSL Stack: client certificate request.

The address of the function that process a given event is passed through the parameter `func`. One function may process several events. The events can be identified by the `func_id` parameter value of the callback function.

In case the parameter `func_old` is not NULL it returns the address of the previous function that was registered as event handler.

RETURN VALUE

On success, zero is returned. On failure the returned value ranging from 1 to 2 indicates the parameter number with incorrect value. Other values are error codes.

ERRORS

wps_error_not_object — The given hwps value is not a WAP Stack handle.

SEE ALSO

Section Callback functions 3.4.2.

NOTES

The developer of the WAP Stack client should take it into account that callback functions may be called from different threads by the WAP Stack at any time.

When the WAP Stack is in the connectionless mode, the callback function registered as func_id_connect is not called. In this case the API wps_connect function 3.6.1 is synchronous and it can not call any callback functions since its returned value will be unknown to the WAP Stack client.

If a callback function is registered as a func_id_wtp_frame function, it shall not call any WAP Stack API function(s) when processing an event with the identifier func_id_wtp_frame. Please note that in case an attempt to call any WAP Stack API function(s) is made, the program may hang or crash.

The code in the above mentioned function should also execute as fast as possible. In other words, the WAP Stack client must not show e.g. a modal window in the func_id_wtp_frame handler.

It is strongly suggested that GUI implementations should not be made in callback functions. The developer should also avoid using algorithms that may block event handlers or put them in waiting mode.

3.4.2 Callback functions

NAME

f_callback — type declaration of the callback function.

SYNOPSIS

```
#include "wps_e2.h"

typedef void f_callback(t_func_id func_id, void *hobj, void *context,
        const union t_callback_data *data);
```

DESCRIPTION

The type definition f_callback defines the syntax declaration of the callback function.

When the callback function is called, the `func_id` parameter contains a callback function identifier (or event identifier) that can be helpful for identification of the event. This is useful in cases when one function is registered as a handler of several events.

The `hobj` parameter is the request or session handle.

The `context` parameter contains the value given by the WAP Stack client through wps_open 3.5.1. This is generally a pointer to the WAP Stack client.

The pointer `data` points to a union that contains data specific for an event. The event can be identified by the value of the `func_id` parameter in case one function is registered as a handler of several events.

NOTES

The developer of the WAP Stack client should take it into account that callback functions may be called from different threads by the WAP Stack at any time.

If a callback function is registered as a func_id_wtp_frame function, it shall not call any WAP Stack API function(s) when processing an event with the identifier func_id_wtp_frame. Please note that in case an attempt to call any WAP Stack API function(s) is made, the program may hang or crash.

The code in the above mentioned function should also execute as fast as possible. In other words, the WAP Stack client must not show e.g. a modal window in the func_id_wtp_frame handler.

It is strongly suggested that GUI implementations should not be made in callback functions. The developer should also avoid using algorithms that may block event handlers or put them in waiting mode.

SEE ALSO

Section Event data 3.4.3.

EXAMPLE

Below is a C++ code sample of an event handler. This event handler reports about all sent and received WTP packets with invocation and result.

```
void on_frame(t_func_id func_id, void *hobj, void *context,
        const union t_callback_data *data)
{
    unsigned long frame_attr = data->wtp_frame_data.attr;
    if (frame_attr & frame_sent)
    {
        switch (frame_attr & frame_type)
        {
        case wtp_pdu_type_invoke: // WTP PDU type Invoke
            std::cout << "--> Invoke ";
            std::cout << ((frame_attr & frame_dir)
                    ? "received\n" : "sent\n");
            break;
        case wtp_pdu_type_result: // WTP PDU type Result
            std::cout << "<-- Result ";
            std::cout << ((frame_attr & frame_dir)
                    ? "received\n" : "sent\n");
            break;
        }
    }
}
```

This handler can be registered in the following way:

```
wps_reg_callback_func(hwps, func_id_wtp_frame, on_frame, NULL);
```

3.4.3 Event data

Although the event handler can be registered to process events of different types, the data passed through the `data` pointer is different for each event type. Below is a description of the data passed through the `data` pointer for each event type.

Connection status data

NAME

t_connect_data — reported data about connection status.

SYNOPSIS

```
#include "wps_e2.h"

struct t_connect_data
{
    t_connect_code code;
    const char *data1;
    int data2;
};
```

DESCRIPTION

The `data` parameter of the callback function func_id_connect points to the t_connect_data.

The members of the structure are:

`code`	returned connection status code;
`data1`	optional connection data, depends on code;
`data2`	optional connection integer data, depends on code.

Replied and pushed data

NAME

t_reply_data, t_push_data — data pushed and replied to by a WAP Gateway.

SYNOPSIS

```
#include "wps_e2.h"

struct t_reply_data
{
    const void *data;
    size_t data_size;
```

```
    const char *ct;
    const char *hdr;
    int status;
};

struct t_push_data
{
    const void *data;
    size_t data_size;
    const char *ct;
    const char *hdr;
    int status;
};
```

DESCRIPTION

The `data` parameter of the callback function func_id_reply points to the t_reply_data structure.

The `data` parameter of the callback function func_id_push points to the t_push_data structure.

The members of the structures are:

`data`	received data;
`data_size`	data size;
`ct`	content type of the received data;
`hdr`	server HTTP messages;
`status`	status code.

The status code is encoded as defined in the WAP WSP specification, Appendix A.

Request data

NAME

t_http_req_data — full information about a received HTTP request.

SYNOPSIS

```
#include "wps_e2.h"

struct t_http_req_data
{
    const char *ip;
    unsigned short port;
    t_method method;
    const char *url;
    const char *hdrs;
    const void *data;
    size_t data_size;
    const char *ct;
```

```
    int accept;
};
```

DESCRIPTION

The data parameter of the callback function func_id_request points to the t_http_req_data structure.

The event func_id_request is called in the WAP HTTP Server Stack when an incoming HTTP request is fully or partially received.

The value accept allows the WAP Stack client application to determine if the request has been fully received and processed by the WAP HTTP Server Stack. Please note the difference between a "WAP Stack client application" and a "WP-HTTP client". The WP-HTTP client is generally a remote application, which sends requests to the WAP HTTP Server. These requests are processed by the WAP Stack client application.

The members of the structure are:

ip	IP address of the WP-HTTP client sending a request;
port	network port number of the WP-HTTP client sending a request;
method	request type:
	method_get — method GET: the WP-HTTP client wishes to retrieve data, identified by the URL;
	method_head — method HEAD: same as the method GET, but the client doesn't wish to retrieve real data;
	method_post — method POST: similar to the method GET, but additional data is available in the data;
	method_options — method OPTIONS: please see RFC-2616 2 for further explanation.
url	requested resource location;
hdrs	HTTP headers, in other words additional messages;
data	additional data supplied with the request;
size	size of supplied data;
ct	0-terminated string with data content type;
accept	this filed can be assigned with the following values: 0 — the request is fully received and processed by the WAP HTTP Server Stack 1 — the request is not yet fully received and only partially processed The field value can be changed by the WAP Stack client. Please see below.

NOTES

For each received request a new WP-HTTP session in an own thread is opened and the func_id_request event is called. The requested data is passed from the structure t_http_req_data. The member accept of the structure can be set to two values:

- 1 — not all data is received.

  The WAP Stack client shall wait for the next func_id_request event or it can assign 0 to the accept member for termination of the current TCP connection.

- 0 — all data is received.
  The WAP Stack client can process the received data and send a reply using the wps_reply function.

After the event is processed, the TCP connection will be terminated if the `accept` field is set to 0.

The wps_reply 3.8.1 function is used to reply on the received request. The wps_reply 3.8.1 function shall be called in the func_id_request event handler or the event handler can be stopped for the time required for the call of the wps_reply 3.8.1 function. For example the Win32 SendMessage function can be used to stop the caller thread execution and execute some functions in another thread. There is no API for termination of TCP connection provided. The TCP connection can be broken only in the func_id_request event handler by assigning 0 to the `accept` member.

EXAMPLE

Below is a C sample code of the event handler reporting into stdout about received HTTP requests.

```
void on_request(t_hreq req, t_http_req_data &data)
{
    const char *templ =
        "Request from address: %s:%u\n"
        "Request method: %s\n"
        "Requested URL: %s\n"
        "Request headers: \n%s\n"
        "Attached data is of type: %s\n"
        "Attached data is of size: %u\n";

    if (0 == data.accept)   // all data ?
    {
        const char *method;
        switch (data.method)
        {
        case method_get:
            method = "GET";
            break;
        case method_post:
            method = "POST";
            break;
        case method_head:
            method = "HEAD";
            break;
        case method_options:
            method = "OPTIONS";
            break;
        }
        printf(templ, data.ip, data.port,
            method, data.url ? data.url : "", data.hdrs,
```

```
                data.ct ? data.ct : "", data.data_size);

            // When function will return, TCP connection will be broken.
        }
    }
```

HTTP client progress indicator

NAME

t_progress_data — HTTP client progress indicator.

SYNOPSIS

```
#include "wps_e2.h"

struct t_progress_data
{
    enum t_progress_id id;
    size_t data_size;
    size_t hdrs_size;
};
```

DESCRIPTION

The t_progress_data structure is pointed by the `data` parameter of the callback function func_id_progress.

The members of the structure:

`id`	identifies the last finished action of the WAP HTTP client:
	progress_id_connect — connection to the HTTP server is established,
	progress_id_send — request is partially or fully sent,
	progress_id_recv — response data is partially or fully received;
`data_size`	size of the received or sent data;
`hdrs_size`	size of the headers in the received data.

The value of the member `data_size` is applicable for both progress_id_send progress_id_recv WAP HTTP client actions. The value of the member `hdrs_size` is only applicable for the progress_id_recv WAP HTTP client action.

NOTES

The event func_id_progress with the indicated progress_id_recv action can be called several times before the event func_id_reply is called.

progress_id_send action indicator can be called several times.

The event func_id_progress with the indicated progress_id_send action allows to determine the count of bytes that are already sent as a request to a HTTP Server. The data_size value

indicates the size of the sent part of the entire HTTP request including HTTP headers and HTTP body. Please see wps_set_prm 3.3.1 for additional information.

The event func_id_progress with the indicated progress_id_recv action allows to determine the count of bytes that are already received as a HTTP Server response. This data is buffered in the WAP HTTP Client Stack and after the required transformations it will be passed through the func_id_reply event when the whole response has been received. The data_size value doesn't indicate the content data and headers size, since the content may be compressed and HTTP headers can be modified after decompression (decompressed content and modified HTTP headers are passed through the func_id_reply event).

The value hdrs_size contains the size of the HTTP headers. The HTTP headers is only part of the entire HTTP response. Generally, the size of the compressed content in the HTTP response can be calculated as data_size − hdrs_size − 4.

SEE ALSO

wps_set_prm 3.3.1

WTP frame data

NAME

t_wtp_frame_data — WTP process event data.

SYNOPSIS

```
#include "wps_e2.h"

struct t_wtp_frame_data
{
    unsigned char id;
    size_t size;
    unsigned long attr;
};
```

DESCRIPTION

The `data` parameter of the callback function func_id_wtp_frame points to the t_wtp_frame_data structure.

The members of the structure are the following:

`id`	frame id (from 0 till 255);
`size`	frame size;
`attr`	frame attributes.

To determine frame attribute values, the following masks are defined:

frame_resend	0 — first sending, 1 — re-sending;
frame_last	last frame in WTP message;
frame_group	last frame in group;
frame_dir	0 — outgoing frame, 1 — incoming frame;
frame_sent	for outgoing frame: 0 — about start of frame sending, 1 — frame has been sent;
frame_type	WTP PDU type. The attribute can contain the following values that define possible types of the WTP PDU:

wtp_pdu_type_invoke	Invoke WTP PDU;
wtp_pdu_type_result	Result WTP PDU;
wtp_pdu_type_ack	Ack WTP PDU;
wtp_pdu_type_abort	Abort WTP PDU;
wtp_pdu_type_s_invoke	Segmented Invoke WTP PDU;
wtp_pdu_type_s_result	Segmented Result WTP PDU;
wtp_pdu_type_nack	Negative Ack WTP PDU.

All above masks can be used with the bitwise operator AND to determine an attribute value.

NOTES

The developer of the WAP Stack client should take it into account that callback functions may be called by the WAP Stack from different threads at any time.

If a callback function is registered as a func_id_wtp_frame function, it shall not call any WAP Stack API function(s) when processing an event with the identifier func_id_wtp_frame. Please note that in case an attempt to call any WAP Stack API function(s) is made, the program may hang or crash.

The code in the above mentioned function should also execute as fast as possible. In other words, the WAP Stack client must not show e.g. a modal window in the func_id_wtp_frame handler.

It is strongly suggested that GUI implementations should not be made in callback functions. The developer should also avoid using algorithms that may block event handlers or put them in waiting mode.

Server certificate

NAME

t_cert_data — received server certificate.

SYNOPSIS

```
#include "wps_e2.h"

struct t_cert_data
```

```
{
    enum t_cert_type type;
    const void *data;
    size_t size;
    int accept;
    int reason;
};
```

DESCRIPTION

The `data` parameter of the callback function func_id_cert points to the t_cert_data structure.

The members of the structure are the following:

`type`	certificate type: cert_type_x509 — X509 certificate;
`data`	pointer to the first byte of the DER-encoded[1] certificate;
`size`	size of the DER-encoded certificate;
`accept`	this field can be assigned the following values: 0 — verification of the received certificate failed 1 — received certificate successfully verified previous processing of the func_id_cert event. The field value can be changed by the WAP Stack client: set it to 0 to terminate SSL handshake, set it to 1 to accept the certificate and continue SSL handshake;
`reason`	the reason for the failed verification.

NOTES

The received certificate is fully and successfully verified if the `accept` is 1 and `reason` is 0. The reason codes can be found in the x509_vfy.h file provided with the OpenSSL include files (please see Appendix A). The reason codes and their descriptions can be found also in the OpenSSL verify(1) man page.

Each received certificate is verified separately. The event func_id_cert is called several times for each received certificate in "Server Hello" message during SSL Handshake. Also the event func_id_cert can be called several times for the same certificate, if there are several failure reasons.

The WAP Stack client can add issuer certificates that will be used for verification to the list of trusted certificates. Additional information is available in the wps_set_prm_ptr function description.

SEE ALSO

wps_set_prm_ptr 3.3.2

[1]The DER format is "Distinguished Encoding Rules" and is the most commonly used binary form for ASN.1-specified objects.

Client certificate request

NAME

t_cert_req_data — data of the client certificate request.

SYNOPSIS

```
#include "wps_e2.h"

struct t_cert_req_data
{
    enum t_cert_type cert_type;
    int send;
};
```

DESCRIPTION

The `data` parameter of the callback function func_id_cert_req points to the t_cert_req_data structure.

The members of the structure are as follows:

`type` requested client certificate type:
cert_type_x509 — X509 certificate;

`send` this field can be assigned the following values:
0 — the WAP Stack client should assign its certificate using wps_set_prm_ptr,
1 — informs that the client certificate is already assigned in previous processing of the func_id_cert_req event.
The field value can be changed by the WAP Stack client:
set it to 0 to terminate SSL handshake,
set it to 1 to send assigned certificate and continue SSL handshake.

During a SSL handshake a server may request a certificate from the client. A client certificate will only be sent, when the server has sent the certificate request.

SEE ALSO

wps_set_prm_ptr 3.3.2

3.5 WAP Stack

3.5.1 Creation

NAME

wps_open — Open the WAP Stack.

SYNOPSIS

```
#include "wps_e2.h"

int wps_open(t_hwps *hwps, void *client, t_conn_mode mode);
```

DESCRIPTION

The wps_open function creates a new WAP Stack. With this object, the client application can open sessions with any WAP Gateway.

The function wps_open writes the handle of the created WAP Stack into a location given by the `hwps` parameter. The pointer `hwps` must not be NULL.

The value of the parameter `client` is used as a context value in the callback functions. It allows identification of a client in the callback handler. Generally the value `client` is a pointer to the client object.

The `mode` parameter sets the connection mode that will be used for all sessions of the opened WAP Stack. The `mode` parameter can be set to one of the following values, defined in the file wps_e2.h:

CL	WAP WSP connectionless mode protocol;
CO	WAP WSP connection-oriented protocol;
SCL	WAP WSP secure connectionless protocol;
SCO	WAP WSP secure connection-oriented protocol;
PUSHCL	WAP WSP PUSH connectionless protocol;
HTTP	WP-HTTP protocol, client mode;
HTTPS	WP-HTTP/SSL protocol, client mode;
SRV_HTTP	WP-HTTP protocol, server mode.

RETURN VALUE

On success, zero is returned. On failure the returned value 1 indicates that the parameter `hwps` is NULL. Other values are error codes.

ERRORS

wps_error_no_memory — There is not enough memory for creation of a new WAP Stack.

NOTES

WP-HTTP/SSL protocol can be used only if OpenSSL libraries (http://www.openssl.org/) are available in the system.

3.5.2 Destroying

NAME

wps_close — Close the WAP Stack.

SYNOPSIS

```
#include "wps_e2.h"

int wps_close(t_hwps hwps);
```

DESCRIPTION

The function wps_close closes the opened WAP Stack.

The `wps` parameter is a handle of the opened WAP Stack and is returned by the wps_open 3.5.1 function.

RETURN VALUE

On success, zero is returned. On failure an error is returned.

ERRORS

wps_error_not_object — The given `hwps` value is not a WAP Stack handle.

SEE ALSO

wps_open 3.5.1.

3.5.3 Binding to system network interface

NAME

wps_bind — Bind the WAP Stack to a system network interface.

SYNOPSIS

```
#include "wps_e2.h"

int wps_bind(t_hwps hwps, const char *addr, unsigned short port);
```

DESCRIPTION

The wps_bind function is used to bind the WAP Stack to a custom system network address. The operation of binding is necessary in order to start receiving data from the network. It can also be used if the WAP Stack client does not listen to all local network interfaces. It allows receiving of responses from a WAP gateway or HTTP proxy only over a given IP address.

The `wps` parameter is a handle of the opened WAP Stack. It is returned by the wps_open 3.5.1 function.

The `addr` parameter is an IP address of the available system network interface. The WAP Stack will be bound to all available system network interfaces If the parameter is NULL.

The `port` parameter sets the correct listening port number. It is applicable only for the connectionless WAP PUSH Stack and the WAP HTTP Server Stack. The `port` value is ignored for other WAP Stack protocols and instead a port number provided by the system will be used.

RETURN VALUE

On success, zero is returned. On failure the returned value, ranging from 1 to 2, indicates the parameter number with incorrect value. Other values are error codes.

ERRORS

wps_error_not_object — The given `hwps` value is not a WAP Stack handle.

SEE ALSO

wps_open 3.5.1

NOTES

Outgoing requests are sent accordingly to the system IP routing table and requests may be sent from a different IP address. This function is also used for binding of the connectionless WAP PUSH Stack. This function must be called before any other operation with the WAP Stack. It can also be called at any time when there are no active sessions connected. During the binding to the network interface, the WAP Stack tries to determine the maximum sizes of system socket buffers for user data reading and sending. The retrieved values are used for tuning of the WAP Stack internal buffers and during capability negotiation. The applied actual values are logged with the log level LOG_INFORMATION.

3.6 Session establishment and termination

3.6.1 wps_connect

NAME

wps_connect — Create/establish new session.

SYNOPSIS

```
#include "wps_e2.h"

int wps_connect(t_hsession *hses, t_hwps wps, const char *ga, int gp,
        const char *session_headers);
```

DESCRIPTION

The wps_connect function is used for the following:

- establishing an asynchronous connection to the WAP gateway in case of connection-oriented (CO), secure connectionless (SCL) and secure connection-oriented (SCO) modes;
- assigning WAP Gateway addresses in case of connectionless (CL) mode;
- enabling/disabling HTTP Proxy usage. It is also used for HTTP Proxy address assigning in case of WP-HTTP (HTTP) and WP-HTTP/SSL (HTTPS) modes.

The parameter hses is an address of the memory place where a session handle will be placed.

The parameter wps is a WAP Stack handle returned by the wps_open 3.5.1 function.

The parameters ga and gp are interpreted as described below in the different modes:

CL, CO, SCL, SCO
: The parameter ga is a WAP Gateway address. It may be a host name but must not be NULL
The parameter gp is a WAP Gateway port number.

HTTP, HTTPS
: The parameter ga is an HTTP Proxy address, which may be a host name. In case the parameter value is NULL, the WAP HTTP Stack will use direct connections to requested hosts.
The parameter gp is an HTTP Proxy port number.

SRV_HTTP
: The function wps_connect does nothing.

RETURN VALUE

On success, zero is returned. On failure the returned value 1 indicates that the parameter hses is NULL. Other values are error codes.

ERRORS

wps_error_not_object — The given hwps value is not a WAP Stack handle.

wps_error_system — System error occurred.

SEE ALSO

wps_disconnect 3.6.2

NOTES

For connection-oriented (CO), secure connectionless (SCL) and secure connection-oriented (SCO) modes the wps_connect function 3.6.1 is asynchronous. It initializes internal structures, assigns WAP Gateway address, sends connection request to the WAP Gateway and then exits.

In this case the WAP Stack client shall wait for the event func_id_connect to signal about success, failure connection or redirection to an another WAP Gateway address.

In other modes the WAP Stack does not send connection request and the WAP Stack client shall not wait for the event func_id_connect.

All opened session handles, even if the failed result in a func_id_connect event handler is signaled, shall be closed by wps_disconnect 3.6.2.

The session_headers parameter is needed for setting some specific HTTP headers that are required for a successful connection to a WAP Gateway or HTTP proxy. Below is a list of some known HTTP headers that might be required by a WAP Gateway:

Proxy-Authorization
Profile
User-Agent

Some WAP Gateways might use different names for these headers: X-Authorization, X-WAP-Authorization; X-Profile, X-WAP-Profile ...

In case of WAP WSP Stack, connection-oriented (CO) and secure connection-oriented (SCO) modes these session headers are used once during session establishment and generally for authorization.

In case of WAP WSP Stack, connectionless (CL) and secure connectionless (SCL) modes these session headers are not used during session establishment, since in connectionless modes the WSP session is not established.

In case of a WAP HTTP Stack, a new TCP session for each request is established. Therefore these session headers are appended to the HTTP headers that are given in the wps_get 3.7.1 or wps_post 3.7.2 functions. Therefore the developer of the WAP Stack client must be careful and not pass HTTP headers that are already passed through the session_headers parameter. If one HTTP header is included two or more times in the sent HTTP request and this header can not contain a list of values according to the RFC-2616 2, the "Bad Request" response might be received.

3.6.2 wps_disconnect

NAME

wps_disconnect — Disconnect/terminate the session.

SYNOPSIS

```
#include "wps_e2.h"

int wps_disconnect(t_hsession ses);
```

DESCRIPTION

The wps_disconnect function terminates all active requests within the given session, sends session termination request if needed, and closes the session handle.

RETURN VALUE

On success zero is returned. Any other value is an error code.

ERRORS

wps_error_not_object — The given `ses` value is not a WAP Stack session handle.

wps_error_system — System error occurred.

3.7 Data requesting

3.7.1 wps_get

NAME

wps_get — Request a resource through the method GET.

SYNOPSIS

```
#include "wps_e2.h"

int wps_get(t_hreq *hreq, t_hsession session, const char *uri,
        const char *headers);
```

DESCRIPTION

wps_get sends a GET request to a WAP Gateway or HTTP Proxy. The function is asynchronous and therefore the WAP Stack client shall wait for the func_id_reply event.

The `hreq` parameter points to the memory place where the new request handler shall be placed. This handler is needed for request identification.

The `session` parameter identifies a session in which the request will be sent.

The `uri` parameter identifies a requested resource.

The `headers` parameter sets HTTP headers, which will be sent with the request. If the parameter is NULL, default headers will be sent. The headers will be not sent if the string is empty.

The default headers are the following (the version number in the User-Agent header may be different) :

```
Accept-Charset: utf-8, iso-8859-1, iso-10646-ucs-2\r\n
Accept: application/vnd.wap.wmlc,
        application/vnd.wap.wmlscriptc,
        image/vnd.wap.wbmp,
        image/gif\r\n
User-Agent: WAP-Stack-Client/2.6.1
```

RETURN VALUE

On success, zero is returned. On failure the returned value ranging from 1 to 4 indicates the parameter number with incorrect value. Another value is an error code.

ERRORS

wps_error_not_object — The given `session` value is not a WAP Stack session handle.

wps_error_network — (HTTP/HTTPS mode) Can't open or bind TCP socket; can't connect to remote host or proxy. Please verify: proxy address/port; server hostname in URL

wps_error_ssl — (HTTPS mode) Can't connect to remote host using SSL; can't open SSL tunnel through the proxy.

wps_error_system — System error occurred.

wps_error_timedout — Timeout while attempting request. Please see wps_get_prm and wps_set_prm 3.3.1 description for more explanations.

SEE ALSO

wps_abort 3.7.3, wps_set_prm 3.3.1

NOTES

The HTTP headers must be well-formed according to the RFC-2616 2.

3.7.2 wps_post

NAME

wps_post — Request a resource through the method POST.

SYNOPSIS

```
#include "wps_e2.h"

int wps_post(t_hreq *hreq, t_hsession session, const char *uri,
        const void *data, size_t data_size, const char *cont_type,
        const char *headers);
```

DESCRIPTION

wps_post sends a POST request to a WAP Gateway or HTTP Proxy, in other words it posts data from an application. The function is asynchronous and therefore the WAP Stack client shall wait for the func_id_reply event.

The `hreq` parameter points to the memory place where the new request handler shall be placed. This handler is needed for request identification.

The `session` parameter identifies a session in which the request will be sent.

The `uri` parameter identifies a requested resource.

The `data` parameter is a pointer to the data that will be posted.

The `data_size` is a data size.

The `cont_type` is a type of data content. If the value is NULL the "application/x-www-form-urlencoded" value will be sent

The `headers` parameter sets HTTP headers, which will be sent with the request. If the parameter is NULL, default headers will be sent. The headers will be not sent if the string is empty.

The default headers are the following (the version number in the User-Agent header may be different) :

```
Accept-Charset: utf-8, iso-8859-1, iso-10646-ucs-2\r\n
Accept: application/vnd.wap.wmlc,
        application/vnd.wap.wmlscriptc,
        image/vnd.wap.wbmp,
        image/gif\r\n
User-Agent: WAP-Stack-Client/2.6.1\r\n
Content-Length: < content_size >
```

RETURN VALUE

On success zero is returned. On failure the returned value ranging from 1 to 7 indicates the parameter number with incorrect value.

ERRORS

wps_error_not_object — The given `session` value is not a WAP Stack session handle.

wps_error_network — (HTTP/HTTPS mode) Can't open or bind TCP socket; can't connect to remote host or proxy. Please verify: proxy address/port; server hostname in URL

wps_error_ssl — (HTTPS mode) Can't connect to remote host using SSL; can't open SSL tunnel through the proxy.

wps_error_system — System error occurred.

wps_error_timedout — Timeout while attempting request. Please see wps_get_prm and wps_set_prm 3.3.1 description for more explanations.

SEE ALSO

wps_abort 3.7.3, wps_set_prm 3.3.1

NOTES

The HTTP headers must be well-formed according to the RFC-2616 2. The WAP Stack client shall not pass Content-Type and Content-Length HTTP headers in the `headers` parameter.

3.7.3 wps_abort

NAME

wps_abort — Abort request.

SYNOPSIS

```
#include "wps_e2.h"

int wps_abort(t_hreq req);
```

DESCRIPTION

The wps_abort function terminates an active request identified by the `req` parameter.

RETURN VALUE

On success zero is returned. Another value is an error code.

ERRORS

wps_error_not_object — The given `hreq` value is not a WAP Stack request handle.

wps_error_system — System error occurred.

3.8 Sending requested data

3.8.1 wps_reply

NAME

wps_reply — Send requested data.

SYNOPSIS

```
#include "wps_e2.h"

int wps_reply(t_hreq hreq, const struct t_http_reply_data *rdata);
```

DESCRIPTION

When the WAP HTTP Server Stack receives an HTTP request, it opens a new WP-HTTP session and calls the func_id_http_req event. When the WAP Stack client has processed the event, it can send a reply to the received request using the wps_reply function.

The received request is identified by `hreq` handle, which is given in the wps_id_http_req event.

The `rdata` points to the structure that will receive reply data. The structure is defined in the file "wps_types.h":

```
struct t_http_reply_data
{
    int status;
    const char *desc;
    const char *hdrs;
    const void *data;
    size_t data_size;
    const char *ct;
};
```

The members of the structure are similar to the wps_post 3.7.2 function parameters:

`status` — HTTP status code as defined in the RFC-2616 2;
`desc` — status description, if NULL default description from the RFC-2616 2 will be used;
`hdrs` — HTTP headers to be sent;
`data` — pointer to the data to be sent;
`data_size` — data size;
`ct` — type of data content.

RETURN VALUE

On success zero is returned. On failure the returned value 2 indicates that the pointer rdata is NULL or some of the structure members has a wrong value. Any other value is an error code.

ERRORS

wps_error_not_object — The given `hreq` value is not a WAP Stack request handle.

wps_error_system — System error occurred.

NOTES

The HTTP headers must be well-formed according to the RFC-2616 2. The WAP Stack client shall not pass Content-Type and Content-Length HTTP headers in the `headers` parameter.

The wps_reply function shall be called inside the func_id_http_req event handler or somewhere else before the handler is returned. In other cases the wps_reply returns the error wps_error_not_object.

A Appendix

A.1 List of server certificate validation codes

An exhaustive list of the SSL error codes and messages is shown below, this also includes the name of the error code as defined in the SSL header file x509_vfy.h.

0 X509_V_OK
ok
The operation was successful.

2 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT
Unable to get issuer certificate
The issuer certificate could not be found: this occurs if the issuer certificate of an untrusted certificate cannot be found.

4 X509_V_ERR_UNABLE_TO_DECRYPT_CERT_SIGNATURE
Unable to decrypt certificate's signature
The certificate signature could not be decrypted. This means that the actual signature value could not be determined rather than it not matching the expected value, this is only meaningful for RSA keys.

6 X509_V_ERR_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY
Unable to decode issuer public key
the public key in the certificate SubjectPublicKeyInfo could not be read.

7 X509_V_ERR_CERT_SIGNATURE_FAILURE
Certificate signature failure
The signature of the certificate is invalid.

9 X509_V_ERR_CERT_NOT_YET_VALID
Certificate is not yet valid
The certificate is not yet valid: the notBefore date is after the current time.

10 X509_V_ERR_CERT_HAS_EXPIRED
Certificate has expired
The certificate has expired: that is the notAfter date is before the current time.

13 X509_V_ERR_ERROR_IN_CERT_NOT_BEFORE_FIELD
Format error in certificate's notBefore field
The certificate notBefore field contains an invalid time.

14 X509_V_ERR_ERROR_IN_CERT_NOT_AFTER_FIELD
Format error in certificate's notAfter field
The certificate notAfter field contains an invalid time.

17 X509_V_ERR_OUT_OF_MEM
Out of memory
An error occurred trying to allocate memory. This should never happen.

18 X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT
Self signed certificate

The passed certificate is self signed and the same certificate cannot be found in the list of trusted certificates.

19 X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN

Self signed certificate in certificate chain

The certificate chain could be built up using the untrusted certificates but the root could not be found locally.

20 X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY

Unable to get local issuer certificate

The issuer certificate of a locally looked up certificate could not be found. This normally means the list of trusted certificates is not complete.

21 X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE

Unable to verify the first certificate

No signatures could be verified because the chain contains only one certificate and it is not self signed.

24 X509_V_ERR_INVALID_CA

Invalid CA certificate

A CA certificate is invalid. Either it is not a CA or its extensions are not consistent with the supplied purpose.

25 X509_V_ERR_PATH_LENGTH_EXCEEDED

Path length constraint exceeded The basicConstraints pathlength parameter has been exceeded.

26 X509_V_ERR_INVALID_PURPOSE

Unsupported certificate purpose

The supplied certificate cannot be used for the specified purpose.

27 X509_V_ERR_CERT_UNTRUSTED

Certificate not trusted

The root CA is not marked as trusted for the specified purpose.

28 X509_V_ERR_CERT_REJECTED

Certificate rejected

The root CA is marked to reject the specified purpose.

29 X509_V_ERR_SUBJECT_ISSUER_MISMATCH

Subject issuer mismatch

The current candidate issuer certificate was rejected because its subject name did not match the issuer name of the current certificate.

30 X509_V_ERR_AKID_SKID_MISMATCH

Authority and subject key identifier mismatch

The current candidate issuer certificate was rejected because its subject key identifier was present and did not match the authority key identifier current certificate.

31 X509_V_ERR_AKID_ISSUER_SERIAL_MISMATCH

Authority and issuer serial number mismatch

The current candidate issuer certificate was rejected because its issuer name and serial number was present and did not match the authority key identifier of the current certificate.

32 X509_V_ERR_KEYUSAGE_NO_CERTSIGN

Key usage does not include certificate signing

The current candidate issuer certificate was rejected because its keyUsage extension does not permit certificate signing.

Index