

Winwap Technologies Oy

## **Client WAP Stack ActiveX component**

Application Programming Interface



WAP stack version 2.6

WAP specification version: 2.0

Document dated: 31-Dec-2005



## Notice of Confidentiality

This document contains proprietary and confidential information that belongs to Winwap Technologies Oy.

The recipient agrees to maintain this information in confidence and to not reproduce or otherwise disclose this information to any person outside of the group directly responsible for the evaluation of the content.

## Revision history

Date	Author	Description
15-Nov-2004	S Markelov	Initial version of the WAP Stack ActiveX API specification.

## 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.
- Basic knowledge in ActiveX technique.



# Contents

<b>1</b>	<b>Normative references</b>	<b>3</b>
<b>2</b>	<b>Getting started</b>	<b>4</b>
2.1	What is the WAP Stack ActiveX?	4
2.2	System requirements	5
2.3	Installation	6
2.4	Simple program for Microsoft Visual Basic 6.0	7
2.5	Simple program for Microsoft Visual C++ 6.0	9
2.6	Simple program for Microsoft .NET/C#	13
2.7	Simple program for Borland Delphi 7.0	15
<b>3</b>	<b>API specification</b>	<b>17</b>
3.1	Declarations	17
3.2	Session establishing	18
3.2.1	Properties	18
3.2.2	Methods	22
3.2.3	Events	24
3.3	Requesting	25
3.3.1	Methods	25
3.3.2	Events	28
3.4	WAP specific events	32
3.5	Tuning	34
3.5.1	Properties	34
3.5.2	Methods	41
3.6	Diagnostic	43
3.6.1	Last error code	43
3.6.2	Logging	44
<b>A</b>	<b>Appendix</b>	<b>45</b>
A.1	List of error codes	45
A.2	List of server certificate validation codes	46



# 1 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/>.

## 2 Getting started

### 2.1 What is the WAP Stack ActiveX?

The WAP Stack ActiveX is an ActiveX component based on the WAP Stack SDK, the complete library for using WAP connections. The WAP Stack ActiveX provides both WSP/WTP (WAP Gateway) and WP-HTTP (direct and HTTP proxy) connections, and is quickly implemented into your own software application.

By using the WAP Stack ActiveX from your own application you get full WAP connection functionality from within your own software. An extensive set of methods, functions and events provides full control over the WAP connections so you can build your own browser, testing, measuring or assurance applications.

It is a quick and simple process to begin using the WAP Stack ActiveX. The WAP Stack ActiveX comes with source code samples and API documentation that show exactly how to use it so that you can get quickly started even without having much knowledge about WAP. You can create a wide array of tools for automatically or manually opening WAP connections and retrieving data. All features are well documented and the methods, functions and events are easy to use.

## 2.2 System requirements

- OS Microsoft Windows 98/ME, Microsoft Windows NT 4/2000, Microsoft Windows XP/2003.
- Microsoft CAPICOM 2.0 or later — COM interface to Microsoft CryptoAPI. It is a new security technology from Microsoft that allows Microsoft Visual Basic, Visual Basic Script, ASP, and C++ programmers to easily incorporate digital signing and encryption into their application. CAPICOM is available today for download from the MSDN Web site. CAPICOM is delivered with the Platform SDK and is redistributable free of charge. The CAPICOM can be downloaded from the following URL:

<http://www.microsoft.com/msdownload/platformsdk/setuplauncher.asp>

To register CAPICOM.dll, at the command prompt, change directories to the directory where CAPICOM.dll is stored, and enter the command: `regsvr32 CAPICOM.dll`.

- For both CAPICOM 1.0 and CAPICOM 2.0, Microsoft Internet Explorer 5 or later is required.
- OpenSSL libraries — the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The OpenSSL libraries can be download as an installation package from the following official OpenSSL site:

<http://www.openssl.org/>

If the OpenSSL libraries are not installed in the system, the HTTPS protocol will not be available.



## 2.3 Installation

Simply run the installation program setup.exe and follow the installation procedure. It will copy needed files and register the WAP Stack ActiveX type library in the system.

## 2.4 Simple program for Microsoft Visual Basic 6.0

1. Run the Microsoft Visual Basic 6.0 and create new "Standard EXE" project.
2. Add the WAP Stack ActiveX component into the project: select the menu command Project | Components, find the item "WPSClient 1.0 Type Library" in the list of the available ActiveX controls and click on the check box, press the "OK" button. The component shall appear on the Toolbox.
3. Select the WAP Stack ActiveX component on the Toolbox and place it on the form.
4. Place three labels, four text boxes, one button and change its names and captions as it shown in the Fig. 1 on this page.

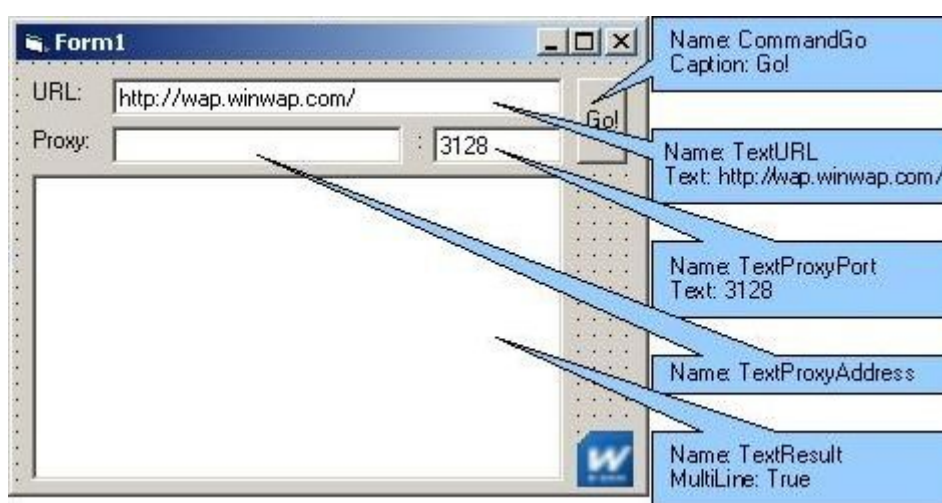


Figure 1: Visual Basic form designer

5. Add the next event handlers for events "Connect", "Reply" of the object WPS\_Client1 and for event "Click" of the button CommandGo!:

```
Private Sub CommandGo_Click()
    ' Clear previous output
    TextResult.Text = ""

    ' Set connection/session type
    WPS_Client1.Mode = WPS_MODE_HTTP

    ' Set proxy settings
    WPS_Client1.ProxyAddress = TextProxyAddress.Text
    WPS_Client1.ProxyPort = TextProxyPort.Text

    Call WPS_Client1.Connect("")
End Sub

Private Sub WPS_Client1_Connect( _
    ByVal code As WPSClientCtl.TxWPS_ConnectCode, _
    ByVal description As String)
```

```
TextResult = "Connected" & Chr(13) & Chr(10)

' Load resource
Call WPS_Client1.Get(TextURL.Text, "Accept: text/vnd.wap.wml")
End Sub

Private Sub WPS_Client1_Reply(ByVal hRequest As Long, _
    ByVal data As Variant, ByVal contentType As String, _
    ByVal Headers As String, ByVal status As Long)
TextResult.Text = TextResult.Text & _
    "Received data is of content type " & _
    contentType & Chr(13) & Chr(10) & Chr(13) & Chr(10)
TextResult.Text = TextResult.Text & data
End Sub
```

6. Run the application, input proxy settings if it is needed and press the button "Go!". In the "TextResult" text box you should see the received data as in the Fig. 2 on the current page.

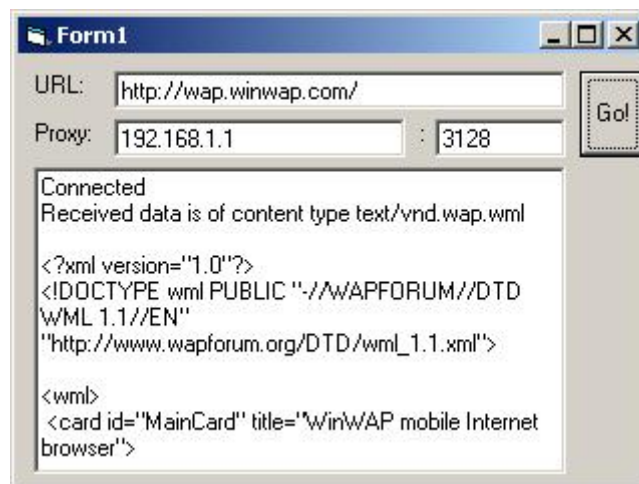


Figure 2: Result output in the Visual Basic application

## 2.5 Simple program for Microsoft Visual C++ 6.0

1. Run the Microsoft Visual C++ and create in the MFC Application wizard new dialog based application with the name "vc".
2. Add the WAP Stack ActiveX component into the project. Select the menu command Project | Add To Project | Components and Controls. Choose the "Registered ActiveX Controls" folder, find the item "WPSClient Class" in the list of the available ActiveX controls, press the "Insert" and two times "OK" buttons. The component shall appear on the "Controls" toolbox. Then press the "Close" button to close the "Components and Controls Gallery" dialog.
3. Select the WAP Stack ActiveX component on the "Controls" toolbox and place it on the dialog.
4. Place three static texts, four edit boxes, one button and change its identifiers, captions and styles as it shown in the Fig. 3 on this page.

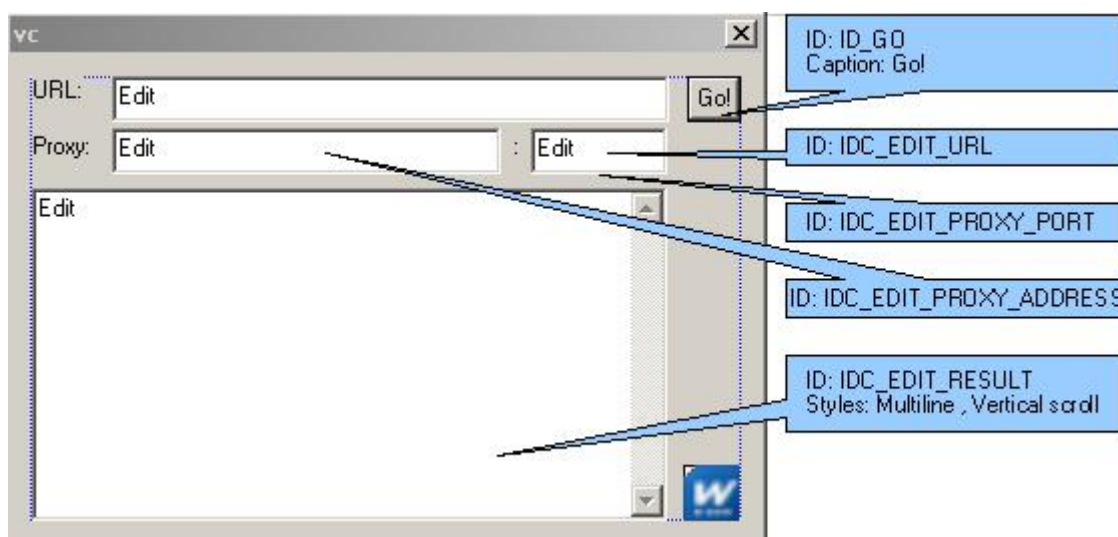


Figure 3: Visual C++ dialog designer

5. Open the MFC ClassWizard (Ctrl+W) and associate the controls with the variables as shown in the Fig. 4 on the next page.
6. Initialize the proxy port and default URL in the dialog constructor:

```
m_proxy_port = _T("3128");
m_url = _T("http://wap.winwap.com/");
```

7. Add the next event handlers for the message BN\_CLICKED of the button m\_gofor and for events "Connect", "Reply" of the object m\_wps\_client:

```
void CVcDlg::OnGo()
{
    const int WPS_MODE_HTTP = 10;
    const int WPS_STATE_NULL = 0;
```

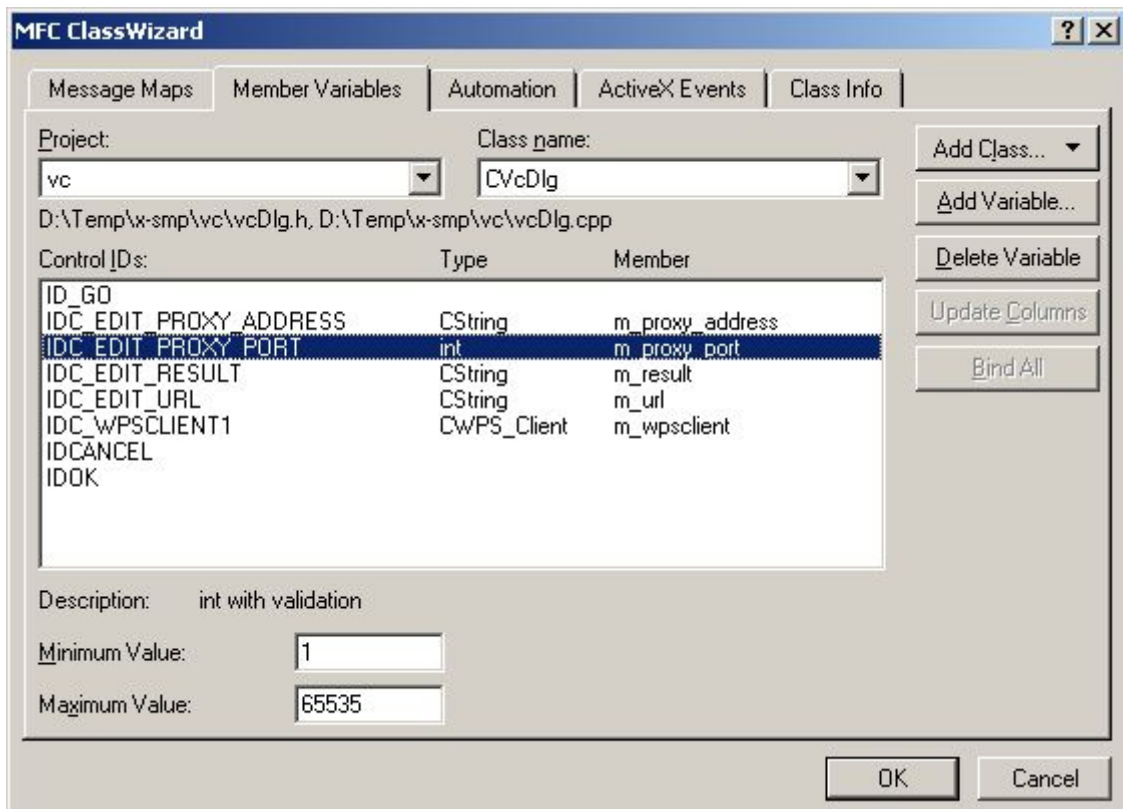


Figure 4: Visual C++ MFC ClassWizard

```
UpdateData (TRUE) ;

// Clear previous output
m_result = "";
UpdateData (FALSE) ;

// Set connection/session type
m_wpsclient.SetMode (WPS_MODE_HTTP) ;

// Set proxy settings
m_wpsclient.SetProxyAddress (m_proxy_address) ;
m_wpsclient.SetProxyPort (m_proxy_port) ;

if (WPS_STATE_NULL == m_wpsclient.GetState ())
    m_wpsclient.Connect ("") ;
else
    OnConnectWpsclient1 (0, "");
}

void CVcDlg::OnConnectWpsclient1 (long code, LPCTSTR description)
{
    const int WPS_CC_OK = 0 ;

    UpdateData (TRUE) ;
```

```

m_result = (WPS_CC_OK == code) ? "Connected\r\n"
        : "Couldn't connect\r\n";

UpdateData(FALSE);

// Load resource
if (WPS_CC_OK == code)
    m_wpsclient.Get(m_url, "Accept: text/vnd.wap.wml");
}

void CVcDlg::OnReplyWpsclient1(long hRequest, const VARIANT FAR& data,
    LPCTSTR contentType, LPCTSTR Headers, long status)
{
    UpdateData(TRUE);

    // Get array bounds.
    LONG LowBound, UpperBound;
    if (FAILED(SafeArrayGetLBound(V_ARRAY(&data), 1, &LowBound)))
        return;
    if (FAILED(SafeArrayGetUBound(V_ARRAY(&data), 1, &UpperBound)))
        return;

    // Get a pointer to the elements of the array.
    const char *d;
    if (FAILED(SafeArrayAccessData(V_ARRAY(&data), (void **) &d)))
        return;

    LONG size = UpperBound - LowBound + 1;
    CString s('\0', size); // make string of given size
    TCHAR *buf = s.LockBuffer();

    for (int i = 0; i < size; ++i)
        buf[i] = d[i];

    s.UnlockBuffer();

    m_result += "Received data is of content type ";
    m_result += contentType;
    m_result += "\r\n\r\n";
    m_result += s;

    UpdateData(FALSE);
}

```

8. Run the application, input proxy settings if it is needed and press the button "Go!". In the m\_result edit box you should see the received data as in the Fig. 5 on the following page.

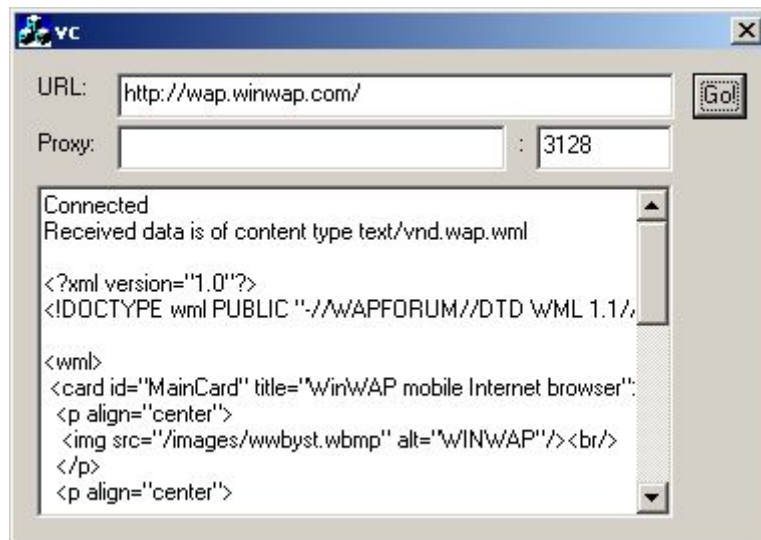


Figure 5: Result output in the Visual C++ application

## 2.6 Simple program for Microsoft .NET/C#

1. Run the Microsoft Visual Studio .NET and create new Visual C# Project of the type "Windows Application".
2. Add the WAP Stack ActiveX component into the project: select the menu command Project | Add reference, find in the list of the available ActiveX controls on the page "COM" the item "WPSClient 1.0 Type Library", then click doubly on the item and press the "OK" button. The WAP Stack ActiveX component shall appear in the "Components" page of the Toolbox.
3. Select the WAP Stack ActiveX component on the Toolbox and place it on the form.
4. Place three labels, four text boxes, one button and change its names and captions as it shown in the Fig. 6 on the current page.

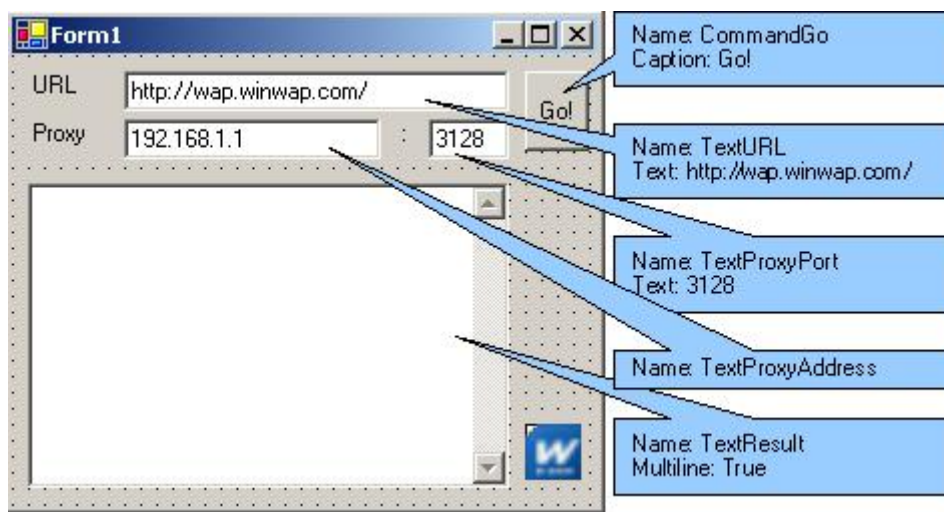


Figure 6: Visual Studio .NET form designer

5. Add the next event handlers for events "Connect", "Reply" of the object WPS\_Client1 and for event "Click" of the button CommandGo:

```
private void CommandGo_Click(object sender, System.EventArgs e)
{
    // Clear previous output
    TextResult.Text = "";

    // Set connection/session type
    axWPS_Client1.Mode = WPSClient.TxWPS_Mode.WPS_MODE_HTTP;

    // Set proxy settings
    axWPS_Client1.ProxyAddress = TextProxyAddress.Text;
    axWPS_Client1.ProxyPort = Int32.Parse(TextProxyPort.Text);

    // Perform connection
    axWPS_Client1.Connect("");
}
```

```
private void axWPS_Client1_ConnectEvent(object sender,
    AxWPSClient.DWPS_ClientEvents_ConnectEvent e)
{
    TextResult.Text = "Connected" + "\r\n";

    // Load resource
    axWPS_Client1.Get(TextURL.Text, "Accept: text/vnd.wap.wml");
}

private void axWPS_Client1_Reply(object sender,
    AxWPSClient.DWPS_ClientEvents_ReplyEvent e)
{
    TextResult.Text += "Received data is of content type "
        + e.contentType + "\r\n\r\n";

    // Reply event gives the requested data in byte-format,
    // as a server does reply.
    // To use it as a string, we shell convert it as necessary.
    Byte[] theData = (Byte[]) e.data;
    String strData = "";
    for (int idx = 0; idx < theData.Length; idx++)
        strData += (char) theData[idx];
    TextResult.Text += strData;
}
}
```

6. Run the application, input proxy settings if it is needed and press the button "Go!". In the "TextResult" text box you should see the received data as in the Fig. 7 on this page.

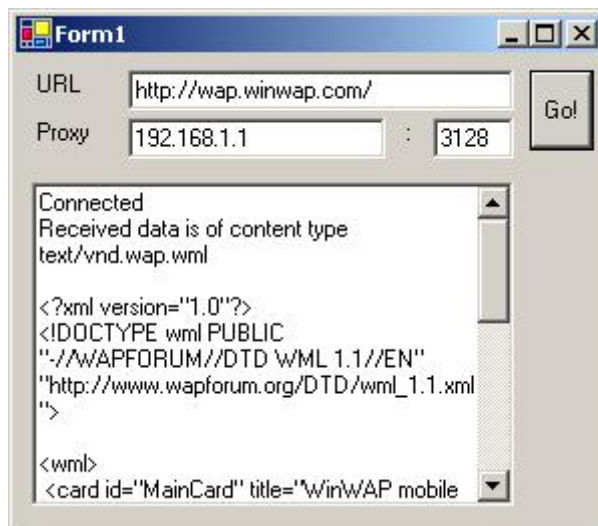


Figure 7: Result output in the Visual Studio .NET application

## 2.7 Simple program for Borland Delphi 7.0

1. Run the Borland Delphi 7.0.
2. Add the WAP Stack ActiveX component into the Component Palette: select the menu command Component | Import ActiveX Control, find the item "WPSClient 1.0 Type Library" in the list of the available ActiveX controls and then press the "Install..." and then "OK" buttons. The WAP Stack ActiveX component shall appear in the "ActiveX" page of the Component Palette.
3. Select the WAP Stack ActiveX component on the Component Palette and place it on the form.
4. Place three labels, three edit, one memo controls, one button and change its names and captions as it shown in the Fig. 8 on the current page.

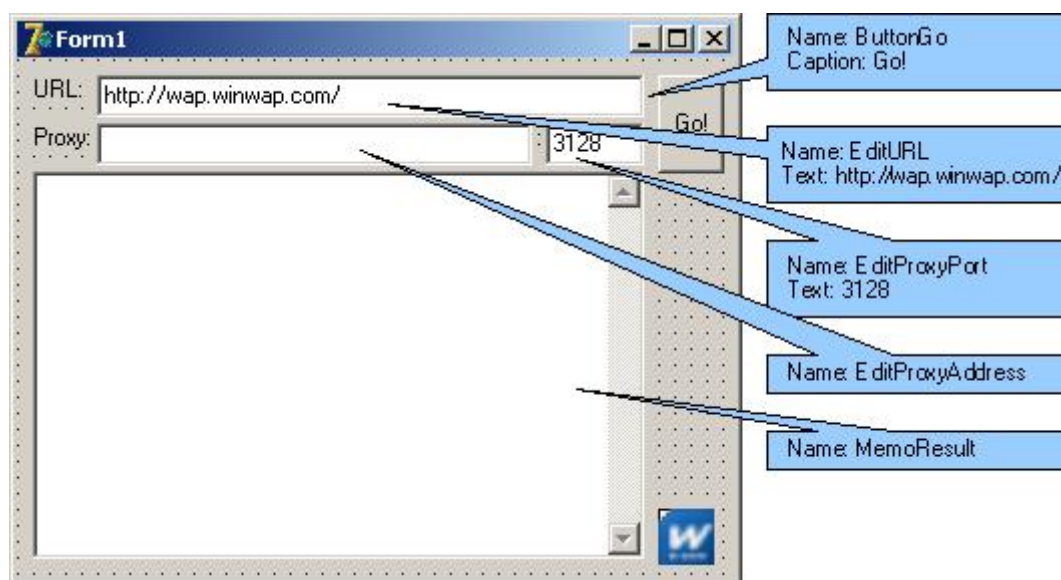


Figure 8: Delphi form designer

5. Add the next event handlers for events "Connect", "Reply" of the object WPS\_Client1 and for event "Click" of the button CommandGo:

```
procedure TForm1.ButtonGoClick(Sender: TObject);  
begin  
    // Clear previous output  
    MemoResult.Lines.Text := '';  
  
    // Set connection/session type  
    WPS_Client1.Mode := WPS_MODE_HTTP;  
  
    // Set proxy settings  
    WPS_Client1.ProxyAddress := EditProxyAddress.Text;  
    WPS_Client1.ProxyPort := StrToInt(EditProxyPort.Text);  
    WPS_Client1.Connect('');  
end;
```

```

procedure TForm1.WPS_Client1Connect (ASender: TObject; code: TOleEnum;
    const description: WideString);
begin
    MemoResult.Lines.Add('Connected');

    // Load resource
    WPS_Client1.Get(EditURL.Text, 'Accept: text/vnd.wap.wml');
end;

procedure TForm1.WPS_Client1Reply (ASender: TObject; hRequest: Integer;
    data: OleVariant; const contentType, Headers: WideString;
    status: Integer);
    type a = Array of Byte;
begin
    MemoResult.Lines.Add('Received data is of type '
        + contentType);
    MemoResult.Lines.Add('');
    MemoResult.Lines.Add(PChar(a(data)));
end;
    
```

- Run the application, input proxy settings if it is needed and press the button "Go!". In the memo control you should see the received data as in the Fig. 9 on this page.

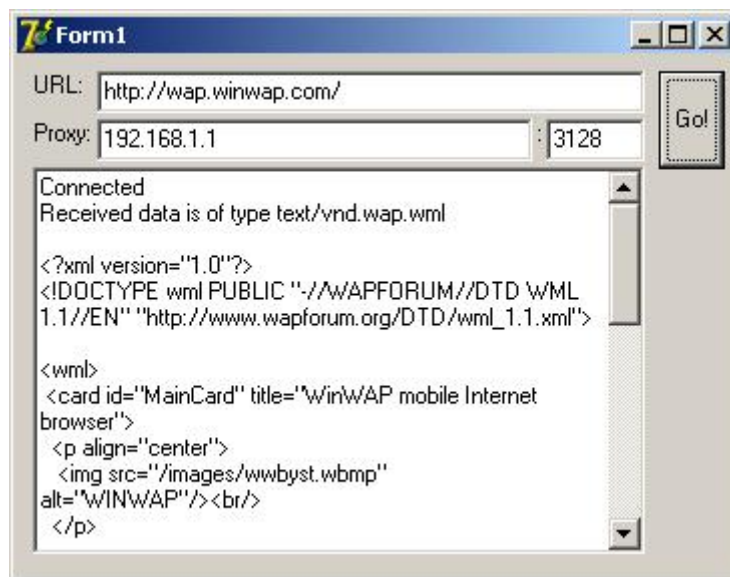


Figure 9: Result output in the Delphi application



## 3 API specification

### 3.1 Declarations

All interfaces and type declarations are available in the following IDL-file:

WPSClientX.idl — Interfaces and type definitions.



## 3.2 Session establishing

Firstly the WAP Stack ActiveX component needs to be initialized. This section describes the properties and the methods that are used for this purpose.

### 3.2.1 Properties

#### NAME

Mode — connection mode; session type

#### SYNOPSIS

```
HRESULT Mode([out, retval] TxWPS_Mode *pVal);  
HRESULT Mode([in] TxWPS_Mode newVal);
```

#### DESCRIPTION

The **Mode** property sets the session type that will be used for all requests of the opened WAP Stack.

The following constants can be used as values for **Mode** property:

<b>WPS_MODE_CL</b>	WAP connectionless mode;
<b>WPS_MODE_CO</b>	WAP connection-oriented mode;
<b>WPS_MODE_SCL</b>	WAP secure connectionless mode;
<b>WPS_MODE_SCO</b>	WAP secure connectionless mode;
<b>WPS_MODE_HTTP</b>	WP-HTTP mode;
<b>WPS_MODE_HTTPS</b>	WP-HTTP/SSL mode.

## NAME

ProxyAddress, ProxyPort — WAP Gateway or HTTP Proxy address settings

## SYNOPSIS

```
HRESULT ProxyAddress([out, retval] BSTR *pVal);
HRESULT ProxyAddress([in] BSTR newVal);

HRESULT ProxyPort([out, retval] long *pVal);
HRESULT ProxyPort([in] long newVal);
```

## DESCRIPTION

The [ProxyAddress](#) and [ProxyPort](#) properties are interpreted for different modes as described below:

[WPS\\_MODE\\_HTTP](#), [WPS\\_MODE\\_HTTPS](#): the [ProxyAddress](#) is a HTTP Proxy address, which can be a host name. In case if the parameter value is empty, the WAP Stack ActiveX component will use direct connections to requested hosts. The [ProxyPort](#) is a HTTP Proxy port number.

Other modes: the [ProxyAddress](#) is a WAP Gateway address. It can be a host name but must not be empty. The [ProxyPort](#) is a WAP Gateway port number.

## NOTES

[ProxyAddress](#) and [ProxyPort](#) properties shall be updated to desired values before first call of [Connect](#) function.

To change current [ProxyAddress](#), [ProxyPort](#) properties, the client should call [Disconnect](#) [3.2.2, p. 22], set desired HTTP proxy or WAP gateway settings and then call [Connect](#) [3.2.2, p. 22].



## NAME

LocalAddress — accepted local address/network interface

## SYNOPSIS

```
HRESULT LocalAddress([out, retval] BSTR *pVal);  
HRESULT LocalAddress([in] BSTR newVal);
```

## DESCRIPTION

The [LocalAddress](#) property is used to bind the WAP Stack to a custom system network address. It can be used if the WAP Stack ActiveX component does not have to 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 [LocalAddress](#) property is an IP address of the available system network interface. The WAP Stack will be bound to all available system network interfaces when the parameter is empty or equals "0.0.0.0". Outgoing requests are sent accordingly to the system IP routing table and requests may be sent from a different IP address.



## NAME

State — current session state of the WAP Stack ActiveX component

## SYNOPSIS

```
HRESULT State([out, retval] TxWPS_ConnectState *pVal);
```

## DESCRIPTION

The `State` property allows to retrieve the current session state of the WAP Stack ActiveX component.

The following session states can be determined:

- `WPS_STATE_NULL` — not connected;
- `WPS_STATE_CONNECTING` — connecting, in progress;
- `WPS_STATE_CONNECTED` — connected, a session is established.

## 3.2.2 Methods

### NAME

Connect — create/establish new session  
Disconnect — disconnect/terminate the session

### SYNOPSIS

```
HRESULT Connect(  
    [in, defaultvalue("")] BSTR headers,  
    [out, retval] VARIANT_BOOL *pRet);  
HRESULT Disconnect([out, retval] VARIANT_BOOL *pRet);
```

### DESCRIPTION

The [Connect](#) method initializes the WAP Stack ActiveX component and establishes asynchronously new session.

The [Disconnect](#) method aborts all active requests and terminates the established session.

The [Connect](#) method initializes internal structures, assigns WAP Gateway or HTTP Proxy address, sends connection request to the WAP Gateway in case of connection-oriented (CO), secure connectionless (SCL) or secure connection-oriented (SCO) mode and then exits. The container (client) of the WAP Stack ActiveX component shall wait for the [Connect](#) [3.2.3, p. 24] event provided by the dispatch interface [DWPS\\_Client](#) that shall signal about successful or failed connection and for [Redirect](#) [3.2.3, p. 24] event, that shall signal about redirection request to another WAP Gateway address.

All active requests will be aborted and the previously established session will be terminated.

The [Headers](#) parameter is needed for setting some specific HTTP headers that are required for a successful connection to a WAP Gateway or HTTP proxy.

### RETURN VALUE

Upon successful completion [Connect](#) and [Disconnect](#) methods return [VARIANT\\_TRUE](#). Otherwise, [VARIANT\\_FALSE](#) is returned and the method [LastError](#) [3.6.1, p. 43] can be used to indicate the error.

### NOTES

Below is a list of some known HTTP headers that might be required by a WAP Gateway for successful connecting:

```
Proxy-Authorization  
Profile  
User-Agent
```

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

In case of WAP connection-oriented (**CO**) or secure connection-oriented (**SCO**) modes these session headers are used once during session establishment and generally for authorization. In case of WAP connectionless (**CL**) or secure connectionless (**SCL**) modes these session headers are not used during session establishment, since connectionless modes are not session oriented, but in any case the **Connect** method shall be used for the WAP Stack ActiveX component initialization. In case of **HTTP/HTTPS** modes, new TCP session for each request is established. Therefore these session headers are appended to the HTTP headers that are given in the **Get** [3.3.1, p. 25] or **Post** [3.3.1, p. 26] methods. Therefore the developer of the WAP Stack ActiveX client must be careful and not pass HTTP headers in these methods that are already passed through the **Headers** parameter of the **Connect** method. 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.

## SEE ALSO

Properties **Mode** [3.2.1, p. 18], **State** [3.2.1, p. 21], **ProxyAddress** [3.2.1, p. 19], **HTTPSend-Timeout** [3.5.1, p. 40]. **HTTPWaitTimeout** [3.5.1, p. 40].

Events **Event!Connect** [??, p. ??] **Event!Redirect**.



### 3.2.3 Events

#### NAME

Connect — signals about successful or failed connection  
Redirect — signals about redirection request during session establishing

#### SYNOPSIS

```
HRESULT Connect([in] long code, [in] BSTR description);  
HRESULT Redirect([in] BSTR host, [in] long port);
```

#### DESCRIPTION

The **Connect** event occurs when the session establishing is completed. Upon successful session establishing `code` parameter contains **WPS\_CC\_OK**. Otherwise, one of the values below is returned and the `description` parameter contains an error code description:

- WPS\_CC\_END** — the session was terminated by a WAP Gateway;
- WPS\_CC\_FAIL** — could not establish new session.

The **Redirect** event occurs when a WAP Gateway terminates the session negotiation and provides with another WAP Gateway address to continue session negotiation.

The **Redirect** event can occur in the WAP modes only. When it occurs the WAP Stack ActiveX container shall call the **Disconnect** [3.2.2, p. 22] method. Then the container can assign new WAP Gateway address and port given in the `host` and `port` parameters and call the **Connect** [3.2.2, p. 22] event to establish a session with new parameters.

## 3.3 Requesting

After the WAP Stack ActiveX component is successfully initialized and a session is successfully established, the WAP Stack ActiveX client can request and download resources from the internet.

### 3.3.1 Methods

#### NAME

Get — Request a resource through the GET method

#### SYNOPSIS

```
HRESULT Get (  
    [in] BSTR uri,  
    [in, defaultvalue("Accept: */*")] BSTR headers,  
    [out, retval] long *hRequest);
```

#### DESCRIPTION

The `Get` method sends a GET request to a WAP Gateway or HTTP Proxy. The function is asynchronous (doesn't wait for a server result) and therefore the WAP Stack ActiveX client shall wait for the `Reply` [3.3.2, p. 28] event.

The `uri` parameter identifies a requested resource.

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

#### RETURN VALUE

Upon successful completion `Get` method returns identifier of new request. Otherwise, 0 is returned and the method `LastError` [3.6.1, p. 43] can be used to indicate the error.

#### NOTES

The HTTP headers must be well-formed according to the [RFC-2616](#) [1].

## NAME

Post — Request a resource through the POST method

## SYNOPSIS

```
HRESULT Post(  
    [in] BSTR uri,  
    [in, defaultValue("Accept: */*")] BSTR headers,  
    [in] VARIANT data,  
    [in, defaultValue("application/x-wwwform-urlencoded")]  
        BSTR contentType,  
    [out, retval] long *hRequest);
```

## DESCRIPTION

The `Post` method sends a POST request to a WAP Gateway or HTTP Proxy, in other words it posts data from an application. The function is asynchronous (doesn't wait for a server result) and therefore the WAP Stack ActiveX client shall wait for the `Reply` [3.3.2, p. 28] event.

The `uri` parameter identifies a requested resource.

The `data` parameter is the data that will be posted.

The `contentType` parameter is a type of data. If the value is not set the "application/x-wwwform-urlencoded" value would be used.

## RETURN VALUE

Upon successful completion `Post` method returns identifier of new request. Otherwise, 0 is returned and the method `LastError` [3.6.1, p. 43] can be used to indicate the error.

## NOTES

Please note that the post data specified by the `data` shall be passed as a SAFEARRAY structure. The VARIANT should be of type VT\_ARRAY and point to a SAFEARRAY. The SAFEARRAY should be of element type VT\_UI1, dimension one, and have an element count equal to the number of bytes of post data.



## NAME

Abort — abort request

## SYNOPSIS

```
HRESULT Abort(  
    [in] long hRequest,  
    [out, retval] VARIANT_BOOL *pRes);
```

## DESCRIPTION

The [Abort](#) method terminates an active request identified by the `hRequest` parameter.

## RETURN VALUE

Upon successful completion [Abort](#) method returns value `VARIANT_TRUE`. Otherwise, `VARIANT_FALSE` is returned and the method [LastError](#) [3.6.1, p.43] can be used to indicate the error.

### 3.3.2 Events

#### NAME

Reply — reply from the server (received data)

#### SYNOPSIS

```
HRESULT Reply(  
    [in] long hRequest,  
    [in] VARIANT data,  
    [in] BSTR contentType,  
    [in] BSTR headers,  
    [in] long status);
```

#### DESCRIPTION

The **Reply** event occurs when a response from the server is received. The `hRequest` parameter identifies the request to that the received data is corresponding. It is the returned value of the **Get** [3.3.1, p. 25] or **Post** [3.3.1, p. 26] method.

The `data` parameter is an array with received data.

The `contentType` parameter indicates the type of received data content (<http://www.wapforum.org/wina>).

The `headers` parameter indicates the received HTTP headers.

The `status` parameter indicates the HTTP status code encoded according to the table in WAP-230-WSP-2001-07-5-a, Appendix A.

#### NOTES

Please note that the received data specified is a SAFEARRAY structure. The VARIANT is of type VT\_ARRAY and points to a SAFEARRAY. The SAFEARRAY is of element type VT\_UI1, dimension one, and has an element count equal to the number of bytes of the received data.

## NAME

Certificate — server certificate

## SYNOPSIS

```
HRESULT Certificate(  
    [in] long hRequest,  
    [in] ICertificate2 *cert,  
    [in] long failReason,  
    [in, out] VARIANT_BOOL *accepted);
```

## DESCRIPTION

The [Certificate](#) event can occur in HTTP/HTTPS modes only. It occurs when the server sends its certificate in order to be authenticated by the client.

The `hRequest` parameter identifies the request to that the received certificates is corresponding. It is the returned value of the [Get](#) [3.3.1, p. 25] or [Post](#) [3.3.1, p. 26] method.

The `cert` parameter indicates the certificate object.

Upon successful completion of the validation procedure in WAP Stack ActiveX the `accepted` parameter contains `VARIANT_TRUE`. Otherwise, it contains `VARIANT_FALSE` and the `failReason` parameter can be used to indicate the reason why is the validation procedure failed. Please see the section [A.2](#) on page [46](#).

The WAP Stack ActiveX client can set `VARIANT_TRUE` to the `accepted` in order to continue the request ignoring the certificate validation result. Or it can set `VARIANT_FALSE` to the `accepted` in order to abort the request.

The WAP Stack ActiveX client can upload trusted certificates into WAP Stack ActiveX using the [AddTrustedCertificate](#) [3.5.2, p. 41] method in order to allow the WAP Stack ActiveX component to do full validation of server certificates.

## SEE ALSO

[AddTrustedCertificate](#) [3.5.2, p. 41] method.

## NAME

CertificateRequest — HTTPS server requests client certificate

## SYNOPSIS

```
HRESULT CertificateRequest(  
    [in] long hRequest,  
    [in] TxWPS_CertType certType,  
    [in, out] VARIANT_BOOL *send);
```

## DESCRIPTION

The [CertificateRequest](#) event is called when a client certificate is requested by a server and no certificate was yet set for the WAP Stack ActiveX object. 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.

The `hRequest` parameter identifies the request to that the request is corresponding. It is the returned value of the [Get](#) [3.3.1, p. 25] or [Post](#) [3.3.1, p. 26] method.

The `type` parameter is requested client certificate type:

[WPS\\_CERTTYPE\\_X509](#) — X509 certificate.

When the [Certificate](#) event occurs, the client can set its certificate to the [Certificate](#) [3.5.1, p. 34] property, if it is needed, and shall to set the `send` parameter to one from the next values:

[VARIANT\\_FALSE](#) — to terminate SSL handshake;

[VARIANT\\_TRUE](#) — to send assigned certificate and continue SSL handshake,

## SEE ALSO

[Certificate](#) [3.5.1, p. 34] property.

## NAME

Progress — HTTP client progress indicator

## SYNOPSIS

```
HRESULT Progress(  
    [in] long hRequest,  
    [in] TxWPS_ProgressId id,  
    [in] long dataSize,  
    [in] long hdrSize);
```

## DESCRIPTION

The [Progress](#) event is an indicator of WAP HTTP client activity.

The `hRequest` parameter identifies the request to that the received certificates is corresponding. It is the returned value of the [Get](#) [3.3.1, p. 25] or [Post](#) [3.3.1, p. 26] method.

The `id` parameter identifies the last finished action of the WAP HTTP client:

- [WPS\\_PROGRESS\\_CONNECT](#) — connection to the HTTP server is established;
- [WPS\\_PROGRESS\\_SEND](#) — request is partially or fully sent;
- [WPS\\_PROGRESS\\_RECV](#) — response data is partially or fully received.

The `dataSize` parameter is size of the received or sent data.

The `hdrSize` parameter is size of the headers in the received data.

The value of the `dataSize` is applicable for both actions [WPS\\_PROGRESS\\_SEND](#) and [WPS\\_PROGRESS\\_RECV](#). The value of the `hdrSize` is only applicable for the [WPS\\_PROGRESS\\_RECV](#) WAP HTTP client action.

## NOTES

The [Progress](#) event with the indicated [WPS\\_PROGRESS\\_RECV](#) action can be called several times before the event [Reply](#) [3.3.2, p. 28] is called.

The [WPS\\_PROGRESS\\_SEND](#) action indicator can be called several times.

The event [Progress](#) with the indicated [WPS\\_PROGRESS\\_SEND](#) action allows to determine the count of bytes that are already sent as a request to a HTTP Server. The `dataSize` value indicates the size of the sent part of the entire HTTP request including HTTP headers and HTTP body. Please see [HTTPSendTimeout](#) [3.5.1, p. 40] property for additional information.

The event [Progress](#) with the indicated [WPS\\_PROGRESS\\_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 Stack ActiveX and after the required transformations it will be passed through the [Reply](#) [3.3.2, p. 28] event when the whole response has been received. The `dataSize` 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 [Reply](#) [3.3.2, p. 28] event).

The value `hdrSize` 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 `dataSize - hdrSize - 4`.

## 3.4 WAP specific events

### NAME

Push — received push data

### SYNOPSIS

```
HRESULT Push(  
    [in] VARIANT data,  
    [in] BSTR contentType,  
    [in] BSTR headers);
```

### DESCRIPTION

When WAP secure or unsecure connection-oriented session is established, the WAP Stack ActiveX component can receive not requested data pushed by a WAP gateway. This event is very similar to the [Reply \[3.3.2, p. 28\]](#) event, but it is not corresponding to any request.

The `data` parameter is an array with received data.

The `contentType` parameter indicates the type of received data content (<http://www.wapforum.org/wina>).

The `headers` parameter indicates the received HTTP headers.

### NOTES

Please note that the received data specified is a SAFEARRAY structure. The VARIANT is of type VT\_ARRAY and points to a SAFEARRAY. The SAFEARRAY is of element type VT\_UI1, dimension one, and has an element count equal to the number of bytes of the received data.

## NAME

Frame — WAP WTP process indicator.

## SYNOPSIS

```
HRESULT Frame (
    [in] long hRequest,
    [in] long id,
    [in] long size,
    [in] long attr);
```

## DESCRIPTION

The `Frame` event is an indicator of WAP WTP transactions.

The `hRequest` parameter identifies the request to that the received certificates is corresponding. It is the returned value of the `Get` [3.3.1, p. 25] or `Post` [3.3.1, p. 26] method.

The parameters:

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

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

Mask	Value interpretation
<code>WPS_WTP_FRAME_RESEND</code>	0 — first sending, 1 — re-sending;
<code>WPS_WTP_FRAME_LAST</code>	1 — last frame in WTP message;
<code>WPS_WTP_FRAME_GROUP</code>	1 — last frame in WTP SAR group;
<code>WPS_WTP_FRAME_DIR</code>	0 — outgoing frame, 1 — incoming frame;
<code>WPS_WTP_FRAME_SENT</code>	for outgoing frame: 0 — about start of frame sending, 1 — frame has been sent;
<code>WPS_WTP_FRAME_TYPE</code>	WTP PDU type. The attribute can contain the following values that define possible types of the WTP PDU:
<code>WPS_WTP_PDU_TYPE_INVOKE</code>	Invoke WTP PDU;
<code>WPS_WTP_PDU_TYPE_RESULT</code>	Result WTP PDU;
<code>WPS_WTP_PDU_TYPE_ACK</code>	Ack WTP PDU;
<code>WPS_WTP_PDU_TYPE_ABORT</code>	Abort WTP PDU;
<code>WPS_WTP_PDU_TYPE_S_INVOKE</code>	Segmented Invoke WTP PDU;
<code>WPS_WTP_PDU_TYPE_S_RESULT</code>	Segmented Result WTP PDU;
<code>WPS_WTP_PDU_TYPE_NACK</code>	Negative Ack WTP PDU.

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



## 3.5 Tuning

### 3.5.1 Properties

#### NAME

Certificate — client certificate

#### SYNOPSIS

```
HRESULT Certificate([out, retval] ICertificate2 **pVal);  
HRESULT Certificate([in] ICertificate2 *newVal);
```

#### DESCRIPTION

In order to allow a HTTPS server to authenticate a client, the WAP Stack ActiveX client shall assign its x509 certificate and private key encapsulated in [ICertificate2](#) object to the [Certificate](#) property. Please see [CAPICOM](#) documentation.

#### SEE ALSO

[CertificateRequest](#) [3.3.2, p. 30] event.



## NAME

TrustedCertificates — list of trusted certificates.

## SYNOPSIS

```
HRESULT TrustedCertificates([out, retval] ICertificates2 **pVal);
```

## DESCRIPTION

The [TrustedCertificates](#) property allows WAP Stack ActiveX client to check the list of loaded trusted certificates. The result list is encapsulated in [ICertificates2](#) collection of certificates. Please see [CAPICOM](#) documentation.

## SEE ALSO

[AddTrustedCertificate](#) [3.5.2, p. 41] method.

## NAME

`WriteBufferSize` — size of buffer for outgoing data.

`ReadBufferSize` — size of buffer for incoming data.

## SYNOPSIS

```
HRESULT WriteBufferSize([out, retval] long *pVal);
```

```
HRESULT WriteBufferSize([in] long newVal);
```

```
HRESULT ReadBufferSize([out, retval] long *pVal);
```

```
HRESULT ReadBufferSize([in] long newVal);
```

## DESCRIPTION

The `WriteBufferSize` and `ReadBufferSize` properties tune internal WAP WDP buffers for outgoing and incoming data. Value of the properties can be changed at any moment, but this newly assigned values will be first used in `Connect` [3.2.2, p. 22] method.

## NOTES

By default the sizes of the WAP WDP buffers are set to the maximum values supported by the operating system (OS). 1024 bytes of the WAP WDP buffers are reserved for the WAP WTLS layer. WAP WDP buffers for incoming and outgoing data can therefore not be greater than the buffers provided by the OS minus 1024 bytes. The `WriteBufferSize` and `ReadBufferSize` properties accept very large sizes without any errors. After the `Connect` [3.2.2, p. 22] is called, the `WriteBufferSize` and `ReadBufferSize` properties can be used to determine the actual set sizes of the WAP WDP buffers for outgoing and incoming data.

## SEE ALSO

`SARFrameSize` [3.5.1, p. 38] property.



## NAME

SAREnabled — enable/disable WTP Segmentation and re-assembly.

## SYNOPSIS

```
HRESULT SAREnabled([out, retval] VARIANT_BOOL *pVal);  
HRESULT SAREnabled([in] VARIANT_BOOL newVal);
```

## DESCRIPTION

The [SAREnabled](#) property enables or disables WTP Segmentation and re-assembly (SAR). The [VARIANT\\_FALSE](#) value disables SAR and the [VARIANT\\_TRUE](#) value enables SAR. When SAR is enabled and a WAP Gateway supports it, the WAP Stack ActiveX can send data of size about 256 times [SARFrameSize](#) [3.5.1, p. 38].

By default SAR is enabled.

## SEE ALSO

[SARFrameSize](#) [3.5.1, p. 38] property.



## NAME

SARFrameSize — size of WAP WTP SAR Frame.

## SYNOPSIS

```
HRESULT SARFrameSize([out, retval] long *pVal);  
HRESULT SARFrameSize([in] long newVal);
```

## DESCRIPTION

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 outgoing messages. The SAR frame size can not be smaller than 16 bytes or greater than the size of the WDP outgoing buffer.

The default SAR frame size is 1400 bytes.

## SEE ALSO

SAREnabled [3.5.1, p. 37] and WriteBufferSize [3.5.1, p. 36] properties.

## NAME

HeaderEncodingVersion — WSP header encoding version.

## SYNOPSIS

```
HRESULT HeaderEncodingVersion(  
    [out, retval] TxWPS_EncodingVersion *pVal);  
HRESULT HeaderEncodingVersion(  
    [in] TxWPS_EncodingVersion newVal);
```

## DESCRIPTION

Currently the WAP Stack ActiveX accepts the following values:

- WPS\_HDR\_ENCODING\_12 — version 1.2,
- WPS\_HDR\_ENCODING\_13 — version 1.3,
- WPS\_HDR\_ENCODING\_14 — version 1.4.

The default header encoding version is 1.3.

## NAME

HTTPSendTimeout, HTTPWaitTimeout — HTTP request and server reply timeouts.

## SYNOPSIS

```
HRESULT HTTPSendTimeout([out, retval] long *pVal);
HRESULT HTTPSendTimeout([in] long newVal);

HRESULT HTTPWaitTimeout([out, retval] long *pVal);
HRESULT HTTPWaitTimeout([in] long newVal);
```

## DESCRIPTION

The [HTTPSendTimeout](#) and [HTTPWaitTimeout](#) properties 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 [HTTPSendTimeout](#) property, then [Get](#) [3.3.1, p. 25] and [Post](#) [3.3.1, p. 26] methods return null request identifier.

If either 0 or big enough number of milliseconds is assigned to the [HTTPSendTimeout](#) property, then the [Get](#) [3.3.1, p. 25] and [Post](#) [3.3.1, p. 26] methods 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 [HTTPWaitTimeout](#) property, then the [Reply](#) [3.3.2, p. 28] event occurs with 0x48 (HTTP 408 Request Timeout) status code.

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

## 3.5.2 Methods

### NAME

AddTrustedCertificate — add new issuer certificate

### SYNOPSIS

```
HRESULT AddTrustedCertificate(  
    [in] ICertificate2 *cert,  
    [out, retval] VARIANT_BOOL *pRet);
```

### DESCRIPTION

In order to add new issuer certificate and to allow the WAP Stack ActiveX to do full validation of server certificates, the WAP Stack ActiveX client should call the [AddTrustedCertificate](#) method.

The `cert` parameter is the trusted x509 certificate encapsulated in [ICertificate2](#) object. Please see [CAPICOM](#) documentation.

### RETURN VALUE

Upon successful completion [AddTrustedCertificate](#) method returns [VARIANT\\_TRUE](#). Otherwise, [VARIANT\\_FALSE](#) is returned and the method [LastError](#) [[3.6.1](#), p. 43] can be used to indicate the error.

### NOTES

The WAP Stack ActiveX client can only add an issuer certificate to the internal list of trusted certificates. To clear the list of trusted certificates and assign new issuer certificates new WAP Stack ActiveX instance shall be used.

## NAME

SetWTLSCientId — set WTLS client identifier.

## SYNOPSIS

```
HRESULT SetWTLSCientId(  
    [in] TxWPS_WTLSCientIdType type,  
    [in] VARIANT id,  
    [in] long charset,  
    [out, retval] VARIANT_BOOL *pRet);
```

## DESCRIPTION

The `SetWTLSCientId` identifier allows to set the client identifier. The client identifier used to identify the client in a relevant way for the key exchange suite. The server can use this information to fetch a client certificate from its database. The parameter `type` specifies the type of identifier used, and can be assigned to the following values:

<code>WPS_ID_TYPE_NULL</code>	— supply no identity;
<code>WPS_ID_TYPE_TEXT</code>	— textual name with character set;
<code>WPS_ID_TYPE_BINARY</code>	— binary identity;
<code>WPS_ID_TYPE_KEY_HASH_SHA</code>	— SHA-1 hash of the public key;
<code>WPS_ID_TYPE_X509_NAME</code>	— X.509 distinguished name.

The `id` is an identifier. The identifier specified by the `id` parameter shall be passed as a SAFEARRAY structure. The VARIANT should be of type VT\_ARRAY and point to a SAFEARRAY. The SAFEARRAY should be of element type VT\_UI1, dimension one, and have an element count equal to the length of the identifier. The default identifier is a value of binary type: 0x00000000A5B5C5D5E50000.

In case of textual identifier the `charset` parameter maps to [IANA](#) defined character set.

## RETURN VALUE

Upon successful completion `SetWTLSCientId` method returns `VARIANT_TRUE`. Otherwise, `VARIANT_FALSE` is returned and the method `LastError` [[3.6.1](#), p. 43] can be used to indicate the error.

## NOTES

In the most cases the identifier shall be textual, and the character set shall be UTF-8 (`charset = 106`).



## 3.6 Diagnostic

### 3.6.1 Last error code

#### NAME

LastError — obtain an error code

#### SYNOPSIS

```
HRESULT LastError([out, retval] long *pVal);
```

#### DESCRIPTION

The [LastError](#) method retrieves the last error code value. Please see [A.1](#) for possible error codes and descriptions.

#### NOTES

You should call the [LastError](#) method immediately when a method's return value indicates failure. All methods doesn't change last error code when they succeed.



### 3.6.2 Logging



## A Appendix

### A.1 List of error codes

This table describes possible error codes returned by the [LastError](#) [3.6.1, p. 43] method.

Value	Description
WPS_S_OK	Operation has been completed successfully
WPS_E_NOT_ALLOWED	Operation could not be completed at this moment
WPS_E_NO_SPACE	Not enough place for an operation result
WPS_E_NO_MEMORY	Operation could not be completed due to low memory
WPS_E_NETWORK	Operation could not be completed due to network error
WPS_E_NOT_OBJECT	Invalid request handle has been passed
WPS_E_SYSTEM	Operation could not be completed due to system error
WPS_E_SSL	Operation could not be completed due to SSL error
WPS_E_TIMEDOUT	Operation could not be completed due to timed out
WPS_E_STATE_NOT_NULL	Operation could not be completed at this moment
WPS_E_CONNECT_ERROR	Session is already established
WPS_E_NOT_CONNECTED	Session is not yet established
WPS_E_DISCONNECT_ERROR	Session was not established
WPS_E_REDIRECT	<a href="#">Redirect</a> [3.2.3, p. 24] event occurred
WPS_E_EINVAL	Operation could not be completed due to invalid parameter
1 ... 5	Number of invalid parameter of the <a href="#">Get</a> [3.3.1, p. 25] or <a href="#">Post</a> [3.3.1, p. 26] method

## A.2 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

API, [19](#)

## Event

- Certificate, [31](#)
- CertificateRequest, [32](#)
- Connect, [26](#)
- Frame, [35](#)
- Progress, [33](#)
- Push, [34](#)
- Redirect, [26](#)
- Reply, [30](#)

## Method

- Abort, [29](#)
- AddTrustedCertificate, [43](#)
- Connect, [24](#)
- Disconnect, [24](#)
- Get, [27](#)
- LastError, [45](#)
- Post, [28](#)
- SetWTLSClientId, [44](#)

## Property

- Certificate, [36](#)
- HeaderEncodingVersion, [41](#)
- HTTPSendTimeout, [42](#)
- HTTPWaitTimeout, [42](#)
- LocalAddress, [22](#)
- Mode, [20](#)
- ProxyAddress, [21](#)
- ProxyPort, [21](#)
- ReadBufferSize, [38](#)
- SAREnabled, [39](#)
- SARFrameSize, [40](#)
- State, [23](#)
- TrustedCertificates, [37](#)
- WriteBufferSize, [38](#)