Contents

WFDCloseHandle function

WFDCloseSession function

WFDOpenHandle function

WFDOpenLegacySession function

WFDStartOpenSession function

WFDUpdateDeviceVisibility function

WL_DISPLAY_PAGES enumeration

WLAN_ASSOCIATION_ATTRIBUTES structure

WLAN_AUTH_CIPHER_PAIR_LIST structure

WLAN_AVAILABLE_NETWORK structure

WLAN_AVAILABLE_NETWORK_LIST structure

WLAN_BSS_ENTRY structure

WLAN_BSS_LIST structure

WLAN_CONNECTION_ATTRIBUTES structure

WLAN_CONNECTION_MODE enumeration

WLAN_CONNECTION_NOTIFICATION_DATA structure

WLAN_CONNECTION_PARAMETERS structure

WLAN_COUNTRY_OR_REGION_STRING_LIST structure

WLAN_DEVICE_SERVICE_GUID_LIST structure

WLAN_DEVICE_SERVICE_NOTIFICATION_DATA structure

WLAN_FILTER_LIST_TYPE enumeration

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS structure

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE structure

WLAN_HOSTED_NETWORK_NOTIFICATION_CODE enumeration

WLAN_HOSTED_NETWORK_OPCODE enumeration

WLAN_HOSTED_NETWORK_PEER_AUTH_STATE enumeration

WLAN_HOSTED_NETWORK_PEER_STATE structure

WLAN_HOSTED_NETWORK_RADIO_STATE structure

WLAN_HOSTED_NETWORK_REASON enumeration

WLAN_HOSTED_NETWORK_SECURITY_SETTINGS structure

WLAN_HOSTED_NETWORK_STATE enumeration

WLAN_HOSTED_NETWORK_STATE_CHANGE structure

WLAN_HOSTED_NETWORK_STATUS structure

WLAN_INTERFACE_CAPABILITY structure

WLAN_INTERFACE_INFO structure

WLAN_INTERFACE_INFO_LIST structure

WLAN_INTERFACE_TYPE enumeration

WLAN_MAC_FRAME_STATISTICS structure

WLAN_MSM_NOTIFICATION_DATA structure

WLAN_NOTIFICATION_CALLBACK callback function

WLAN_PHY_FRAME_STATISTICS structure

WLAN_PHY_RADIO_STATE structure

WLAN_PROFILE_INFO structure

WLAN_PROFILE_INFO_LIST structure

WLAN_RADIO_STATE structure

WLAN_RATE_SET structure

WLAN_RAW_DATA structure

WLAN_RAW_DATA_LIST structure

WLAN_SECURABLE_OBJECT enumeration

WLAN_SECURITY_ATTRIBUTES structure

WLAN_STATISTICS structure

WlanAllocateMemory function

WlanCloseHandle function

WlanConnect function

WlanDeleteProfile function

WlanDeviceServiceCommand function

WlanDisconnect function

WlanEnumInterfaces function

WlanExtractPsdIEDataList function

WlanFreeMemory function

WlanGetAvailableNetworkList function

WlanGetFilterList function

WlanGetInterfaceCapability function

WlanGetNetworkBssList function

WlanGetProfile function

WlanGetProfileCustomUserData function

WlanGetProfileList function

WlanGetSecuritySettings function

WlanGetSupportedDeviceServices function

WlanHostedNetworkForceStart function

WlanHostedNetworkForceStop function

WlanHostedNetworkInitSettings function

WlanHostedNetworkQueryProperty function

WlanHostedNetworkQuerySecondaryKey function

WlanHostedNetworkQueryStatus function

WlanHostedNetworkRefreshSecuritySettings function

WlanHostedNetworkSetProperty function

WlanHostedNetworkSetSecondaryKey function

WlanHostedNetworkStartUsing function

WlanHostedNetworkStopUsing function

WlanIhvControl function

WlanOpenHandle function

WlanQueryAutoConfigParameter function

WlanQueryInterface function

WlanReasonCodeToString function

WlanRegisterDeviceServiceNotification function

WlanRegisterNotification function

WlanRegisterVirtualStationNotification function

WlanRenameProfile function

WlanSaveTemporaryProfile function

WlanScan function

WlanSetAutoConfigParameter function

WlanSetFilterList function

WlanSetInterface function

WlanSetProfile function

WlanSetProfileCustomUserData function

Native Wifi

7/1/2021 • 10 minutes to read • Edit Online

Overview of the Native Wifi technology.

To develop Native Wifi, you need these headers:

- adhoc.h
- dot1x.h
- wlanapi.h

For programming guidance for this technology, see:

- Native Wifi

Enumerations

DOT11_ADHOC_AUTH_ALGORITHM Specifies the authentication algorithm for user or machine authentication on an ad hoc network.
DOT11_ADHOC_CIPHER_ALGORITHM Specifies a cipher algorithm used to encrypt and decrypt information on an ad hoc network.
DOT11_ADHOC_CONNECT_FAIL_REASON Specifies the reason why a connection attempt failed.
DOT11_ADHOC_NETWORK_CONNECTION_STATUS Specifies the connection state of an ad hoc network.
ONEX_AUTH_IDENTITY Specifies the possible values of the identity used for 802.1X authentication status.
ONEX_AUTH_RESTART_REASON Specifies the possible reasons that 802.1X authentication was restarted.
ONEX_AUTH_STATUS Specifies the possible values for the 802.1X authentication status.
ONEX_EAP_METHOD_BACKEND_SUPPORT Specifies the possible values for whether the EAP method configured on the supplicant for 802.1X authentication is supported on the authentication server.

ONEX_NOTIFICATION_TYPE	Specifies the possible values of the NotificationCode member of the WLAN_NOTIFICATION_DATA structure for 802.1X module notifications.
ONEX_REASON_CODE	Specifies the possible values that indicate the reason that 802.1X authentication failed.
WL_DISPLAY_PAGES	Specifies the active tab when the wireless profile user interface dialog box appears.
WLAN_CONNECTION_MODE	Defines the mode of connection.
WLAN_FILTER_LIST_TYPE	Indicates types of filter lists.
WLAN_HOSTED_NETWORK_NOTIFICATION_CODE	Specifies the possible values of the NotificationCode parameter for received notifications on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_OPCODE	Specifies the possible values of the operation code for the properties to query or set on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_PEER_AUTH_STATE	Specifies the possible values for the authentication state of a peer on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_REASON	Specifies the possible values for the result of a wireless Hosted Network function call.
WLAN_HOSTED_NETWORK_STATE	Specifies the possible values for the network state of the wireless Hosted Network.
WLAN_INTERFACE_TYPE	Specifies the wireless interface type.
WLAN_SECURABLE_OBJECT	Defines the securable objects used by Native Wifi Functions.

Functions

Clone	Creates a new enumeration interface.
Clone	Creates a new enumeration interface.
Clone	Creates a new enumeration interface.
CommitCreatedNetwork	Initializes a created network and optionally commits the network's profile to the profile store.
Connect	Connects to a previously created wireless ad hoc network.
CreateNetwork	Creates a wireless ad hoc network.
DeleteProfile	Deletes any profile associated with the network.
Disconnect	Disconnects from an ad hoc network.
GetActiveNetwork	Gets the network that is currently active on the interface.
GetContextGuid	Gets the context identifier associated with the network.
GetDeviceSignature	Gets the signature of the NIC.
GetDot11AuthAlgorithm	Gets the authentication algorithm associated with the security settings.
GetDot11CipherAlgorithm	Gets the cipher algorithm associated with the security settings.
GetFriendlyName	Gets the friendly name of the NIC.

GetIEnumDot11AdHocInterfaces

Returns the set of wireless network interface cards (NICs) available on the machine.

GetIEnumDot11AdHocNetworks

Gets the collection of networks associated with this NIC.

GetIEnumDot11AdHocNetworks

Returns a list of available ad hoc network destinations within connection range.

GetIEnumSecuritySettings

Gets the collection of security settings associated with this NIC.

GetInterface

Gets the interface associated with a network.

GetNetwork

Returns the network associated with a signature.

GetProfileName

Gets the profile name associated with the network.

GetSecuritySetting

Gets the security settings for the network.

GetSignalQuality

Gets the signal quality values associated with the network's radio.

GetSignature

Gets the unique signature associated with the ad hoc network.

GetSSID

Gets the SSID of the network.

GetStatus

Gets the connection status of the active network associated with this NIC.

GetStatus

Gets the connection status of the network.

HasProfile

Returns a boolean value that specifies whether there is a saved profile associated with the network.

IsAdHocCapable

Specifies whether a NIC supports the creation or use of an ad hoc network.

IsDot11d

Specifies whether the NIC is 802.11d compliant.

IsRadioOn

Specifies whether the radio is on.

Next

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.

Next

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.

Next

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.

OnConnectFail

Notifies the client that a connection attempt failed.

OnConnectionStatusChange

Notifies the client that the connection status of the network associated with the NIC has changed.

OnInterfaceAdd

Notifies the client that a new network interface card (NIC) is active.

OnInterfaceRemove

Notifies the client that a network interface card (NIC) has become inactive.

OnNetworkAdd

Notifies the client that a new wireless ad hoc network destination is in range and available for connection.

OnNetworkRemove

Notifies the client that a wireless ad hoc network destination is no longer available for connection.

OnStatusChange

Notifies the client that the connection status of the network has changed.

Reset	Resets to the beginning of the enumeration sequence.
Reset	Resets to the beginning of the enumeration sequence.
Reset	Resets to the beginning of the enumeration sequence.
Skip	Skips over the next specified number of elements in the enumeration sequence.
Skip	Skips over the next specified number of elements in the enumeration sequence.
Skip	Skips over the next specified number of elements in the enumeration sequence.
WFD_OPEN_SESSION_COMPLETE_CALLBACK	Defines the callback function that is called by the WFDStartOpenSession function when the WFDStartOpenSession operation completes.
WFDCancelOpenSession	Indicates that the application wants to cancel a pending WFDStartOpenSession function that has not completed.
WFDCloseHandle	Closes a handle to the Wi-Fi Direct service.
WFDCloseSession	Closes a session after a previously successful call to the WFDStartOpenSession function.
WFDOpenHandle	Opens a handle to the Wi-Fi Direct service and negotiates a version of the Wi-FI Direct API to use.
WFDOpenLegacySession	Retrieves and applies a stored profile for a Wi-Fi Direct legacy device.
WFDStartOpenSession	Starts an on-demand connection to a specific Wi-Fi Direct device, which has been previously paired through the Windows Pairing experience.

WFDUpdateDeviceVisibility	Updates device visibility for the Wi-Fi Direct device address for a given installed Wi-Fi Direct device node.
WLAN_NOTIFICATION_CALLBACK	Defines the type of notification callback function.
WlanAllocateMemory	Allocates memory.
WlanCloseHandle	Closes a connection to the server.
WlanConnect	Attempts to connect to a specific network.
WlanDeleteProfile	Deletes a wireless profile for a wireless interface on the local computer.
WlanDeviceServiceCommand	Allows an OEM or IHV component to communicate with a device service on a particular wireless LAN interface.
WlanDisconnect	Disconnects an interface from its current network.
WlanEnumInterfaces	Enumerates all of the wireless LAN interfaces currently enabled on the local computer.
WlanExtractPsdIEDataList	Extracts the proximity service discovery (PSD) information element (IE) data list from raw IE data included in a beacon.
WlanFreeMemory	Frees memory.
WlanGetAvailableNetworkList	Retrieves the list of available networks on a wireless LAN interface.
WlanGetFilterList	Retrieves a group policy or user permission list.
WlanGetInterfaceCapability	Retrieves the capabilities of an interface.

WlanGetNetworkBssList	Retrieves a list of the basic service set (BSS) entries of the wireless network or networks on a given wireless LAN interface.
WlanGetProfile	Retrieves all information about a specified wireless profile.
WlanGetProfileCustomUserData	Gets the custom user data associated with a wireless profile.
WlanGetProfileList	Retrieves the list of profiles.
WlanGetSecuritySettings	Gets the security settings associated with a configurable object.
WlanGetSupportedDeviceServices	Retrieves a list of the supported device services on a given wireless LAN interface.
WlanHostedNetworkForceStart	Transitions the wireless Hosted Network to the wlan_hosted_network_active state without associating the request with the application's calling handle.
WlanHostedNetworkForceStop	Transitions the wireless Hosted Network to the wlan_hosted_network_idle without associating the request with the application's calling handle.
WlanHostedNetworkInitSettings	Configures and persists to storage the network connection settings (SSID and maximum number of peers, for example) on the wireless Hosted Network if these settings are not already configured.
WlanHostedNetworkQueryProperty	Queries the current static properties of the wireless Hosted Network.
WlanHostedNetworkQuerySecondaryKey	Queries the secondary security key that is configured to be used by the wireless Hosted Network.
WlanHostedNetworkQueryStatus	Queries the current status of the wireless Hosted Network.
WlanHostedNetworkRefreshSecuritySettings	Refreshes the configurable and auto-generated parts of the wireless Hosted Network security settings.

WlanHostedNetworkSetProperty	Sets static properties of the wireless Hosted Network.
WlanHostedNetworkSetSecondaryKey	Configures the secondary security key that will be used by the wireless Hosted Network.
WlanHostedNetworkStartUsing	Starts the wireless Hosted Network.
WlanHostedNetworkStopUsing	Stops the wireless Hosted Network.
WlanIhvControl	Provides a mechanism for independent hardware vendor (IHV) control of WLAN drivers or services.
WlanOpenHandle	Opens a connection to the server.
WlanQueryAutoConfigParameter	Queries for the parameters of the auto configuration service.
WlanQueryInterface	The WlanQueryInterface function queries various parameters of a specified interface.
WlanReasonCodeToString	Retrieves a string that describes a specified reason code.
WlanRegisterDeviceServiceNotification	Allows user mode clients with admin privileges, or User-Mode Driver Framework (UMDF) drivers, to register for unsolicited notifications corresponding to device services that they're interested in.
WlanRegisterNotification	Is used to register and unregister notifications on all wireless interfaces.
WlanRegisterVirtualStationNotification	Is used to register and unregister notifications on a virtual station.
WlanRenameProfile	Renames the specified profile.

WlanSaveTemporaryProfile	Saves a temporary profile to the profile store.
WlanScan	Requests a scan for available networks on the indicated interface.
WlanSetAutoConfigParameter	Sets parameters for the automatic configuration service.
WlanSetFilterList	Sets the permit/deny list.
WlanSetInterface	Sets user-configurable parameters.
WlanSetProfile	Sets the content of a specific profile.
WlanSetProfileCustomUserData	Sets the custom user data associated with a profile.
WlanSetProfileEapUserData	Sets the Extensible Authentication Protocol (EAP) user credentials as specified by raw EAP data.
WlanSetProfileEapXmlUserData	Sets the Extensible Authentication Protocol (EAP) user credentials as specified by an XML string.
WlanSetProfileList	Sets the preference order of profiles.
WlanSetProfilePosition	Sets the position of a single, specified profile in the preference list.
WlanSetPsdIEDataList	Sets the proximity service discovery (PSD) information element (IE) data list.
WlanSetSecuritySettings	Sets the security settings for a configurable object.
WlanUIEditProfile	Displays the wireless profile user interface (UI).

Interfaces

IDot11AdHocInterface Represents a wireless network interface card (NIC).
IDot11AdHocInterfaceNotificationSink Defines the notifications supported by IDot11AdHocInterface.
IDot11AdHocManager Creates and manages 802.11 ad hoc networks.
IDot11AdHocManagerNotificationSink Defines the notifications supported by the IDot11AdHocManager interface.
IDot11AdHocNetwork Represents an available ad hoc network destination within connection range.
IDot11AdHocNetworkNotificationSink Defines the notifications supported by the IDot11AdHocNetwork interface.
IDot11AdHocSecuritySettings Specifies the security settings for a wireless ad hoc network.
IEnumDot11AdHocInterfaces Represents the collection of currently visible 802.11 ad hoc network interfaces.
IEnumDot11AdHocNetworks Represents the collection of currently visible 802.11 ad hoc networks.
IEnumDot11AdHocSecuritySettings Represents the collection of security settings associated with each visible wireless ad hoc network.

Structures

DOT11_NETWORK Contains information about an available wireless network.
DOT11_NETWORK_LIST Contains a list of 802.11 wireless networks.

ONEX_AUTH_PARAMS	Contains 802.1X authentication parameters used for 802.1X authentication.
ONEX_EAP_ERROR	Contains 802.1X EAP error when an error occurs with 802.1X authentication.
ONEX_RESULT_UPDATE_DATA	Contains information on a status change to 802.1X authentication.
ONEX_STATUS	Contains the current 802.1X authentication status.
ONEX_VARIABLE_BLOB	Is used as a member of other 802.1X authentication structures to contain variable-sized members.
WLAN_ASSOCIATION_ATTRIBUTES	Contains association attributes for a connection.
WLAN_AUTH_CIPHER_PAIR_LIST	Contains a list of authentication and cipher algorithm pairs.
WLAN_AVAILABLE_NETWORK	Contains information about an available wireless network.
WLAN_AVAILABLE_NETWORK_LIST	Contains an array of information about available networks.
WLAN_BSS_ENTRY	Contains information about a basic service set (BSS).
WLAN_BSS_LIST	Contains a list of basic service set (BSS) entries.
WLAN_CONNECTION_ATTRIBUTES	Defines the attributes of a wireless connection.
WLAN_CONNECTION_NOTIFICATION_DATA	Contains information about connection related notifications.
WLAN_CONNECTION_PARAMETERS	Specifies the parameters used when using the WlanConnect function.

WLAN_COUNTRY_OR_REGION_STRING_LIST

Contains a list of supported country or region strings.

WLAN_DEVICE_SERVICE_GUID_LIST

Contains an array of device service GUIDs.

WLAN_DEVICE_SERVICE_NOTIFICATION_DATA

A structure that represents a device service notification.

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS

Contains information about the connection settings on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE

Contains information about a network state change for a data peer on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_PEER_STATE

Contains information about the peer state for a peer on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_RADIO_STATE

Contains information about the radio state on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_SECURITY_SETTINGS

Contains information about the security settings on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_STATE_CHANGE

Contains information about a network state change on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_STATUS

Contains information about the status of the wireless Hosted Network.

WLAN_INTERFACE_CAPABILITY

Contains information about the capabilities of an interface.

WLAN_INTERFACE_INFO

Contains information about a wireless LAN interface.

WLAN_INTERFACE_INFO_LIST

Array of NIC interface information.

WLAN_MAC_FRAME_STATISTICS

Contains information about sent and received MAC frames.

WLAN_MSM_NOTIFICATION_DATA Contains information about media specific module (MSM) connection related notifications.
WLAN_PHY_FRAME_STATISTICS Contains information about sent and received PHY frames.
WLAN_PHY_RADIO_STATE Specifies the radio state.
WLAN_PROFILE_INFO Basic information about a profile.
WLAN_PROFILE_INFO_LIST Contains a list of wireless profile information.
WLAN_RADIO_STATE Specifies the radio state on a list of physical layer (PHY) types.
WLAN_RATE_SET The set of supported data rates.
WLAN_RAW_DATA Contains raw data in the form of a blob that is used by some Native Wifi functions.
WLAN_RAW_DATA_LIST Contains raw data in the form of an array of data blobs that are used by some Native Wifi functions.
WLAN_SECURITY_ATTRIBUTES Defines the security attributes for a wireless connection.
WLAN_STATISTICS Assorted statistics about an interface.

adhoc.h header

6/2/2021 • 2 minutes to read • Edit Online

This header is used by Native Wifi. For more information, see:

- Native Wifi

adhoc.h contains the following programming interfaces:

Interfaces

IDot11AdHocInterface	Represents a wireless network interface card (NIC).
IDot11AdHocInterfaceNotificationSink	Defines the notifications supported by IDot11AdHocInterface.
IDot11AdHocManager	Creates and manages 802.11 ad hoc networks.
IDot11AdHocManagerNotificationSink	Defines the notifications supported by the IDot11AdHocManager interface.
IDot11AdHocNetwork	Represents an available ad hoc network destination within connection range.
IDot11AdHocNetworkNotificationSink	Defines the notifications supported by the IDot11AdHocNetwork interface.
IDot11AdHocSecuritySettings	Specifies the security settings for a wireless ad hoc network.
IEnumDot11AdHocInterfaces	Represents the collection of currently visible 802.11 ad hoc network interfaces.
IEnumDot11AdHocNetworks	Represents the collection of currently visible 802.11 ad hoc networks.
IEnumDot11AdHocSecuritySettings	Represents the collection of security settings associated with each visible wireless ad hoc network.

Enumerations

DOT11_ADHOC_AUTH_ALGORITHM Specifies the authentication algorithm for user or machine authentication on an ad hoc network.
DOT11_ADHOC_CIPHER_ALGORITHM Specifies a cipher algorithm used to encrypt and decrypt information on an ad hoc network.
DOT11_ADHOC_CONNECT_FAIL_REASON Specifies the reason why a connection attempt failed.
DOT11_ADHOC_NETWORK_CONNECTION_STATUS Specifies the connection state of an ad hoc network.

DOT11_ADHOC_AUTH_ALGORITHM enumeration (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies the authentication algorithm for user or machine authentication on an ad hoc network.

Syntax

```
typedef enum tagDOT11_ADHOC_AUTH_ALGORITHM {
  DOT11_ADHOC_AUTH_ALGO_INVALID,
  DOT11_ADHOC_AUTH_ALGO_80211_OPEN,
  DOT11_ADHOC_AUTH_ALGO_RSNA_PSK
} DOT11_ADHOC_AUTH_ALGORITHM;
```

Constants

`DOT11_ADHOC_AUTH_ALGO_INVALID`
The authentication algorithm specified is invalid.

`DOT11_ADHOC_AUTH_ALGO_80211_OPEN`
Specifies an IEEE 802.11 Open System authentication algorithm.

`DOT11_ADHOC_AUTH_ALGO_RSNA_PSK`
Specifies an IEEE 802.11i Robust Security Network Association (RSNA) algorithm that uses the pre-shared key (PSK) mode. IEEE 802.1X port authorization is performed by the supplicant and authenticator. Cipher keys are dynamically derived through a pre-shared key that is used on both the supplicant and authenticator.

Remarks

Authentication and cipher algorithms are used in pairs. The following table shows valid algorithm pairs for use on an ad hoc network.

PAIR NAME	DOT11_ADHOC_AUTH_ALGORITHM VALUE	DOT11_ADHOC_CIPHER_ALGORITHM VALUE
Open-None	DOT11_ADHOC_AUTH_ALGO_80211_OPEN	DOT11_ADHOC_CIPHER_ALGO_NONE
Open-WEP	DOT11_ADHOC_AUTH_ALGO_80211_OPEN	DOT11_ADHOC_CIPHER_ALGO_WEP
WPA2PSK	DOT11_ADHOC_AUTH_ALGO_RSNA_PSK	DOT11_ADHOC_CIPHER_ALGO_CCMP

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	adhoc.h

See also

DOT11_ADHOC_CIPHER_ALGORITHM

IDot11AdHocSecuritySettings::GetDot11AuthAlgorithm

DOT11_ADHOC_CIPHER_ALGORITHM enumeration (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies a cipher algorithm used to encrypt and decrypt information on an ad hoc network.

Syntax

```
typedef enum tagDOT11_ADHOC_CIPHER_ALGORITHM {
  DOT11_ADHOC_CIPHER_ALGO_INVALID,
  DOT11_ADHOC_CIPHER_ALGO_NONE,
  DOT11_ADHOC_CIPHER_ALGO_CCMP,
  DOT11_ADHOC_CIPHER_ALGO_WEP
} DOT11_ADHOC_CIPHER_ALGORITHM;
```

Constants

`DOT11_ADHOC_CIPHER_ALGO_INVALID`
The cipher algorithm specified is invalid.

`DOT11_ADHOC_CIPHER_ALGO_NONE`
Specifies that no cipher algorithm is enabled or supported.

`DOT11_ADHOC_CIPHER_ALGO_CCMP`
Specifies a Counter Mode with Cipher Block Chaining Message Authentication Code Protocol (CCMP) algorithm. The CCMP algorithm is specified in the IEEE 802.11i-2004 standard and RFC 3610. CCMP is used with the Advanced Encryption Standard (AES) encryption algorithm, as defined in FIPS PUB 197.

`DOT11_ADHOC_CIPHER_ALGO_WEP`
Specifies a Wired Equivalent Privacy (WEP) algorithm of any length.

Remarks

Authentication and cipher algorithms are used in pairs. The following table shows valid algorithm pairs for use on an ad hoc network.

PAIR NAME	DOT11_ADHOC_AUTH_ALGORITHM VALUE	DOT11_ADHOC_CIPHER_ALGORITHM VALUE
Open-None	DOT11_ADHOC_AUTH_ALGO_80211_OPEN	DOT11_ADHOC_CIPHER_ALGO_NONE
Open-WEP	DOT11_ADHOC_AUTH_ALGO_80211_OPEN	DOT11_ADHOC_CIPHER_ALGO_WEP
WPA2PSK	DOT11_ADHOC_AUTH_ALGO_RSNA_PSK	DOT11_ADHOC_CIPHER_ALGO_CCMP

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	adhoc.h

See also

DOT11_ADHOC_AUTH_ALGORITHM

IDot11AdHocSecuritySettings::GetDot11CipherAlgorithm

DOT11_ADHOC_CONNECT_FAIL_REASON enumeration (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies the reason why a connection attempt failed.

Syntax

```
typedef enum tagDOT11_ADHOC_CONNECT_FAIL_REASON {
  DOT11_ADHOC_CONNECT_FAIL_DOMAIN_MISMATCH,
  DOT11_ADHOC_CONNECT_FAIL_PASSPHRASE_MISMATCH,
  DOT11_ADHOC_CONNECT_FAIL_OTHER
} DOT11_ADHOC_CONNECT_FAIL_REASON;
```

Constants

`DOT11_ADHOC_CONNECT_FAIL_DOMAIN_MISMATCH` The local host's configuration is incompatible with the target network. This occurs when the local host is 802.11d compliant and the regulatory domain of the local host is not compatible with the regulatory domain of the target network. For more information about regulatory domains, see the IEEE 802.11d-2001 standard. The standard can be downloaded from the IEEE website.
`DOT11_ADHOC_CONNECT_FAIL_PASSPHRASE_MISMATCH` The passphrase supplied to authenticate the local machine or user on the target network is incorrect.
`DOT11_ADHOC_CONNECT_FAIL_OTHER` The connection failed for another reason.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	adhoc.h

DOT11_ADHOC_NETWORK_CONNECTION_STATUS enumeration (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies the connection state of an ad hoc network.

Syntax

```
typedef enum tagDOT11_ADHOC_NETWORK_CONNECTION_STATUS {
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS_INVALID,
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS_DISCONNECTED,
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS_CONNECTING,
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS_CONNECTED,
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS_FORMED
} DOT11_ADHOC_NETWORK_CONNECTION_STATUS;
```

Constants

`DOT11_ADHOC_NETWORK_CONNECTION_STATUS_INVALID` The connection status cannot be determined. A network with this status should not be used.
`DOT11_ADHOC_NETWORK_CONNECTION_STATUS_DISCONNECTED` There are no hosts or clients connected to the network. There are also no pending connection requests for this network.
`DOT11_ADHOC_NETWORK_CONNECTION_STATUS_CONNECTING` There is an outstanding connection request. Once the client or host succeeds or fails in its connection attempt, the connection status is updated.
`DOT11_ADHOC_NETWORK_CONNECTION_STATUS_CONNECTED` A client or host is connected to the network.
`DOT11_ADHOC_NETWORK_CONNECTION_STATUS_FORMED` The network has been formed. Once a client or host connects to the network, the connection status is updated.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	adhoc.h

IDot11AdHocInterface interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Represents a wireless network interface card (NIC).

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocInterface** interface inherits from the IUnknown interface. **IDot11AdHocInterface** also has these types of members:

Methods

The **IDot11AdHocInterface** interface has these methods.

IDot11AdHocInterface::GetActiveNetwork Gets the network that is currently active on the interface.
IDot11AdHocInterface::GetDeviceSignature Gets the signature of the NIC.
IDot11AdHocInterface::GetFriendlyName Gets the friendly name of the NIC.
IDot11AdHocInterface::GetIEnumDot11AdHocNetworks Gets the collection of networks associated with this NIC.
IDot11AdHocInterface::GetIEnumSecuritySettings Gets the collection of security settings associated with this NIC.
IDot11AdHocInterface::GetStatus Gets the connection status of the active network associated with this NIC.
IDot11AdHocInterface::IsAdHocCapable Specifies whether a NIC supports the creation or use of an ad hoc network.
IDot11AdHocInterface::IsDot11d Specifies whether the NIC is 802.11d compliant.

IDot11AdHocInterface::IsRadioOn Specifies whether the radio is on.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

IDot11AdHocInterface::GetActiveNetwork method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the network that is currently active on the interface.

Syntax

```
HRESULT GetActiveNetwork(
  IDot11AdHocNetwork **ppNetwork
);
```

Parameters

`ppNetwork`

A pointer to an IDot11AdHocNetwork object that represents the active network.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetStatus

IDot11AdHocInterface::GetDeviceSignature method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the signature of the NIC. This signature is stored in the registry and it is used by TCP/IP to uniquely identify the NIC.

Syntax

```
HRESULT GetDeviceSignature(
  GUID *pSignature
);
```

Parameters

`pSignature`

The signature of the NIC.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetFriendlyName method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the friendly name of the NIC.

Syntax

```
HRESULT GetFriendlyName(
  LPWSTR *ppszName
);
```

Parameters

`ppszName`

The friendly name of the NIC. The SSID of the network is used as the friendly name.

You must free this string using CoTaskMemFree.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetIEnumDot11AdHocNetworks method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the collection of networks associated with this NIC.

Syntax

```
HRESULT GetIEnumDot11AdHocNetworks(
  GUID                    *pFilterGuid,
  IEnumDot11AdHocNetworks **ppEnum
);
```

Parameters

`pFilterGuid`

An optional parameter that specifies the GUID of the application that created the network. An application can use this identifier to limit the networks enumerated to networks created by the application. For this filtering to work correctly, all instances of the application on all machines must use the same GUID.

`ppEnum`

A pointer to a IEnumDot11AdHocNetworks interface that contains the enumerated networks.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetIEnumSecuritySettings method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the collection of security settings associated with this NIC.

Syntax

```
HRESULT GetIEnumSecuritySettings(
  IEnumDot11AdHocSecuritySettings **ppEnum
);
```

Parameters

`ppEnum`

A pointer to an IEnumDot11AdHocSecuritySettings interface that contains the collection of security settings.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetStatus method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the connection status of the active network associated with this NIC. You can determine the active network by calling IDot11AdHocInterface::GetActiveNetwork.

Syntax

```
HRESULT GetStatus(
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS *pState
);
```

Parameters

`pState`

A pointer to a DOT11_ADHOC_NETWORK_CONNECTION_STATUS value that specifies the connection state.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::GetActiveNetwork

IDot11AdHocInterface::IsAdHocCapable method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies whether a NIC supports the creation or use of an ad hoc network.

Syntax

```
HRESULT IsAdHocCapable(
  BOOLEAN *pfAdHocCapable
);
```

Parameters

`pfAdHocCapable`

A pointer to a boolean that specifies the NIC's ad hoc network capabilities. The boolean value is set to **TRUE** if the NIC supports the creation and use of ad hoc networks and **FALSE** otherwise.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Remarks

pfAdHocCapable can be set to **FALSE** for many reasons, including the following:

- Group policy prohibits the use of ad hoc networks on this interface
- The machine is configured to only connect to infrastructure networks, or the machine configuration disallows wireless connections
- The NIC does not support ad hoc networks

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::IsDot11d method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies whether the NIC is 802.11d compliant.

Syntax

```
HRESULT IsDot11d(
  BOOLEAN *pf11d
);
```

Parameters

`pf11d`

A pointer to a boolean that specifies 802.11d compliance. The boolean value is set to **TRUE** if the NIC is compliant and **FALSE** otherwise.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterface::IsRadioOn method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Specifies whether the radio is on.

Syntax

```
HRESULT IsRadioOn(
  BOOLEAN *pfIsRadioOn
);
```

Parameters

`pfIsRadioOn`

A pointer to a boolean that specifies the radio state. The value is set to **TRUE** if the radio is on.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterfaceNotificationSink interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocInterfaceNotificationSink** interface defines the notifications supported by IDot11AdHocInterface. To register for notifications, call the Advise method on an instantiated IDot11AdHocManager object with the **IDot11AdHocInterfaceNotificationSink** interface passed as the *pUnk* parameter. To terminate notifications, call the Unadvise method.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocInterfaceNotificationSink** interface inherits from the IUnknown interface. **IDot11AdHocInterfaceNotificationSink** also has these types of members:

Methods

The **IDot11AdHocInterfaceNotificationSink** interface has these methods.

IDot11AdHocInterfaceNotificationSink::OnConnectionStatusChange	Notifies the client that the connection status of the network associated with the NIC has changed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterface

IDot11AdHocInterfaceNotificationSink::OnConnectionStatusChang method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that the connection status of the network associated with the NIC has changed.

Syntax

```
HRESULT OnConnectionStatusChange(
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS eStatus
);
```

Parameters

`eStatus`

A pointer to a DOT11_ADHOC_NETWORK_CONNECTION_STATUS value that specifies the new connection state.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Remarks

This notification is triggered when the connection status changes as a result of connection and disconnection requests issued by the current application. It is also triggered when other applications issue successful connection and disconnection requests using the IDot11AdHocNetwork methods or the Native Wifi functions.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocInterfaceNotificationSink

IDot11AdHocManager interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocManager** interface creates and manages 802.11 ad hoc networks. It is the top-level 802.11 ad hoc interface and the only ad hoc interface with a coclass. As such, it is the only ad hoc interface that can be instantiated by CoCreateInstance.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

The **IDot11AdHocManager** coclass implements the IConnectionPoint interface. The Advise method can be used to register for network manager, network, and interface-related notifications. Notifications are implemented by the IDot11AdHocManagerNotificationSink interface. To register for notifications, call the **Advise** method with the appropriate notification sink interface as the *pUnk* parameter.

Inheritance

The **IDot11AdHocManager** interface inherits from the IUnknown interface. **IDot11AdHocManager** also has these types of members:

Methods

The **IDot11AdHocManager** interface has these methods.

IDot11AdHocManager::CommitCreatedNetwork	Initializes a created network and optionally commits the network's profile to the profile store.
IDot11AdHocManager::CreateNetwork	Creates a wireless ad hoc network.
IDot11AdHocManager::GetIEnumDot11AdHocInterfaces	Returns the set of wireless network interface cards (NICs) available on the machine.
IDot11AdHocManager::GetIEnumDot11AdHocNetworks	Returns a list of available ad hoc network destinations within connection range.
IDot11AdHocManager::GetNetwork	Returns the network associated with a signature.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

IDot11AdHocManager::CommitCreatedNetwork method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Initializes a created network and optionally commits the network's profile to the profile store. The network must be created using CreateNetwork before calling **CommitCreatedNetwork**.

Syntax

```
HRESULT CommitCreatedNetwork(
  IDot11AdHocNetwork *pIAdHoc,
  BOOLEAN            fSaveProfile,
  BOOLEAN            fMakeSavedProfileUserSpecific
);
```

Parameters

`pIAdHoc`

A pointer to a IDot11AdHocNetwork interface that specifies the network to be initialized and committed.

`fSaveProfile`

An optional parameter that specifies whether a wireless profile should be saved. If **TRUE**, the profile is saved to the profile store. Once a profile has been saved, the user can modify the profile using the **Manage Wireless Network** user interface. Profiles can also be modified using the Native Wifi Functions.

Saving a profile modifies the network signature returned by IDot11AdHocNetwork::GetSignature.

`fMakeSavedProfileUserSpecific`

An optional parameter that specifies whether the profile to be saved is an all-user profile. If set to **TRUE**, the profile is specific to the current user. If set to **FALSE**, the profile is an all-user profile and can be used by any user logged into the machine. This parameter is ignored if *fSaveProfile* is **FALSE**.

By default, only members of the Administrators group can persist an all-user profile. These security settings can be altered using the WlanSetSecuritySettings function. Your application must be launched by a user with sufficient privileges for an all-user profile to be persisted successfully.

If your application is running in a Remote Desktop window, you can only save an all-user profile. User-specific profiles cannot be saved from an application running remotely.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.

E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocManager::CreateNetwork

IDot11AdHocManager::CreateNetwork method (adhoc.h)

6/2/2021 • 3 minutes to read • Edit Online

Creates a wireless ad hoc network. Other clients and hosts can connect to this network.

Syntax

```
HRESULT CreateNetwork(
  LPCWSTR                     Name,
  LPCWSTR                     Password,
  LONG                        GeographicalId,
  IDot11AdHocInterface        *pInterface,
  IDot11AdHocSecuritySettings *pSecurity,
  GUID                        *pContextGuid,
  IDot11AdHocNetwork          **pIAdHoc
);
```

Parameters

`Name`

The friendly name of the network. This string should be limited to 32 characters. The SSID should be used as the friendly name. This name is broadcasted in a beacon.

`Password`

The password used for machine or user authentication on the network.

The length of the password string depends on the security settings passed in the *pSecurity* parameter. The following table shows the password length associated with various security settings.

SECURITY SETTINGS	PASSWORD LENGTH
Open-None	0
Open-WEP	5 or 13 characters; 10 or 26 hexadecimal digits
WPA2PSK	8 to 63 characters

For the enumerated values that correspond to the security settings pair above, see DOT11_ADHOC_AUTH_ALGORITHM and DOT11_ADHOC_CIPHER_ALGORITHM

`GeographicalId`

The geographical location in which the network will be created. For a list of possible values, see Table of Geographical Locations.

If the interface is not 802.11d conformant, this value is ignored. That means if IDot11AdHocInterface::IsDot11d returns **FALSE**, this value is ignored.

If you are not sure which value to use, set *GeographicalId* to CTRY_DEFAULT. If you use CTRY_DEFAULT, 802.11d conformance is not enforced.

`pInterface`

An optional pointer to an IDot11AdHocInterface that specifies the network interface upon which the new network is created. If this parameter is **NULL**, the first unused interface is used. If all interfaces are in use, the first enumerated interface is used. In that case, the previous network on the interface is disconnected.

`pSecurity`

A pointer to an IDot11AdHocSecuritySettings interface that specifies the security settings used on the network.

`pContextGuid`

An optional parameter that specifies the GUID of the application that created the network. An application can use this identifier to limit the networks enumerated by GetIEnumDot11AdHocNetworks to networks created by the application. For this filtering to work correctly, all instances of the application on all machines must use the same GUID.

`pIAdHoc`

A pointer to an IDot11AdHocNetwork interface that represents the created network.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.
HRESULT_FROM_WIN32(ERROR_ALREADY_EXISTS)	A network with the specified *Name* already exists.
HRESULT_FROM_WIN32(ERROR_NOT_READY)	The *pInterface* interface reports that its radio is turned off.

HRESULT_FROM_WIN32(ERROR_NOT_CAPABLE)	The *pInterface* interface reports that it is not capable of forming an ad hoc network. This condition can occur because the NIC does not support ad hoc networks, or because the NIC does not support the security settings supplied by *pSecurity*.
HRESULT_FROM_WIN32(ERROR_NOT_SUPPORTED)	The *pSecurity* settings are not supported by the *pInterface* interface.
HRESULT_FROM_WIN32(ERROR_ILL_FORMED_PASSWORD)	The *Password* supplied is invalid. The password supplied may be an invalid length for the security settings supplied by *pSecurity*.
HRESULT_FROM_WIN32(ERROR_NOT_FOUND)	A wireless network interface card was not found on the machine.
HRESULT_FROM_WIN32(ERROR_CURRENT_DOMAIN_NOT_ALLOWED)	Group policy or administrative settings prohibit the creation of the network.

Remarks

After a successful **CreateNetwork** call, the network object returned by *plAdHoc* is provisioned but not constructed. A subsequent call to CommitCreatedNetwork initializes the network. Beacons are not sent until the network is committed.

There are no clients or hosts connected to the network after a **CreateNetwork** call. Applications are notified of both successful and failed connection attempts using the IDot11AdHocManagerNotificationSink interface. For information about registering for notifications on that interface, see IDot11AdHocManager.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

CommitCreatedNetwork

IDot11AdHocManager

IDot11AdHocManager::GetIEnumDot11AdHocInterfaces method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Returns the set of wireless network interface cards (NICs) available on the machine.

Syntax

```
HRESULT GetIEnumDot11AdHocInterfaces(
  IEnumDot11AdHocInterfaces **ppEnum
);
```

Parameters

`ppEnum`

A pointer to an IEnumDot11AdHocInterfaces interface that contains the list of NICs.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocManager::GetIEnumDot11AdHocNetworks method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Returns a list of available ad hoc network destinations within connection range. This list may be filtered.

Syntax

```
HRESULT GetIEnumDot11AdHocNetworks(
  GUID                    *pContextGuid,
  IEnumDot11AdHocNetworks **ppEnum
);
```

Parameters

`pContextGuid`

An optional parameter that specifies the GUID of the application that created the network. An application can use this identifier to limit the networks enumerated to networks created by the application. For this filtering to work correctly, all instances of the application on all machines must use the same GUID.

`ppEnum`

A pointer to an IEnumDot11AdHocNetworks interface that contains the enumerated networks.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocManager::GetNetwork method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Returns the network associated with a signature.

Syntax

```
HRESULT GetNetwork(
  GUID               *NetworkSignature,
  IDot11AdHocNetwork **pNetwork
);
```

Parameters

`NetworkSignature`

A signature that uniquely identifies an ad hoc network. This signature is generated from certain network attributes.

`pNetwork`

A pointer to an IDot11AdHocNetwork interface that represents the network associated with the signature.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocManagerNotificationSink interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocManagerNotificationSink** interface defines the notifications supported by the IDot11AdHocManager interface. To register for notifications, call the Advise method on an instantiated IDot11AdHocManager object with the **IDot11AdHocManagerNotificationSink** interface passed as the *pUnk* parameter. To terminate notifications, call the Unadvise method.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocManagerNotificationSink** interface inherits from the IUnknown interface. **IDot11AdHocManagerNotificationSink** also has these types of members:

Methods

The **IDot11AdHocManagerNotificationSink** interface has these methods.

IDot11AdHocManagerNotificationSink::OnInterfaceAdd	Notifies the client that a new network interface card (NIC) is active.
IDot11AdHocManagerNotificationSink::OnInterfaceRemove	Notifies the client that a network interface card (NIC) has become inactive.
IDot11AdHocManagerNotificationSink::OnNetworkAdd	Notifies the client that a new wireless ad hoc network destination is in range and available for connection.
IDot11AdHocManagerNotificationSink::OnNetworkRemove	Notifies the client that a wireless ad hoc network destination is no longer available for connection.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocManagerNotificationSink::OnInterfaceAdd method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that a new network interface card (NIC) is active.

Syntax

```
HRESULT OnInterfaceAdd(
  IDot11AdHocInterface *pIAdHocInterface
);
```

Parameters

`pIAdHocInterface`

A pointer to an IDot11AdHocInterface interface that represents the activated NIC.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManagerNotificationSink

IDot11AdHocManagerNotificationSink::OnInterfaceRemove method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that a network interface card (NIC) has become inactive.

Syntax

```
HRESULT OnInterfaceRemove(
  GUID *Signature
);
```

Parameters

`Signature`

A pointer to a signature that uniquely identifies the inactive NIC. For more information about signatures, see IDot11AdHocInterface::GetDeviceSignature.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManagerNotificationSink

IDot11AdHocManagerNotificationSink::OnNetworkAdd method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that a new wireless ad hoc network destination is in range and available for connection.

Syntax

```
HRESULT OnNetworkAdd(
  IDot11AdHocNetwork *pIAdHocNetwork
);
```

Parameters

`pIAdHocNetwork`

A pointer to an IDot11AdHocNetwork interface that represents the new network.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManagerNotificationSink

IDot11AdHocManagerNotificationSink::OnNetworkRemove method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that a wireless ad hoc network destination is no longer available for connection.

Syntax

```
HRESULT OnNetworkRemove(
  GUID *Signature
);
```

Parameters

`Signature`

A pointer to a signature that uniquely identifies the newly unavailable network. For more information about signatures, see IDot11AdHocNetwork::GetSignature.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManagerNotificationSink

IDot11AdHocNetwork interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocNetwork** interface represents an available ad hoc network destination within connection range. Before an application can connect to a network, the network must have been created using IDot11AdHocManager::CreateNetwork and committed using IDot11AdHocManager::CommitCreatedNetwork.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocNetwork** interface inherits from the IUnknown interface. **IDot11AdHocNetwork** also has these types of members:

Methods

The **IDot11AdHocNetwork** interface has these methods.

IDot11AdHocNetwork::Connect	Connects to a previously created wireless ad hoc network.
IDot11AdHocNetwork::DeleteProfile	Deletes any profile associated with the network.
IDot11AdHocNetwork::Disconnect	Disconnects from an ad hoc network.
IDot11AdHocNetwork::GetContextGuid	Gets the context identifier associated with the network.
IDot11AdHocNetwork::GetInterface	Gets the interface associated with a network.
IDot11AdHocNetwork::GetProfileName	Gets the profile name associated with the network.
IDot11AdHocNetwork::GetSecuritySetting	Gets the security settings for the network.

IDot11AdHocNetwork::GetSignalQuality Gets the signal quality values associated with the network's radio.
IDot11AdHocNetwork::GetSignature Gets the unique signature associated with the ad hoc network.
IDot11AdHocNetwork::GetSSID Gets the SSID of the network.
IDot11AdHocNetwork::GetStatus Gets the connection status of the network.
IDot11AdHocNetwork::HasProfile Returns a boolean value that specifies whether there is a saved profile associated with the network.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocManager

IDot11AdHocNetwork::Connect method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Connects to a previously created wireless ad hoc network. Before an application can connect to a network, the network must have been created using IDot11AdHocManager::CreateNetwork and committed using IDot11AdHocManager::CommitCreatedNetwork.

Syntax

```
HRESULT Connect(
  LPCWSTR Passphrase,
  LONG    GeographicalId,
  BOOLEAN fSaveProfile,
  BOOLEAN fMakeSavedProfileUserSpecific
);
```

Parameters

`Passphrase`

The password string used to authenticate the user or machine on the network.

The length of the password string depends on the security settings passed in the *pSecurity* parameter of the CreateNetwork call. The following table shows the password length associated with various security settings.

SECURITY SETTINGS	PASSWORD LENGTH
Open-None	0
Open-WEP	5 or 13 characters; 10 or 26 hexadecimal digits
WPA2PSK	8 to 63 characters

For the enumerated values that correspond to the security settings pair above, see DOT11_ADHOC_AUTH_ALGORITHM and DOT11_ADHOC_CIPHER_ALGORITHM.

`GeographicalId`

The geographical location in which the network was created. For a list of possible values, see Table of Geographical Locations.

`fSaveProfile`

An optional parameter that specifies whether a wireless profile should be saved. If **TRUE**, the profile is saved to the profile store. Once a profile is saved, the user can modify the profile using the **Manage Wireless Network** user interface. Profiles can also be modified using the Native Wifi Functions.

Saving a profile modifies the network signature returned by IDot11AdHocNetwork::GetSignature.

`fMakeSavedProfileUserSpecific`

An optional parameter that specifies whether the profile to be saved is an all-user profile. If set to **TRUE**, the

profile is specific to the current user. If set to **FALSE**, the profile is an all-user profile and can be used by any user logged into the machine. This parameter is ignored if *fSaveProfile* is **FALSE**.

By default, only members of the Administrators group can save an all-user profile. These security settings can be altered using the WlanSetSecuritySettings function. Your application must be launched by a user with sufficient privileges for an all-user profile to be saved successfully.

If your application is running in a Remote Desktop window, you can only save an all-user profile. User-specific profiles cannot be saved from an application running remotely.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Remarks

This method is asynchronous. **Connect** returns S_OK immediately if the parameters passed to the method are valid. However, a return code of S_OK does not indicate that the connection was successful. You must register for notifications on the IDot11AdHocNetworkNotificationSink interface to be notified of connection success or failure. The IDot11AdHocNetworkNotificationSink::OnStatusChange method returns the connection status. For more information about registering for notifications, see IDot11AdHocManager.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::Disconnect

IDot11AdHocNetwork::DeleteProfile method (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Deletes any profile associated with the network.

Syntax

```
HRESULT DeleteProfile();
```

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::HasProfile

IDot11AdHocNetwork::Disconnect method (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Disconnects from an ad hoc network.

Syntax

```
HRESULT Disconnect();
```

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::Connect

IDot11AdHocNetwork::GetContextGuid method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the context identifier associated with the network. This GUID identifies the application that created the network.

Syntax

```
HRESULT GetContextGuid(
  GUID *pContextGuid
);
```

Parameters

`pContextGuid`

The context identifier associated with the network. If no ContextGuid was specified when the CreateNetwork call was made, the GUID returned consists of all zeros.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetInterface method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the interface associated with a network.

Syntax

```
HRESULT GetInterface(
  IDot11AdHocInterface **pAdHocInterface
);
```

Parameters

`pAdHocInterface`

A pointer to an IDot11AdHocInterface.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetProfileName method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the profile name associated with the network.

Syntax

```
HRESULT GetProfileName(
  LPWSTR *ppszwProfileName
);
```

Parameters

`ppszwProfileName`

The name of the profile associated with the network. If the network has no profile, this parameter is **NULL**.

You must free this string using CoTaskMemFree.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::DeleteProfile

IDot11AdHocNetwork::HasProfile

IDot11AdHocNetwork::GetSecuritySetting method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the security settings for the network.

Syntax

```
HRESULT GetSecuritySetting(
  IDot11AdHocSecuritySettings **pAdHocSecuritySetting
);
```

Parameters

`pAdHocSecuritySetting`

A pointer to an IDot11AdHocSecuritySettings interface that contains the security settings for the network.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetSignalQuality method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the signal quality values associated with the network's radio.

Syntax

```
HRESULT GetSignalQuality(
  ULONG *puStrengthValue,
  ULONG *puStrengthMax
);
```

Parameters

`puStrengthValue`

The current signal strength. This parameter takes a ULONG value between 0 and *puStrengthMax*.

`puStrengthMax`

The maximum signal strength value. This parameter takes a ULONG value between 0 and 100. By default, *puStrengthMax* is set to 100.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Remarks

Signal strength, in this context, is measured on a linear scale. When *puStrengthMax* is set to the default value of

100, *puStrengthValue* represents the percentage of the maximum signal strength currently used for transmission.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetSignature method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the unique signature associated with the ad hoc network. The signature uniquely identifies an IDot11AdHocNetworkobject with a particular set of attributes.

Syntax

```
HRESULT GetSignature(
  GUID *pSignature
);
```

Parameters

`pSignature`

A signature that uniquely identifies an ad hoc network. This signature is generated from certain network attributes.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Remarks

Do not cache the returned signature locally. Whenever a network object changes, its signature changes. Actions that are not associated with notifications, such as saving the network's profile, can cause the signature to change.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetSSID method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the SSID of the network.

Syntax

```
HRESULT GetSSID(
  LPWSTR *ppszwSSID
);
```

Parameters

`ppszwSSID`

The SSID of the network.

You must free this string using CoTaskMemFree.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::GetStatus method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the connection status of the network.

Syntax

```
HRESULT GetStatus(
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS *eStatus
);
```

Parameters

`eStatus`

A pointer to a DOT11_ADHOC_NETWORK_CONNECTION_STATUS value that specifies the connection state.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::HasProfile method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Returns a boolean value that specifies whether there is a saved profile associated with the network.

Syntax

```
HRESULT HasProfile(
  BOOLEAN *pf11d
);
```

Parameters

`pf11d`

Specifies whether the network has a profile. This value is set to **TRUE** if the network has a profile and **FALSE** otherwise.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetwork::DeleteProfile

IDot11AdHocNetwork::GetProfileName

IDot11AdHocNetworkNotificationSink interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocNetworkNotificationSink** interface defines the notifications supported by the IDot11AdHocNetwork interface. To register for notifications, call the Advise method on an instantiated IDot11AdHocManager object with the **IDot11AdHocNetworkNotificationSink** interface passed as the *pUnk* parameter. To terminate notifications, call the Unadvise method.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocNetworkNotificationSink** interface inherits from the IUnknown interface. **IDot11AdHocNetworkNotificationSink** also has these types of members:

Methods

The **IDot11AdHocNetworkNotificationSink** interface has these methods.

IDot11AdHocNetworkNotificationSink::OnConnectFail	Notifies the client that a connection attempt failed.
IDot11AdHocNetworkNotificationSink::OnStatusChange	Notifies the client that the connection status of the network has changed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetwork

IDot11AdHocNetworkNotificationSink::OnConnectFail method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that a connection attempt failed. The connection attempt may have been initiated by the client itself or by another application using the IDot11AdHocNetwork methods or the Native Wifi functions.

Syntax

```
HRESULT OnConnectFail(
  DOT11_ADHOC_CONNECT_FAIL_REASON eFailReason
);
```

Parameters

`eFailReason`

A DOT11_ADHOC_CONNECT_FAIL_REASON value that specifies the reason the connection attempt failed.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocNetworkNotificationSink

IDot11AdHocNetworkNotificationSink::OnStatusChange method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Notifies the client that the connection status of the network has changed.

Syntax

```
HRESULT OnStatusChange(
  DOT11_ADHOC_NETWORK_CONNECTION_STATUS eStatus
);
```

Parameters

`eStatus`

A DOT11_ADHOC_NETWORK_CONNECTION_STATUS value that specifies the updated connection status.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Remarks

This notification is triggered when the connection status changes as a result of connection and disconnection requests issued by the current application. It is also triggered when other applications issue successful connection and disconnection requests using the IDot11AdHocNetwork methods or the Native Wifi functions. Connection and disconnection requests triggered by the user interface will also trigger the **OnStatusChange** notification.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	adhoc.h

See also

IDot11AdHocNetworkNotificationSink

IDot11AdHocSecuritySettings interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

The **IDot11AdHocSecuritySettings** interface specifies the security settings for a wireless ad hoc network.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IDot11AdHocSecuritySettings** interface inherits from the IUnknown interface.
IDot11AdHocSecuritySettings also has these types of members:

Methods

The **IDot11AdHocSecuritySettings** interface has these methods.

IDot11AdHocSecuritySettings::GetDot11AuthAlgorithm	Gets the authentication algorithm associated with the security settings.
IDot11AdHocSecuritySettings::GetDot11CipherAlgorithm	Gets the cipher algorithm associated with the security settings.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

IDot11AdHocSecuritySettings::GetDot11AuthAlgorithm method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the authentication algorithm associated with the security settings. The authentication algorithm is used to authenticate machines and users connecting to the ad hoc network associated with an interface.

Syntax

```
HRESULT GetDot11AuthAlgorithm(
  DOT11_ADHOC_AUTH_ALGORITHM *pAuth
);
```

Parameters

`pAuth`

A pointer to a DOT11_ADHOC_AUTH_ALGORITHM value that specifies the authentication algorithm.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	The value pointed to by *pAuth* is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	The pointer *pAuth* is invalid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocSecuritySettings

IDot11AdHocSecuritySettings::GetDot11CipherAlgorithm

IDot11AdHocSecuritySettings::GetDot11CipherAlgorithm method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the cipher algorithm associated with the security settings. The cipher algorithm is used to encrypt and decrypt information sent on the ad hoc network associated with an interface.

Syntax

```
HRESULT GetDot11CipherAlgorithm(
  DOT11_ADHOC_CIPHER_ALGORITHM *pCipher
);
```

Parameters

`pCipher`

A pointer to a DOT11_ADHOC_CIPHER_ALGORITHM value that specifies the cipher algorithm.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	The value pointed to by *pCipher* is invalid.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	The pointer *pCipher* is invalid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	adhoc.h

See also

IDot11AdHocSecuritySettings

IDot11AdHocSecuritySettings::GetDot11AuthAlgorithm

IEnumDot11AdHocInterfaces interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

This interface represents the collection of currently visible 802.11 ad hoc network interfaces. It is a standard enumerator.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IEnumDot11AdHocInterfaces** interface inherits from the IUnknown interface. **IEnumDot11AdHocInterfaces** also has these types of members:

Methods

The **IEnumDot11AdHocInterfaces** interface has these methods.

IEnumDot11AdHocInterfaces::Clone	Creates a new enumeration interface.
IEnumDot11AdHocInterfaces::Next	Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.
IEnumDot11AdHocInterfaces::Reset	Resets to the beginning of the enumeration sequence.
IEnumDot11AdHocInterfaces::Skip	Skips over the next specified number of elements in the enumeration sequence.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

IEnumDot11AdHocInterfaces::Clone method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Creates a new enumeration interface.

Syntax

```
HRESULT Clone(
  IEnumDot11AdHocInterfaces **ppEnum
);
```

Parameters

`ppEnum`

A pointer that, on successful return, points to an IEnumDot11AdHocInterfacesinterface.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocInterfaces

IEnumDot11AdHocInterfaces::Next method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved. If there are fewer than the requested number of elements left in the sequence, it retrieves the remaining elements.

Syntax

```
HRESULT Next(
  ULONG                cElt,
  IDot11AdHocInterface **rgElt,
  ULONG                *pcEltFetched
);
```

Parameters

`cElt`

The number of elements requested.

`rgElt`

A pointer to a variable that, on successful return, points to an array of pointers to IDot11AdHocInterface interfaces. The array is of size *cElt*.

`pcEltFetched`

A pointer to a variable that specifies the number of elements returned in *rgElt*.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.

E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocInterfaces

IEnumDot11AdHocInterfaces::Reset method (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Resets to the beginning of the enumeration sequence.

Syntax

```
HRESULT Reset();
```

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocInterfaces

IEnumDot11AdHocInterfaces::Skip method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Skips over the next specified number of elements in the enumeration sequence.

Syntax

```
HRESULT Skip(
  ULONG cElt
);
```

Parameters

`cElt`

The number of elements to skip.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocInterfaces

IEnumDot11AdHocNetworks interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

This interface represents the collection of currently visible 802.11 ad hoc networks. It is a standard enumerator.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IEnumDot11AdHocNetworks** interface inherits from the IUnknown interface.
IEnumDot11AdHocNetworks also has these types of members:

Methods

The **IEnumDot11AdHocNetworks** interface has these methods.

IEnumDot11AdHocNetworks::Clone	Creates a new enumeration interface.
IEnumDot11AdHocNetworks::Next	Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.
IEnumDot11AdHocNetworks::Reset	Resets to the beginning of the enumeration sequence.
IEnumDot11AdHocNetworks::Skip	Skips over the next specified number of elements in the enumeration sequence.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

IEnumDot11AdHocNetworks::Clone method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Creates a new enumeration interface.

Syntax

```
HRESULT Clone(
  IEnumDot11AdHocNetworks **ppEnum
);
```

Parameters

`ppEnum`

A pointer to a variable that, on successful return, points to an IEnumDot11AdHocNetworksinterface.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocNetworks

IEnumDot11AdHocNetworks::Next method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved. If there are fewer than the requested number of elements left in the sequence, it retrieves the remaining elements.

Syntax

```
HRESULT Next(
  ULONG              cElt,
  IDot11AdHocNetwork **rgElt,
  ULONG              *pcEltFetched
);
```

Parameters

`cElt`

The number of elements requested.

`rgElt`

A pointer to the first element in an array of IDot11AdHocNetwork interfaces. The array is of size *cElt*. The array must exist and be of size *cElt* (at a minimum) before the **Next** method is called, although the array need not be initialized. Upon return, the previously existing array will contain pointers to **IDot11AdHocNetwork** objects.

`pcEltFetched`

A pointer to a variable that specifies the number of elements returned in *rgElt*.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.

E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocNetworks

IEnumDot11AdHocNetworks::Reset method (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Resets to the beginning of the enumeration sequence.

Syntax

```
HRESULT Reset();
```

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocNetworks

IEnumDot11AdHocNetworks::Skip method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Skips over the next specified number of elements in the enumeration sequence.

Syntax

```
HRESULT Skip(
  ULONG cElt
);
```

Parameters

`cElt`

The number of elements to skip.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocNetworks

IEnumDot11AdHocSecuritySettings interface (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

This interface represents the collection of security settings associated with each visible wireless ad hoc network. It is a standard enumerator.

> **Note** Ad hoc mode might not be available in future versions of Windows. Starting with Windows 8.1 and Windows Server 2012 R2, use Wi-Fi Direct instead.

Inheritance

The **IEnumDot11AdHocSecuritySettings** interface inherits from the IUnknown interface. **IEnumDot11AdHocSecuritySettings** also has these types of members:

Methods

The **IEnumDot11AdHocSecuritySettings** interface has these methods.

IEnumDot11AdHocSecuritySettings::Clone	Creates a new enumeration interface.
IEnumDot11AdHocSecuritySettings::Next	Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved.
IEnumDot11AdHocSecuritySettings::Reset	Resets to the beginning of the enumeration sequence.
IEnumDot11AdHocSecuritySettings::Skip	Skips over the next specified number of elements in the enumeration sequence.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	adhoc.h

IEnumDot11AdHocSecuritySettings::Clone method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Creates a new enumeration interface.

Syntax

```
HRESULT Clone(
  IEnumDot11AdHocSecuritySettings **ppEnum
);
```

Parameters

`ppEnum`

A pointer that, on successful return, points to an IEnumDot11AdHocSecuritySettingsinterface.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.
E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocSecuritySettings

IEnumDot11AdHocSecuritySettings::Next method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Gets the specified number of elements from the sequence and advances the current position by the number of items retrieved. If there are fewer than the requested number of elements left in the sequence, it retrieves the remaining elements.

Syntax

```
HRESULT Next(
  ULONG                        cElt,
  IDot11AdHocSecuritySettings **rgElt,
  ULONG                        *pcEltFetched
);
```

Parameters

`cElt`

The number of elements requested.

`rgElt`

A pointer to a variable that, on successful return, points an array of pointers to IDot11AdHocSecuritySettings interfaces. The array is of size *cElt*.

`pcEltFetched`

A pointer to a variable that specifies the number of elements returned in *rgElt*.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.
E_INVALIDARG	One of the parameters is invalid.
E_NOINTERFACE	A specified interface is not supported.

E_OUTOFMEMORY	The method could not allocate the memory required to perform this operation.
E_POINTER	A pointer passed as a parameter is not valid.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocSecuritySettings

IEnumDot11AdHocSecuritySettings::Reset method (adhoc.h)

7/1/2021 • 2 minutes to read • Edit Online

Resets to the beginning of the enumeration sequence.

Syntax

```
HRESULT Reset();
```

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocSecuritySettings

IEnumDot11AdHocSecuritySettings::Skip method (adhoc.h)

6/2/2021 • 2 minutes to read • Edit Online

Skips over the next specified number of elements in the enumeration sequence.

Syntax

```
HRESULT Skip(
  ULONG cElt
);
```

Parameters

`cElt`

The number of elements to skip.

Return value

Possible return values include, but are not limited to, the following.

RETURN CODE	DESCRIPTION
S_OK	The method completed successfully.
E_FAIL	The method failed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	adhoc.h

See also

IEnumDot11AdHocSecuritySettings

dot1x.h header

6/2/2021 • 2 minutes to read • Edit Online

This header is used by Native Wifi. For more information, see:

- Native Wifi

dot1x.h contains the following programming interfaces:

Structures

ONEX_AUTH_PARAMS	Contains 802.1X authentication parameters used for 802.1X authentication.
ONEX_EAP_ERROR	Contains 802.1X EAP error when an error occurs with 802.1X authentication.
ONEX_RESULT_UPDATE_DATA	Contains information on a status change to 802.1X authentication.
ONEX_STATUS	Contains the current 802.1X authentication status.
ONEX_VARIABLE_BLOB	Is used as a member of other 802.1X authentication structures to contain variable-sized members.

Enumerations

ONEX_AUTH_IDENTITY	Specifies the possible values of the identity used for 802.1X authentication status.
ONEX_AUTH_RESTART_REASON	Specifies the possible reasons that 802.1X authentication was restarted.
ONEX_AUTH_STATUS	Specifies the possible values for the 802.1X authentication status.

ONEX_EAP_METHOD_BACKEND_SUPPORT Specifies the possible values for whether the EAP method configured on the supplicant for 802.1X authentication is supported on the authentication server.
ONEX_NOTIFICATION_TYPE Specifies the possible values of the NotificationCode member of the WLAN_NOTIFICATION_DATA structure for 802.1X module notifications.
ONEX_REASON_CODE Specifies the possible values that indicate the reason that 802.1X authentication failed.

ONEX_AUTH_IDENTITY enumeration (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_AUTH_IDENTITY** enumerated type specifies the possible values of the identity used for 802.1X authentication status.

Syntax

```
typedef enum _ONEX_AUTH_IDENTITY {
  OneXAuthIdentityNone,
  OneXAuthIdentityMachine,
  OneXAuthIdentityUser,
  OneXAuthIdentityExplicitUser,
  OneXAuthIdentityGuest,
  OneXAuthIdentityInvalid
} ONEX_AUTH_IDENTITY, PONEX_AUTH_IDENTITY;
```

Constants

`OneXAuthIdentityNone`	No identity is specified in the profile used for 802.1X authentication.
`OneXAuthIdentityMachine`	The identity of the local machine account is used for 802.1X authentication.
`OneXAuthIdentityUser`	The identity of the logged-on user is used for 802.1X authentication.
`OneXAuthIdentityExplicitUser`	The identity of an explicit user as specified in the profile is used for 802.1X authentication. This value is used when performing single signon or when credentials are saved with the profile.
`OneXAuthIdentityGuest`	The identity of the Guest account as specified in the profile is used for 802.1X authentication.
`OneXAuthIdentityInvalid`	The identity is not valid as specified in the profile used for 802.1X authentication.

Remarks

The **ONEX_AUTH_IDENTITY** enumerated type is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The **ONEX_AUTH_IDENTITY** specifies the possible values of the identity used for 802.1X authentication. The **ONEX_AUTH_IDENTITY** is a function of the 802.1X authentication mode selected and various system triggers (user logon and logoff operations, for example).

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the

WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

If the **fOneXAuthParams** member in the ONEX_RESULT_UPDATE_DATA structure is set, then the **authParams** member of the **ONEX_RESULT_UPDATE_DATA** structure contains an ONEX_VARIABLE_BLOB structure with an ONEX_AUTH_PARAMS structure embedded starting at the **dwOffset** member of the **ONEX_VARIABLE_BLOB**. This **ONEX_AUTH_PARAMS** structure that contains a value from the **ONEX_AUTH_IDENTITY** enumeration in the **authIdentity** member.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_AUTH_PARAMS

ONEX_RESULT_UPDATE_DATA

ONEX_VARIABLE_BLOB

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_AUTH_PARAMS structure (dot1x.h)

6/2/2021 • 3 minutes to read • Edit Online

The **ONEX_AUTH_PARAMS** structure contains 802.1X authentication parameters used for 802.1X authentication.

Syntax

```
typedef struct _ONEX_AUTH_PARAMS {
  BOOL               fUpdatePending;
  ONEX_VARIABLE_BLOB oneXConnProfile;
  ONEX_AUTH_IDENTITY authIdentity;
  DWORD              dwQuarantineState;
  DWORD              fSessionId : 1;
  DWORD              fhUserToken : 1;
  DWORD              fOnexUserProfile : 1;
  DWORD              fIdentity : 1;
  DWORD              fUserName : 1;
  DWORD              fDomain : 1;
  DWORD              dwSessionId;
  HANDLE             hUserToken;
  ONEX_VARIABLE_BLOB OneXUserProfile;
  ONEX_VARIABLE_BLOB Identity;
  ONEX_VARIABLE_BLOB UserName;
  ONEX_VARIABLE_BLOB Domain;
} ONEX_AUTH_PARAMS, *PONEX_AUTH_PARAMS;
```

Members

`fUpdatePending`

Indicates if a status update is pending for 802.X authentication.

`oneXConnProfile`

The 802.1X authentication connection profile. This member contains an embedded ONEX_CONNECTION_PROFILE structure starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB.

`authIdentity`

The identity used for 802.1X authentication status. This member is a value from the ONEX_AUTH_IDENTITY enumeration.

`dwQuarantineState`

The quarantine isolation state value of the local computer. The isolation state determines its network connectivity. This member corresponds to a value from the EAPHost ISOLATION_STATE enumeration.

`fSessionId`

Indicates if the **ONEX_AUTH_PARAMS** structure contains a session ID in the **dwSessionId** member.

`fhUserToken`

Indicates if the **ONEX_AUTH_PARAMS** structure contains a user token handle in the **hUserToken** member.

For security reasons, the **hUserToken** member of the **ONEX_AUTH_PARAMS** structure returned in the

authParams member of the ONEX_RESULT_UPDATE_DATA structure is always set to **NULL**.

`fOnexUserProfile`

Indicates if the **ONEX_AUTH_PARAMS** structure contains an 802.1X user profile in the **OneXUserProfile** member.

For security reasons, the **OneXUserProfile** member of the **ONEX_AUTH_PARAMS** structure returned in the **authParams** member of the ONEX_RESULT_UPDATE_DATA structure is always set to **NULL**.

`fIdentity`

Indicates if the **ONEX_AUTH_PARAMS** structure contains an 802.1X identity in the **Identity** member.

`fUserName`

Indicates if the **ONEX_AUTH_PARAMS** structure contains a user name used for 802.1X authentication in the **UserName** member.

`fDomain`

Indicates if the **ONEX_AUTH_PARAMS** structure contains a domain used for 802.1X authentication in the **Domain** member.

`dwSessionId`

The session ID of the user currently logged on to the console. This member corresponds to the value returned by the WTSGetActiveConsoleSessionId function. This member contains a session ID if the **fSessionId** bitfield member is set.

`hUserToken`

The user token handle used for 802.1X authentication. This member contains a user token handle if the **fhUserToken** bitfield member is set.

For security reasons, the **hUserToken** member of the **ONEX_AUTH_PARAMS** structure returned in the **authParams** member of the ONEX_RESULT_UPDATE_DATA structure is always set to **NULL**.

`OneXUserProfile`

The 802.1X user profile used for 802.1X authentication. This member contains an embedded user profile starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fOneXUserProfile** bitfield member is set.

For security reasons, the **OneXUserProfile** member of the **ONEX_AUTH_PARAMS** structure returned in the **authParams** member of the ONEX_RESULT_UPDATE_DATA structure is always set to **NULL**.

`Identity`

The 802.1X identity used for 802.1X authentication. This member contains a NULL-terminated Unicode string with the identity starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fIdentity** bitfield member is set.

`UserName`

The user name used for 802.1X authentication. This member contains a NULL-terminated Unicode string with the user name starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fUserName** bitfield member is set.

`Domain`

The domain used for 802.1X authentication. This member contains a NULL-terminated Unicode string with the

domain starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fDomain** bitfield member is set.

Remarks

The **ONEX_AUTH_PARAMS** structure is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

If the **fOneXAuthParams** member in the ONEX_RESULT_UPDATE_DATA structure is set, then the **authParams** member of the **ONEX_RESULT_UPDATE_DATA** structure contains an ONEX_VARIABLE_BLOB structure with an **ONEX_AUTH_PARAMS** structure embedded starting at the **dwOffset** member of the **ONEX_VARIABLE_BLOB**.

For security reasons, the **hUserToken** and **OneXUserProfile** members of the **ONEX_AUTH_PARAMS** structure returned in the **authParams** member are always set to **NULL**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ISOLATION_STATE

ONEX_AUTH_IDENTITY

ONEX_EAP_ERROR

ONEX_NOTIFICATION_TYPE

ONEX_RESULT_UPDATE_DATA

ONEX_VARIABLE_BLOB

WLAN_NOTIFICATION_DATA

WTSGetActiveConsoleSessionId

WlanRegisterNotification

ONEX_AUTH_RESTART_REASON enumeration (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_AUTH_RESTART_REASON** enumerated type specifies the possible reasons that 802.1X authentication was restarted.

Syntax

```
typedef enum _ONEX_AUTH_RESTART_REASON {
  OneXRestartReasonPeerInitiated,
  OneXRestartReasonMsmInitiated,
  OneXRestartReasonOneXHeldStateTimeout,
  OneXRestartReasonOneXAuthTimeout,
  OneXRestartReasonOneXConfigurationChanged,
  OneXRestartReasonOneXUserChanged,
  OneXRestartReasonQuarantineStateChanged,
  OneXRestartReasonAltCredsTrial,
  OneXRestartReasonInvalid
} ONEX_AUTH_RESTART_REASON, PONEX_AUTH_RESTART_REASON;
```

Constants

`OneXRestartReasonPeerInitiated`
The EAPHost component (the peer) requested the 802.1x module to restart 802.1X authentication. This results from a EapHostPeerProcessReceivedPacket function call that returns an EapHostPeerResponseAction enumeration value of **EapHostPeerResponseStartAuthentication** in the *pEapOutput* parameter.

`OneXRestartReasonMsmInitiated`
The Media Specific Module (MSM) initiated the 802.1X authentication restart.

`OneXRestartReasonOneXHeldStateTimeout`
The 802.1X authentication restart was the result of a state timeout. The timer expiring is the heldWhile timer of the 802.1X supplicant state machine defined in IEEE 802.1X - 2004 standard for Port-Based Network Access Control. The heldWhile timer is used by the supplicant state machine to define periods of time during which it
will not attempt to acquire an authenticator.

`OneXRestartReasonOneXAuthTimeout`
The 802.1X authentication restart was the result of an state timeout. The timer expiring is the authWhile timer of the 802.1X supplicant port access entity defined in IEEE 802.1X - 2004 standard for Port-Based Network Access Control. The authWhile timer is used by the supplicant port access entity to determine how long to wait for a request from
the authenticator before timing it out.

`OneXRestartReasonOneXConfigurationChanged`
The 802.1X authentication restart was the result of a configuration change to the current profile.

`OneXRestartReasonOneXUserChanged`
The 802.1X authentication restart was the result of a change of user. This could occur if the current user logs off and new user logs on to the local computer.

`OneXRestartReasonQuarantineStateChanged`	The 802.1X authentication restart was the result of receiving a notification from the EAP quarantine enforcement client (QEC) due to a network health change. If an EAPHost supplicant is participating in network access protection (NAP), the supplicant will respond to changes in the state of its network health. If that state changes, the supplicant must then initiate a re-authentication session. For more information, see the EapHostPeerBeginSession function.
`OneXRestartReasonAltCredsTrial`	The 802.1X authentication restart was caused by a new authentication attempt with alternate user credentials. EAP methods like MSCHAPv2 prefer to use logged-on user credentials for 802.1X authentication. If these user credentials do not work, then a dialog will be displayed to the user that asks permission to use alternate credentials for 802.1X authentication. For more information, see the EapHostPeerBeginSession function and EAP_FLAG_PREFER_ALT_CREDENTIALS flag in the *dwflags* parameter.
`OneXRestartReasonInvalid`	Indicates the end of the range that specifies the possible reasons that 802.1X authentication was restarted.

Remarks

The **ONEX_AUTH_RESTART_REASON** enumerated type is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The **ONEX_AUTH_RESTART_REASON** specifies the possible values for the reason that 802.1X authentication was restarted. A value from this enumeration is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notifications is **OneXNotificationTypeAuthRestarted**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_AUTH_RESTART_REASON** enumeration value that identifies the reason the authentication was restarted.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

EapHostPeerBeginSession

EapHostPeerProcessReceivedPacket

EapHostPeerResponseAction

ONEX_RESULT_UPDATE_DATA

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_AUTH_STATUS enumeration (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_AUTH_STATUS** enumerated type specifies the possible values for the 802.1X authentication status.

Syntax

```
typedef enum _ONEX_AUTH_STATUS {
  OneXAuthNotStarted,
  OneXAuthInProgress,
  OneXAuthNoAuthenticatorFound,
  OneXAuthSuccess,
  OneXAuthFailure,
  OneXAuthInvalid
} ONEX_AUTH_STATUS, PONEX_AUTH_STATUS;
```

Constants

`OneXAuthNotStarted`	802.1X authentication was not started.
`OneXAuthInProgress`	802.1X authentication is in progress.
`OneXAuthNoAuthenticatorFound`	No 802.1X authenticator was found. The 802.1X authentication was attempted, but no 802.1X peer was found. In this case, either the network does not support or is not configured to support the 802.1X standard.
`OneXAuthSuccess`	802.1X authentication was successful.
`OneXAuthFailure`	802.1X authentication was a failure.
`OneXAuthInvalid`	Indicates the end of the range that specifies the possible values for 802.1X authentication status.

Remarks

The **ONEX_AUTH_STATUS** enumerated type is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The **ONEX_AUTH_STATUS** specifies the possible values for the 802.1X authentication status. A value from this enumeration is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notifications is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an ONEX_RESULT_UPDATE_DATA structure that contains a ONEX_STATUS structure member in the **oneXStatus**

structure member. The **ONEX_STATUS** structure contains a **ONEX_AUTH_STATUS** enumeration value in the **authStatus** member.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_RESULT_UPDATE_DATA

ONEX_STATUS

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_EAP_ERROR structure (dot1x.h)

6/2/2021 • 14 minutes to read • Edit Online

The **ONEX_EAP_ERROR** structure contains 802.1X EAP error when an error occurs with 802.1X authentication.

Syntax

```
typedef struct _ONEX_EAP_ERROR {
  DWORD              dwWinError;
  EAP_METHOD_TYPE    type;
  DWORD              dwReasonCode;
  GUID               rootCauseGuid;
  GUID               repairGuid;
  GUID               helpLinkGuid;
  DWORD              fRootCauseString : 1;
  DWORD              fRepairString : 1;
  ONEX_VARIABLE_BLOB RootCauseString;
  ONEX_VARIABLE_BLOB RepairString;
} ONEX_EAP_ERROR, *PONEX_EAP_ERROR;
```

Members

`dwWinError`

The error value defined in the *Winerror.h* header file. This member also sometimes contains the reason the EAP method failed. The existing values for this member for the reason the EAP method failed are defined in the *Eaphosterror.h* header file.

Some possible values are listed below.

VALUE	MEANING
ERROR_PATH_NOT_FOUND 3L	The system cannot find the path specified.
ERROR_INVALID_DATA 13L	The data is not valid.
ERROR_INVALID_PARAMETER 87L	A parameter is incorrect.
ERROR_BAD_ARGUMENTS 160L	One or more arguments are not correct.
ERROR_CANTOPEN 1011L	The configuration registry key could not be opened.

ERROR_DATATYPE_MISMATCH 1629L	The data supplied is of the wrong type.
EAP_I_USER_ACCOUNT_OTHER_ERROR 0x40420110	The EAPHost received EAP failure after the identity exchange. There is likely a problem with the authenticating user's account.
E_UNEXPECTED 0x8000FFFFL	A catastrophic failure occurred.
EAP_E_CERT_STORE_INACCESSIBLE 0x80420010	The certificate store can't be accessed on either the authenticator or the peer.
EAP_E_EAPHOST_METHOD_NOT_INSTALLED 0x80420011	The requested EAP method is not installed.
EAP_E_EAPHOST_EAPQEC_INACCESSIBLE 0x80420013	The EAPHost is not able to communicate with the EAP quarantine enforcement client (QEC) on a client with Network Access Protection (NAP) enabled.
EAP_E_EAPHOST_IDENTITY_UNKNOWN 0x80420014	The EAPHost returns this error if the authenticator fails the authentication after the peer sent its identity.
EAP_E_AUTHENTICATION_FAILED 0x80420015	The EAPHost returns this error on authentication failure.
EAP_I_EAPHOST_EAP_NEGOTIATION_FAILED 0x80420016	The EAPHost returns this error when the client and the server aren't configured with compatible EAP types.
EAP_E_EAPHOST_METHOD_INVALID_PACKET 0x80420017	The EAPMethod received an EAP packet that cannot be processed.
EAP_E_EAPHOST_REMOTE_INVALID_PACKET 0x80420018	The EAPHost received a packet that cannot be processed.
EAP_E_EAPHOST_XML_MALFORMED 0x80420019	The EAPHost configuration schema validation failed.
EAP_E_METHOD_CONFIG_DOES_NOT_SUPPORT_SSO 0x8042001A	The EAP method does not support single signon for the provided configuration.

EAP_E_EAPHOST_METHOD_OPERATION_NOT_SUPPORTED 0x80420020	The EAPHost returns this error when a configured EAP method does not support a requested operation (procedure call).
EAP_E_USER_CERT_NOT_FOUND 0x80420100	The EAPHost could not find the user certificate for authentication.
EAP_E_USER_CERT_INVALID 0x80420101	The user certificate being used for authentication does not have a proper extended key usage (EKU) set.
EAP_E_USER_CERT_EXPIRED 0x80420102	The EAPhost found a user certificate which has expired.
EAP_E_USER_CERT_REVOKED 0x80420103	The user certificate being used for authentication has been revoked.
EAP_E_USER_CERT_OTHER_ERROR 0x80420104	An unknown error occurred with the user certificate being used for authentication.
EAP_E_USER_CERT_REJECTED 0x80420105	The authenticator rejected the user certificate being used for authentication.
EAP_E_USER_CREDENTIALS_REJECTED 0x80420111	The authenticator rejected the user credentials for authentication.
EAP_E_USER_NAME_PASSWORD_REJECTED 0x80420112	The authenticator rejected the user credentials for authentication.
EAP_E_NO_SMART_CARD_READER 0x80420113	No smart card reader was present.
EAP_E_SERVER_CERT_INVALID 0x80420201	The server certificate being user for authentication does not have a proper EKU set .
EAP_E_SERVER_CERT_EXPIRED 0x80420202	The EAPhost found a server certificate which has expired.
EAP_E_SERVER_CERT_REVOKED 0x80420203	The server certificate being used for authentication has been revoked.

EAP_E_SERVER_CERT_OTHER_ERROR 0x80420204	An unknown error occurred with the server certificate being used for authentication.
EAP_E_USER_ROOT_CERT_NOT_FOUND 0x80420300	The EAPHost could not find a certificate in trusted root certificate store for user certificate validation.
EAP_E_USER_ROOT_CERT_INVALID 0x80420301	The authentication failed because the root certificate used for this network is not valid.
EAP_E_USER_ROOT_CERT_EXPIRED 0x80420302	The trusted root certificate needed for user certificate validation has expired.
EAP_E_SERVER_ROOT_CERT_NOT_FOUND 0x80420400	The EAPHost could not find a root certificate in the trusted root certificate store for server certificate velidation.

`type`

The EAP method type that raised the error during 802.1X authentication. The EAP_METHOD_TYPE structure is defined in the *Eaptypes.h* header file.

`dwReasonCode`

The reason the EAP method failed. Some of the values for this member are defined in the *Eaphosterror.h* header file and some are defined in in the *Winerror.h* header file, although other values are possible.

Possible values are listed below.

VALUE	MEANING
ERROR_BAD_ARGUMENTS	One or more arguments are not correct.
ERROR_INVALID_DATA	The data is not valid.
ERROR_INVALID_PARAMETER	A parameter is incorrect.
EAP_I_USER_ACCOUNT_OTHER_ERROR	The EAPHost received EAP failure after the identity exchange. There is likely a problem with the authenticating user's account.
Other	Use FormatMessage to obtain the message string for the returned error.

`rootCauseGuid`

A unique ID that identifies cause of error in EAPHost. An EAP method can define a new GUID and associate the GUID with a specific root cause. The existing values for this member are defined in the *Eaphosterror.h* header file.

VALUE	MEANING
GUID_EapHost_Default {0x00000000, 0x0000, 0x0000, 0, 0, 0, 0, 0, 0, 0, 0}	The default error cause. This is not a fixed GUID when it reaches supplicant, but the first portion will be filled by a generic Win32/RAS error. This helps create a unique GUID for every unique error.
GUID_EapHost_Cause_MethodDLLNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 1}}	EAPHost cannot locate the DLL for the EAP method.
GUID_EapHost_Cause_CertStoreInaccessible {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 4}}	Both the authenticator and the peer are unable to access the certificate store.
GUID_EapHost_Cause_Server_CertExpired {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 5}}	EAPHost found an expired server certificate.
GUID_EapHost_Cause_Server_CertInvalid {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 6}}	The server certificate being user for authentication does not have a proper extended key usage (EKU) set.
GUID_EapHost_Cause_Server_CertNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 7}}	EAPHost could not find the server certificate for authentication.
GUID_EapHost_Cause_Server_CertRevoked {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 8}}	The server certificate being used for authentication has been revoked.
GUID_EapHost_Cause_User_CertExpired {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 9}}	EAPHost found an expired user certificate.
GUID_EapHost_Cause_User_CertInvalid {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xA}}	The user certificate being user for authentication does not have proper extended key usage (EKU) set.
GUID_EapHost_Cause_User_CertNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xB}}	EAPHost could not find a user certificate for authentication.

GUID_EapHost_Cause_User_CertOtherError {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xC}}	An unknown error occurred with the user certification being used for authentication.
GUID_EapHost_Cause_User_CertRejected {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xD}}	The authenticator rejected the user certification.
GUID_EapHost_Cause_User_CertRevoked {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xE}}	The user certificate being used for authentication has been revoked.
GUID_EapHost_Cause_User_Root_CertExpired {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0xF}}	The trusted root certificate needed for user certificate validation has expired.
GUID_EapHost_Cause_User_Root_CertInvalid {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x10}}	The authentication failed because the root certificate used for this network is not valid.
GUID_EapHost_Cause_User_Root_CertNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x11}}	EAPHost could not find a certificate in a trusted root certificate store for user certification validation.
GUID_EapHost_Cause_Server_Root_CertNameRequired {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x12}}	The authentication failed because the certificate on the server computer does not have a server name specified.
GUID_EapHost_Cause_EapNegotiationFailed {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1C}}	The authentication failed because Windows does not have the authentication method required for this network.
GUID_EapHost_Cause_XmlMalformed {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1D}}	The EAPHost configuration schema validation failed.
GUID_EapHost_Cause_MethodDoesNotSupportOperation {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1E}}	EAPHost returns this error when a configured EAP method does not support a requested operation (procedure call).

GUID_EapHost_Cause_No_SmartCardReader_Found {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x2B}}	A valid smart card needs to be present for authentication to be proceed. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7 .
GUID_EapHost_Cause_Generic_AuthFailure {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 1, 4}}	EAPHost returns this error on a generic, unspecified authentication failure.
GUID_EapHost_Cause_Server_CertOtherError {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 1, 8}}	An unknown error occurred with the server certificate.
GUID_EapHost_Cause_User_Account_OtherProblem {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 1, 0xE}}	An EAP failure was received after an identity exchange, indicating the likelihood of a problem with the authenticating user's account.
GUID_EapHost_Cause_Server_Root_CertNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 1, 0x12}}	EAPHost could not find a root certificate in a trusted root certificate store for the server certification validation.
GUID_EapHost_Cause_IdentityUnknown {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 2, 4}}	EAPHost returns this error if the authenticator fails the authentication after the peer identity was submitted.
GUID_EapHost_Cause_User_CredsRejected {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 2, 0xE}}	The authenticator rejected user credentials for authentication.
GUID_EapHost_Cause_ThirdPartyMethod_Host_Reset {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 2, 0x12}}	The host of the third party method is not responding and was automatically restarted.
GUID_EapHost_Cause_EapQecInaccessible {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 3, 0x12}}	EAPHost was not able to communicate with the EAP quarantine enforcement client (QEC) on a client with Network Access Protection (NAP) enabled. This error may occur when the NAP service is not responding.
GUID_EapHost_Cause_Method_Config_Does_Not_Support_Sso {0xda18bd32, 0x004f, 0x41fa, {0xae, 0x08, 0x0b, 0xc8, 0x5e, 0x58, 0x45, 0xac}}	The EAP method does not support single signon for the provided configuration data. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.

`repairGuid`

A unique ID that maps to a localizable string that identifies the repair action that can be taken to fix the reported

error. The existing values for this member are defined in the *Eaphosterror.h* header file.

VALUE	MEANING
GUID_EapHost_Repair_ContactSysadmin {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 2}}	The user should contact the network administrator.
GUID_EapHost_Repair_Server_ClientSelectServerCert {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x18}}	The user should choose a different and valid certificate for authentication with this network.
GUID_EapHost_Repair_User_AuthFailure {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x19}}	The user should contact your network administrator. Your administrator can verify your user name and password for network authentication.
GUID_EapHost_Repair_User_GetNewCert {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1A}}	The user should obtain an updated certificate from the network administrator. The certificate required to connect to this network can't be found on your computer.
GUID_EapHost_Repair_User_SelectValidCert {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1B}}	The user should use a different and valid user certificate for authentication with the network.
GUID_EapHost_Repair_ContactAdmin_AuthFailure {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x1F}}	The user should contact your network administrator. Windows can't verify your identity for connection to this network. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_IdentityUnknown {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x20}}	The user should contact your network administrator. Windows can't verify your identity for connection to this network. This GUID is supported on Windows Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_NegotiationFailed {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x21}}	The user should contact your network administrator. Windows needs to be configured to use the authentication method required for this network. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.

GUID_EapHost_Repair_ContactAdmin_MethodNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x22}}	The user should contact your network administrator. Windows needs to be configured to use the authentication method required for this network. This GUID is supported on Windows Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_RestartNap {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x23}}	The user should start the Network Access Protection service. The Network Access Protection service is not responding. Start or restart the Network Access Protection service, and then try connecting again. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7 .
GUID_EapHost_Repair_ContactAdmin_CertStoreInaccessible {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x24}}	The user should contact your network administrator. The certificate store on this computer needs to be repaired. This GUID is supported on Windows Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_InvalidUserAccount {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x25}}	The user should contact your network administrator. A problem with your user account needs to be resolved. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_RootCertInvalid {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x26}}	The user should contact your network administrator. The root certificate used for this network needs to be repaired. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_RootCertNotFound {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x27}}	The user should contact your network administrator. The certificate used by the server for this network needs to be properly installed on your computer. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_RootExpired {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x28}}	The user should contact your network administrator. The root certificate used for this network needs to be renewed. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_CertNameAbsent {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x29}}	The user should contact your network administrator. A problem with the server certificate used for this network needs to be resolved. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.

GUID_EapHost_Repair_ContactAdmin_NoSmartCardReader {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x2A}}	The user should connect a smart card reader to your computer, insert a smart card, and attempt to connect again. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_ContactAdmin_InvalidUserCert {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x2C}}	The user should contact your network administrator. The user certificate on this computer needs to be repaired. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_Method_Not_Support_Sso {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x2D}}	The user should contact your network administrator. Windows needs to be configured to use the authentication method required for this network. This GUID is supported on Windows Server 2008 R2 with the Wireless LAN Service installed and on Windows 7.
GUID_EapHost_Repair_Retry_Authentication {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 1, 0x1B}}	The user should try to connect to the network again.

`helpLinkGuid`

A unique ID that maps to a localizable string that specifies an URL for a page that contains additional information about an error or repair message. An EAP method can potentially define a new GUID and associate with one specific help link. Some of the existing values for this member are defined in the *Eaphosterror.h* header file.

VALUE	MEANING
GUID_EapHost_Help_Troubleshooting {0x33307acf, 0x0698, 0x41ba, {0xb0, 0x14, 0xea, 0x0a, 0x2e, 0xb8, 0xd0, 0xa8}}	The URL for the page with more information about troubleshooting. This currently is a generic networking troubleshooting help page, not EAP specific.
GUID_EapHost_Help_EapConfigureTypes {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x03}}	The URL for the page with more information about configuring EAP types.
GUID_EapHost_Help_FailedAuth {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x13}}	The URL for the page with more information about authentication failures. This GUID is supported on Windows Vista
GUID_EapHost_Help_SelectingCerts {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x15}}	The URL for the page with more information about selecting the appropriate certificate to use for authentication.

GUID_EapHost_Help_SetupEapServer {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x16}}	The URL for the page with more information about setting up an EAP server. This GUID is supported on Windows Vista
GUID_EapHost_Help_Troubleshooting {0x9612fc67, 0x6150, 0x4209, {0xa8, 0x5e, 0xa8, 0xd8, 0, 0, 0, 0x17}}	The URL for the page with more information about troubleshooting. This GUID is supported on Windows Vista
GUID_EapHost_Help_ObtainingCerts {0xf535eea3, 0x1bdd, 0x46ca, {0xa2, 0xfc, 0xa6, 0x65, 0x59, 0x39, 0xb7, 0xe8}}	The URL for the page with more information about getting EAP certificates.

`fRootCauseString`

Indicates if the **ONEX_EAP_ERROR** structure contains a root cause string in the **RootCauseString** member.

`fRepairString`

Indicates if the **ONEX_EAP_ERROR** structure contains a repair string in the **RepairString** member.

`RootCauseString`

A localized and readable string that describes the root cause of the error. This member contains a NULL-terminated Unicode string starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fRootCauseString** bitfield member is set.

`RepairString`

A localized and readable string that describes the possible repair action. This member contains a NULL-terminated Unicode string starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fRepairString** bitfield member is set.

Remarks

The **ONEX_EAP_ERROR** structure is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

Many members of the **ONEX_EAP_ERROR** structure correspond with similar members in the EAP_ERROR structure

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

If the **fEapError** member in the ONEX_RESULT_UPDATE_DATA structure is set, then the **eapError** member of the **ONEX_RESULT_UPDATE_DATA** structure contains an ONEX_VARIABLE_BLOB structure with an **ONEX_EAP_ERROR** structure embedded starting at the **dwOffset** member of the **ONEX_VARIABLE_BLOB**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

Common EAPHost API Structures

EAP_ERROR

EAP_METHOD_TYPE

ONEX_NOTIFICATION_TYPE

ONEX_RESULT_UPDATE_DATA

ONEX_VARIABLE_BLOB

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_EAP_METHOD_BACKEND_SUPPORT enumeration (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_EAP_METHOD_BACKEND_SUPPORT** enumerated type specifies the possible values for whether the EAP method configured on the supplicant for 802.1X authentication is supported on the authentication server.

Syntax

```
typedef enum _ONEX_EAP_METHOD_BACKEND_SUPPORT {
  OneXEapMethodBackendSupportUnknown,
  OneXEapMethodBackendSupported,
  OneXEapMethodBackendUnsupported
} ONEX_EAP_METHOD_BACKEND_SUPPORT;
```

Constants

`OneXEapMethodBackendSupportUnknown`	It is not known whether the EAP method configured on the supplicant for 802.1X authentication is supported on the authentication server. This value can be returned if the 802.1X authentication process is in the initial state.
`OneXEapMethodBackendSupported`	The EAP method configured on the supplicant for 802.1X authentication is supported on the authentication server. The 802.1X handshake is used to decide what is an acceptable EAP method to use.
`OneXEapMethodBackendUnsupported`	The EAP method configured on the supplicant for 802.1X authentication is not supported on the authentication server.

Remarks

The **ONEX_EAP_METHOD_BACKEND_SUPPORT** enumeration is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

The **BackendSupport** member of the ONEX_RESULT_UPDATE_DATA struct contains a value from the **ONEX_EAP_METHOD_BACKEND_SUPPORT** enumeration.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_NOTIFICATION_TYPE

ONEX_RESULT_UPDATE_DATA

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_NOTIFICATION_TYPE enumeration (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_NOTIFICATION_TYPE** enumerated type specifies the possible values of the **NotificationCode** member of the WLAN_NOTIFICATION_DATA structure for 802.1X module notifications.

Syntax

```
typedef enum _ONEX_NOTIFICATION_TYPE {
  OneXPublicNotificationBase,
  OneXNotificationTypeResultUpdate,
  OneXNotificationTypeAuthRestarted,
  OneXNotificationTypeEventInvalid,
  OneXNumNotifications
} ONEX_NOTIFICATION_TYPE, PONEX_NOTIFICATION_TYPE;
```

Constants

`OneXPublicNotificationBase`
Indicates the beginning of the range that specifies the possible values for 802.1X notifications.

`OneXNotificationTypeResultUpdate`
Indicates that 802.1X authentication has a status change.

The **pData** member of the WLAN_NOTIFICATION_DATA structure points to a ONEX_RESULT_UPDATE_DATA structure that contains 802.1X update data.

`OneXNotificationTypeAuthRestarted`
Indicates that 802.1X authentication restarted.

The **pData** member of the WLAN_NOTIFICATION_DATA structure points to an ONEX_AUTH_RESTART_REASON enumeration value that identifies the reason the authentication was restarted.

`OneXNotificationTypeEventInvalid`
Indicates the end of the range that specifies the possible values for 802.1X notifications.

`OneXNumNotifications`
Indicates the end of the range that specifies the possible values for 802.1X notifications.

Remarks

The **ONEX_NOTIFICATION_TYPE** enumerated type is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The **ONEX_NOTIFICATION_TYPE** specifies the possible values for the **NotificationCode** member of the WLAN_NOTIFICATION_DATA structure for received notifications when the **NotificationSource** member of the **WLAN_NOTIFICATION_DATA** structure is **WLAN_NOTIFICATION_SOURCE_ONEX**.

The WlanRegisterNotification function is used by an application to register and unregister notifications on all

wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter passed to the **WlanRegisterNotification** function. The prototype for this callback function is the WLAN_NOTIFICATION_CALLBACK. This callback function will receive notifications that have been registered in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function.

The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter that contains detailed information on the notification. The callback function also receives a second parameter that contains a pointer to the client context passed in the *pCallbackContext* parameter to the WlanRegisterNotification function. This client context can be a **NULL** pointer if that is what was passed to the **WlanRegisterNotification** function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_AUTH_RESTART_REASON

ONEX_RESULT_UPDATE_DATA

WLAN_NOTIFICATION_CALLBACK

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_REASON_CODE enumeration (dot1x.h)

6/2/2021 • 4 minutes to read • Edit Online

The **ONEX_REASON_CODE** enumerated type specifies the possible values that indicate the reason that 802.1X authentication failed.

Syntax

```
typedef enum _ONEX_REASON_CODE {
  ONEX_REASON_CODE_SUCCESS,
  ONEX_REASON_START,
  ONEX_UNABLE_TO_IDENTIFY_USER,
  ONEX_IDENTITY_NOT_FOUND,
  ONEX_UI_DISABLED,
  ONEX_UI_FAILURE,
  ONEX_EAP_FAILURE_RECEIVED,
  ONEX_AUTHENTICATOR_NO_LONGER_PRESENT,
  ONEX_NO_RESPONSE_TO_IDENTITY,
  ONEX_PROFILE_VERSION_NOT_SUPPORTED,
  ONEX_PROFILE_INVALID_LENGTH,
  ONEX_PROFILE_DISALLOWED_EAP_TYPE,
  ONEX_PROFILE_INVALID_EAP_TYPE_OR_FLAG,
  ONEX_PROFILE_INVALID_ONEX_FLAGS,
  ONEX_PROFILE_INVALID_TIMER_VALUE,
  ONEX_PROFILE_INVALID_SUPPLICANT_MODE,
  ONEX_PROFILE_INVALID_AUTH_MODE,
  ONEX_PROFILE_INVALID_EAP_CONNECTION_PROPERTIES,
  ONEX_UI_CANCELLED,
  ONEX_PROFILE_INVALID_EXPLICIT_CREDENTIALS,
  ONEX_PROFILE_EXPIRED_EXPLICIT_CREDENTIALS,
  ONEX_UI_NOT_PERMITTED
} ONEX_REASON_CODE, PONEX_REASON_CODE;
```

Constants

`ONEX_REASON_CODE_SUCCESS`	Indicates the 802.1X authentication was a success.
`ONEX_REASON_START`	Indicates the start of the range that specifies the possible values for 802.1X reason code.
`ONEX_UNABLE_TO_IDENTIFY_USER`	The 802.1X module was unable to identify a set of credentials to be used. An example is when the authentication mode is set to user, but no user is logged on.
`ONEX_IDENTITY_NOT_FOUND`	The EAP module was unable to acquire an identity for the user. Thus value is not currently used. All EAP-specific errors are returned as **ONEX_EAP_FAILURE_RECEIVED**.

`ONEX_UI_DISABLED` To proceed with 802.1X authentication, the system needs to request user input, but the user interface is disabled. On Windows Vista and on Windows Server 2008, this value can be returned if an EAP method requested user input for a profile for Guest or local machine authentication. On Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed, this value should not be returned.
`ONEX_UI_FAILURE` The 802.1X authentication module was unable to return the requested user input. On Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed, this value can be returned if an EAP method requested user input, but the UI could not be displayed (the network icon was configured to not show in the taskbar, for example).
`ONEX_EAP_FAILURE_RECEIVED` The EAP module returned an error code. The ONEX_EAP_ERROR structure may contain additional information about the specific EAP error (a certificate not found, for example).
`ONEX_AUTHENTICATOR_NO_LONGER_PRESENT` The peer with which the 802.1X module was negotiating is no longer present or is not responding (a laptop client moved out of range of the wireless access point, for example).
`ONEX_NO_RESPONSE_TO_IDENTITY` No response was received to an EAP identity response packet. This value indicates a problem with the infrastructure (a link between the wireless access point and the authentication server is not functioning, for example).
`ONEX_PROFILE_VERSION_NOT_SUPPORTED` The 802.1X module does not support this version of the profile.
`ONEX_PROFILE_INVALID_LENGTH` The length member specified in the 802.1X profile is invalid.
`ONEX_PROFILE_DISALLOWED_EAP_TYPE` The EAP type specified in the 802.1X profile is not allowed for this media. An example is when the keyed MD5 algorithm is used for wireless transmission.
`ONEX_PROFILE_INVALID_EAP_TYPE_OR_FLAG` The EAP type or EAP flags specified in the 802.1X profile are not valid. An example is when EAP type is not installed on the system.
`ONEX_PROFILE_INVALID_ONEX_FLAGS` The 802.1X flags specified in the 802.1X profile are not valid.
`ONEX_PROFILE_INVALID_TIMER_VALUE` One or more timer values specified in the 802.1X profile is out of its valid range.
`ONEX_PROFILE_INVALID_SUPPLICANT_MODE` The supplicant mode specified in the 802.1X profile is not valid.
`ONEX_PROFILE_INVALID_AUTH_MODE` The authentication mode specified in the 802.1X profile is not valid.
`ONEX_PROFILE_INVALID_EAP_CONNECTION_PROPERTIES` The EAP connection properties specified in the 802.1X profile are not valid.

`ONEX_UI_CANCELLED` User input was canceled. This value can be returned if an EAP method requested user input, but the user hit the Cancel button or dismissed the user input dialog. This value is supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.
`ONEX_PROFILE_INVALID_EXPLICIT_CREDENTIALS` The saved user credentials are not valid. This value can be returned if a profile was saved with bad credentials (an incorrect password, for example), since the credentials are not tested until the profile is actually used to establish a connection. This value is supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.
`ONEX_PROFILE_EXPIRED_EXPLICIT_CREDENTIALS` The saved user credentials have expired. This value can be returned if a profile was saved with credentials and the credentials subsequently expired (password expiration after some period of time, for example). This value is supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.
`ONEX_UI_NOT_PERMITTED` User interface is not permitted. On Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed, this value can be returned if an EAP method requested user input and the profile is configured with user credentials saved by another user and not the currently logged in user. This value is supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

Remarks

The **ONEX_REASON_CODE** enumerated type is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

The **oneXStatus** member of the ONEX_RESULT_UPDATE_DATA structure contains an ONEX_STATUS structure. If an error occurred during 802.1X authentication, the dwReason member of this **ONEX_STATUS** structure contains the reason for the error specified as a value from the **ONEX_REASON_CODE** enumeration.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

ONEX_RESULT_UPDATE_DATA structure (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_RESULT_UPDATE_DATA** structure contains information on a status change to 802.1X authentication.

Syntax

```
typedef struct _ONEX_RESULT_UPDATE_DATA {
  ONEX_STATUS                     oneXStatus;
  ONEX_EAP_METHOD_BACKEND_SUPPORT BackendSupport;
  BOOL                            fBackendEngaged;
  DWORD                           fOneXAuthParams : 1;
  DWORD                           fEapError : 1;
  ONEX_VARIABLE_BLOB              authParams;
  ONEX_VARIABLE_BLOB              eapError;
} ONEX_RESULT_UPDATE_DATA, *PONEX_RESULT_UPDATE_DATA;
```

Members

`oneXStatus`

Specifies the current 802.1X authentication status. For more information, see the ONEX_STATUS structure.

`BackendSupport`

Indicates if the configured EAP method on the supplicant is supported on the 802.1X authentication server.

EAP permits the use of a backend authentication server, which may implement some or all authentication methods, with the authenticator acting as a pass-through for some or all methods and peers. For more information, see RFC 3748 published by the IETF and the ONEX_EAP_METHOD_BACKEND_SUPPORT enumeration.

`fBackendEngaged`

Indicates if a response was received from the 802.1X authentication server.

`fOneXAuthParams`

Indicates if the **ONEX_RESULT_UPDATE_DATA** structure contains 802.1X authentication parameters in the **authParams** member.

`fEapError`

Indicates if the **ONEX_RESULT_UPDATE_DATA** structure contains an EAP error in the **eapError** member.

`authParams`

The 802.1X authentication parameters. This member contains an embedded ONEX_AUTH_PARAMS structure starting at the **dwOffset** member of the ONEX_VARIABLE_BLOB if the **fOneXAuthParams** bitfield member is set.

`eapError`

An EAP error value. This member contains an embedded ONEX_EAP_ERROR structure starting at the **dwOffset**

member of the ONEX_VARIABLE_BLOB if the **fEapError** bitfield member is set.

Remarks

The **ONEX_RESULT_UPDATE_DATA** structure is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The **ONEX_RESULT_UPDATE_DATA** contains information on a status change to 802.1X authentication.This structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_AUTH_PARAMS

ONEX_EAP_ERROR

ONEX_EAP_METHOD_BACKEND_SUPPORT

ONEX_NOTIFICATION_TYPE

ONEX_STATUS

ONEX_VARIABLE_BLOB

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_STATUS structure (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_STATUS** structure contains the current 802.1X authentication status.

Syntax

```
typedef struct _ONEX_STATUS {
  ONEX_AUTH_STATUS authStatus;
  DWORD            dwReason;
  DWORD            dwError;
} ONEX_STATUS, *PONEX_STATUS;
```

Members

`authStatus`

The current status of the 802.1X authentication process. Any error that may have occurred during authentication is indicated below by the value of the **dwReason** and **dwError** members of the **ONEX_STATUS** structure. For more information, see the ONEX_AUTH_STATUS enumeration.

`dwReason`

If an error occurred during 802.1X authentication, this member contains the reason for the error specified as a value from the ONEX_REASON_CODE enumeration. This member is normally **ONEX_REASON_CODE_SUCCESS** when 802.1X authentication is successful and no error occurs.

`dwError`

If an error occurred during 802.1X authentication, this member contains the error. This member is normally NO_ERROR, except when an EAPHost error occurs.

Remarks

The **ONEX_STATUS** structure is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

The **oneXStatus** member of the ONEX_RESULT_UPDATE_DATA structure contains an **ONEX_STATUS** structure.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_NOTIFICATION_TYPE

ONEX_REASON_CODE

ONEX_RESULT_UPDATE_DATA

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

ONEX_VARIABLE_BLOB structure (dot1x.h)

6/2/2021 • 2 minutes to read • Edit Online

The **ONEX_VARIABLE_BLOB** structure is used as a member of other 802.1X authentication structures to contain variable-sized members.

Syntax

```
typedef struct _ONEX_VARIABLE_BLOB {
  DWORD dwSize;
  DWORD dwOffset;
} ONEX_VARIABLE_BLOB, *PONEX_VARIABLE_BLOB;
```

Members

`dwSize`

The size, in bytes, of this **ONEX_VARIABLE_BLOB** structure.

`dwOffset`

The offset, in bytes, from the beginning of the containing outer structure (where the **ONEX_VARIABLE_BLOB** structure is a member) to the data contained in the **ONEX_VARIABLE_BLOB** structure.

Remarks

The **ONEX_VARIABLE_BLOB** structure is used by the 802.1X module, a new wireless configuration component supported on Windows Vista and later.

The ONEX_RESULT_UPDATE_DATA contains information on a status change to 802.1X authentication. The **ONEX_RESULT_UPDATE_DATA** structure is returned when the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure is **WLAN_NOTIFICATION_SOURCE_ONEX** and the **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure for received notification is **OneXNotificationTypeResultUpdate**. For this notification, the **pData** member of the **WLAN_NOTIFICATION_DATA** structure points to an **ONEX_RESULT_UPDATE_DATA** structure that contains information on the 802.1X authentication status change.

A number of the nested structure members in the ONEX_RESULT_UPDATE_DATA structure contains members of the **ONEX_VARIABLE_BLOB** type.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	dot1x.h

See also

About the ACM Architecture

ONEX_AUTH_PARAMS

ONEX_EAP_ERROR

ONEX_NOTIFICATION_TYPE

ONEX_RESULT_UPDATE_DATA

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

wlanapi.h header

7/1/2021 • 6 minutes to read • Edit Online

This header is used by Native Wifi. For more information, see:

- Native Wifi

wlanapi.h contains the following programming interfaces:

Functions

WFDCancelOpenSession	Indicates that the application wants to cancel a pending WFDStartOpenSession function that has not completed.
WFDCloseHandle	Closes a handle to the Wi-Fi Direct service.
WFDCloseSession	Closes a session after a previously successful call to the WFDStartOpenSession function.
WFDOpenHandle	Opens a handle to the Wi-Fi Direct service and negotiates a version of the Wi-FI Direct API to use.
WFDOpenLegacySession	Retrieves and applies a stored profile for a Wi-Fi Direct legacy device.
WFDStartOpenSession	Starts an on-demand connection to a specific Wi-Fi Direct device, which has been previously paired through the Windows Pairing experience.
WFDUpdateDeviceVisibility	Updates device visibility for the Wi-Fi Direct device address for a given installed Wi-Fi Direct device node.
WlanAllocateMemory	Allocates memory.
WlanCloseHandle	Closes a connection to the server.
WlanConnect	Attempts to connect to a specific network.

WlanDeleteProfile	Deletes a wireless profile for a wireless interface on the local computer.
WlanDeviceServiceCommand	Allows an OEM or IHV component to communicate with a device service on a particular wireless LAN interface.
WlanDisconnect	Disconnects an interface from its current network.
WlanEnumInterfaces	Enumerates all of the wireless LAN interfaces currently enabled on the local computer.
WlanExtractPsdIEDataList	Extracts the proximity service discovery (PSD) information element (IE) data list from raw IE data included in a beacon.
WlanFreeMemory	Frees memory.
WlanGetAvailableNetworkList	Retrieves the list of available networks on a wireless LAN interface.
WlanGetFilterList	Retrieves a group policy or user permission list.
WlanGetInterfaceCapability	Retrieves the capabilities of an interface.
WlanGetNetworkBssList	Retrieves a list of the basic service set (BSS) entries of the wireless network or networks on a given wireless LAN interface.
WlanGetProfile	Retrieves all information about a specified wireless profile.
WlanGetProfileCustomUserData	Gets the custom user data associated with a wireless profile.
WlanGetProfileList	Retrieves the list of profiles.
WlanGetSecuritySettings	Gets the security settings associated with a configurable object.

WlanGetSupportedDeviceServices	Retrieves a list of the supported device services on a given wireless LAN interface.
WlanHostedNetworkForceStart	Transitions the wireless Hosted Network to the wlan_hosted_network_active state without associating the request with the application's calling handle.
WlanHostedNetworkForceStop	Transitions the wireless Hosted Network to the wlan_hosted_network_idle without associating the request with the application's calling handle.
WlanHostedNetworkInitSettings	Configures and persists to storage the network connection settings (SSID and maximum number of peers, for example) on the wireless Hosted Network if these settings are not already configured.
WlanHostedNetworkQueryProperty	Queries the current static properties of the wireless Hosted Network.
WlanHostedNetworkQuerySecondaryKey	Queries the secondary security key that is configured to be used by the wireless Hosted Network.
WlanHostedNetworkQueryStatus	Queries the current status of the wireless Hosted Network.
WlanHostedNetworkRefreshSecuritySettings	Refreshes the configurable and auto-generated parts of the wireless Hosted Network security settings.
WlanHostedNetworkSetProperty	Sets static properties of the wireless Hosted Network.
WlanHostedNetworkSetSecondaryKey	Configures the secondary security key that will be used by the wireless Hosted Network.
WlanHostedNetworkStartUsing	Starts the wireless Hosted Network.
WlanHostedNetworkStopUsing	Stops the wireless Hosted Network.
WlanIhvControl	Provides a mechanism for independent hardware vendor (IHV) control of WLAN drivers or services.

WlanOpenHandle	Opens a connection to the server.
WlanQueryAutoConfigParameter	Queries for the parameters of the auto configuration service.
WlanQueryInterface	The WlanQueryInterface function queries various parameters of a specified interface.
WlanReasonCodeToString	Retrieves a string that describes a specified reason code.
WlanRegisterDeviceServiceNotification	Allows user mode clients with admin privileges, or User-Mode Driver Framework (UMDF) drivers, to register for unsolicited notifications corresponding to device services that they're interested in.
WlanRegisterNotification	Is used to register and unregister notifications on all wireless interfaces.
WlanRegisterVirtualStationNotification	Is used to register and unregister notifications on a virtual station.
WlanRenameProfile	Renames the specified profile.
WlanSaveTemporaryProfile	Saves a temporary profile to the profile store.
WlanScan	Requests a scan for available networks on the indicated interface.
WlanSetAutoConfigParameter	Sets parameters for the automatic configuration service.
WlanSetFilterList	Sets the permit/deny list.
WlanSetInterface	Sets user-configurable parameters.

WlanSetProfile Sets the content of a specific profile.
WlanSetProfileCustomUserData Sets the custom user data associated with a profile.
WlanSetProfileEapUserData Sets the Extensible Authentication Protocol (EAP) user credentials as specified by raw EAP data.
WlanSetProfileEapXmlUserData Sets the Extensible Authentication Protocol (EAP) user credentials as specified by an XML string.
WlanSetProfileList Sets the preference order of profiles.
WlanSetProfilePosition Sets the position of a single, specified profile in the preference list.
WlanSetPsdIEDataList Sets the proximity service discovery (PSD) information element (IE) data list.
WlanSetSecuritySettings Sets the security settings for a configurable object.
WlanUIEditProfile Displays the wireless profile user interface (UI).

Callback functions

WFD_OPEN_SESSION_COMPLETE_CALLBACK Defines the callback function that is called by the WFDStartOpenSession function when the WFDStartOpenSession operation completes.
WLAN_NOTIFICATION_CALLBACK Defines the type of notification callback function.

Structures

DOT11_NETWORK	Contains information about an available wireless network.
DOT11_NETWORK_LIST	Contains a list of 802.11 wireless networks.
WLAN_ASSOCIATION_ATTRIBUTES	Contains association attributes for a connection.
WLAN_AUTH_CIPHER_PAIR_LIST	Contains a list of authentication and cipher algorithm pairs.
WLAN_AVAILABLE_NETWORK	Contains information about an available wireless network.
WLAN_AVAILABLE_NETWORK_LIST	Contains an array of information about available networks.
WLAN_BSS_ENTRY	Contains information about a basic service set (BSS).
WLAN_BSS_LIST	Contains a list of basic service set (BSS) entries.
WLAN_CONNECTION_ATTRIBUTES	Defines the attributes of a wireless connection.
WLAN_CONNECTION_NOTIFICATION_DATA	Contains information about connection related notifications.
WLAN_CONNECTION_PARAMETERS	Specifies the parameters used when using the WlanConnect function.
WLAN_COUNTRY_OR_REGION_STRING_LIST	Contains a list of supported country or region strings.
WLAN_DEVICE_SERVICE_GUID_LIST	Contains an array of device service GUIDs.
WLAN_DEVICE_SERVICE_NOTIFICATION_DATA	A structure that represents a device service notification.

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS

Contains information about the connection settings on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE

Contains information about a network state change for a data peer on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_PEER_STATE

Contains information about the peer state for a peer on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_RADIO_STATE

Contains information about the radio state on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_SECURITY_SETTINGS

Contains information about the security settings on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_STATE_CHANGE

Contains information about a network state change on the wireless Hosted Network.

WLAN_HOSTED_NETWORK_STATUS

Contains information about the status of the wireless Hosted Network.

WLAN_INTERFACE_CAPABILITY

Contains information about the capabilities of an interface.

WLAN_INTERFACE_INFO

Contains information about a wireless LAN interface.

WLAN_INTERFACE_INFO_LIST

Array of NIC interface information.

WLAN_MAC_FRAME_STATISTICS

Contains information about sent and received MAC frames.

WLAN_MSM_NOTIFICATION_DATA

Contains information about media specific module (MSM) connection related notifications.

WLAN_PHY_FRAME_STATISTICS

Contains information about sent and received PHY frames.

WLAN_PHY_RADIO_STATE

Specifies the radio state.

WLAN_PROFILE_INFO Basic information about a profile.
WLAN_PROFILE_INFO_LIST Contains a list of wireless profile information.
WLAN_RADIO_STATE Specifies the radio state on a list of physical layer (PHY) types.
WLAN_RATE_SET The set of supported data rates.
WLAN_RAW_DATA Contains raw data in the form of a blob that is used by some Native Wifi functions.
WLAN_RAW_DATA_LIST Contains raw data in the form of an array of data blobs that are used by some Native Wifi functions.
WLAN_SECURITY_ATTRIBUTES Defines the security attributes for a wireless connection.
WLAN_STATISTICS Assorted statistics about an interface.

Enumerations

DOT11_RADIO_STATE
WL_DISPLAY_PAGES Specifies the active tab when the wireless profile user interface dialog box appears.
WLAN_ADHOC_NETWORK_STATE
WLAN_AUTOCONF_OPCODE
WLAN_CONNECTION_MODE Defines the mode of connection.

WLAN_FILTER_LIST_TYPE Indicates types of filter lists.
WLAN_HOSTED_NETWORK_NOTIFICATION_CODE Specifies the possible values of the NotificationCode parameter for received notifications on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_OPCODE Specifies the possible values of the operation code for the properties to query or set on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_PEER_AUTH_STATE Specifies the possible values for the authentication state of a peer on the wireless Hosted Network.
WLAN_HOSTED_NETWORK_REASON Specifies the possible values for the result of a wireless Hosted Network function call.
WLAN_HOSTED_NETWORK_STATE Specifies the possible values for the network state of the wireless Hosted Network.
WLAN_IHV_CONTROL_TYPE
WLAN_INTERFACE_STATE
WLAN_INTERFACE_TYPE Specifies the wireless interface type.
WLAN_INTF_OPCODE
WLAN_NOTIFICATION_ACM
WLAN_NOTIFICATION_MSM
WLAN_OPCODE_VALUE_TYPE
WLAN_POWER_SETTING
WLAN_SECURABLE_OBJECT Defines the securable objects used by Native Wifi Functions.

DOT11_NETWORK structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **DOT11_NETWORK** structure contains information about an available wireless network.

Syntax

```
typedef struct _DOT11_NETWORK {
  DOT11_SSID     dot11Ssid;
  DOT11_BSS_TYPE dot11BssType;
} DOT11_NETWORK, *PDOT11_NETWORK;
```

Members

`dot11Ssid`

A DOT11_SSID structure that contains the SSID of a visible wireless network.

`dot11BssType`

A DOT11_BSS_TYPE value that indicates the BSS type of the network.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

DOT11_NETWORK_LIST

DOT11_NETWORK_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **DOT11_NETWORK_LIST** structure contains a list of 802.11 wireless networks.

Syntax

```
typedef struct _DOT11_NETWORK_LIST {
  DWORD         dwNumberOfItems;
  DWORD         dwIndex;
#if ...
  DOT11_NETWORK *Network[];
#else
  DOT11_NETWORK Network[1];
#endif
} DOT11_NETWORK_LIST, *PDOT11_NETWORK_LIST;
```

Members

`dwNumberOfItems`

Contains the number of items in the **Network** member.

`dwIndex`

The index of the current item. The index of the first item is 0. **dwIndex** must be less than **dwNumberOfItems**.

This member is not used by the wireless service. Applications can use this member when processing individual networks in the **DOT11_NETWORK_LIST** structure. When an application passes this structure from one function to another, it can set the value of **dwIndex** to the index of the item currently being processed. This can help an application maintain state.

dwIndex should always be initialized before use.

`Network`

An array of DOT11_NETWORK structures that contain 802.11 wireless network information.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_NETWORK

WlanGetFilterList

WlanSetFilterList

WFD_OPEN_SESSION_COMPLETE_CALLBACK callback function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFD_OPEN_SESSION_COMPLETE_CALLBACK** function defines the callback function that is called by the WFDStartOpenSession function when the **WFDStartOpenSession** operation completes.

Syntax

```
WFD_OPEN_SESSION_COMPLETE_CALLBACK WfdOpenSessionCompleteCallback;

void WfdOpenSessionCompleteCallback(
  HANDLE hSessionHandle,
  PVOID pvContext,
  GUID guidSessionInterface,
  DWORD dwError,
  DWORD dwReasonCode
)
{...}
```

Parameters

`hSessionHandle`

A session handle to a Wi-Fi Direct session. This is a session handle previously returned by the WFDStartOpenSession function.

`pvContext`

An context pointer passed to the callback function from the WFDStartOpenSession function.

`guidSessionInterface`

The interface GUID of the local network interface on which this Wi-Fi Direct device has an open session. This parameter is useful if higher-layer protocols need to determine which network interface a Wi-Fi Direct session is bound to. This value is only returned if the *dwError* parameter is ERROR_SUCCESS.

`dwError`

A value that specifies whether there was an error encountered during the call to the WFDStartOpenSession function. If this value is ERROR_SUCCESS, then no error occurred and the operation to open the session completed successfully.

The following other values are possible:

VALUE	MEANING
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *hClientHandle* parameter is **NULL** or not valid.

ERROR_INVALID_STATE	The group or resource is not in the correct state to perform the requested operation. This error is returned if the Wi-Fi Direct service is disabled by group policy on a domain.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
RPC_STATUS	Various RPC and other error codes. Use **FormatMessage** to obtain the message string for the returned error.

`dwReasonCode`

A value that specifies the more detail if an error occurred during WFDStartOpenSession.

Return value

None

Remarks

The **WFD_OPEN_SESSION_COMPLETE_CALLBACK** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

The WFDStartOpenSession function starts an asynchronous operation to start an on-demand connection to a specific Wi-Fi Direct device. The target Wi-Fi device must previously have been paired through the Windows Pairing experience. When the asynchronous operation to make the Wi-FI Direct connection completes, the callback function specified in the *pfnCallback* parameter is called.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h

See also

WFDCancelOpenSession

WFDCloseHandle

WFDCloseSession

WFDOpenHandle

WFDStartOpenSession

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDCancelOpenSession function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDCancelOpenSession** function indicates that the application wants to cancel a pending WFDStartOpenSession function that has not completed.

Syntax

```
DWORD WFDCancelOpenSession(
  HANDLE hSessionHandle
);
```

Parameters

`hSessionHandle`

A session handle to a Wi-Fi Direct session to cancel. This is a session handle previously returned by the WFDStartOpenSession function.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle is invalid. This error is returned if the handle specified in the *hSessionHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *hSessionHandle* parameter is **NULL** or not valid.
RPC_STATUS	Various error codes.

Remarks

The **WFDCancelOpenSession** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

A call to the **WFDCancelOpenSession** function notifies the Wi-Fi Direct service that the client requests a

cancellation of this session. The **WFDCancelOpenSession** function does not modify the expected WFDStartOpenSession behavior. The callback function specified to the **WFDStartOpenSession** function will still be called, and the **WFDStartOpenSession** function may not be completed immediately.

It is the responsibility of the caller to pass the **WFDCancelOpenSession** function a handle in the *hSessionHandle*parameter that was returned from call to the WFDStartOpenSession function.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCloseHandle

WFDCloseSession

WFDOpenHandle

WFDOpenLegacySession

WFDStartOpenSession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDCloseHandle function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDCloseHandle** function closes a handle to the Wi-Fi Direct service.

Syntax

```
DWORD WFDCloseHandle(
  HANDLE hClientHandle
);
```

Parameters

`hClientHandle`

A client handle to the Wi-Fi Direct service. This handle was obtained by a previous call to the WFDOpenHandle function.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *hClientHandle* parameter is **NULL** or not valid.
RPC_STATUS	Various error codes.

Remarks

The **WFDCloseHandle** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

In order to use Wi-Fi Direct, an application must first obtain a handle to the Wi-Fi Direct service by calling the WFDOpenHandle function. The Wi-Fi Direct (WFD) handle returned by the **WFDOpenHandle** function is used for subsequent calls made to the Wi-Fi Direct service. Once an application is done using the Wi-Fi Direct service,

the application should call the **WFDCloseHandle** function to signal to the Wi-Fi Direct service that the application is done using the service. This allows the Wi-Fi Direct service to release resources used by the application.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseSession

WFDOpenHandle

WFDOpenLegacySession

WFDStartOpenSession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDCloseSession function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDCloseSession** function closes a session after a previously successful call to the WFDStartOpenSession function.

Syntax

```
DWORD WFDCloseSession(
  HANDLE hSessionHandle
);
```

Parameters

`hSessionHandle`

A session handle to a Wi-Fi Direct session. This is a session handle previously returned by the WFDStartOpenSession function.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle is invalid. This error is returned if the handle specified in the *hSessionHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *hSessionHandle* parameter is **NULL** or not valid.
ERROR_INVALID_STATE	The group or resource is not in the correct state to perform the requested operation. This error is returned if the Wi-Fi Direct service is disabled by group policy on a domain.
RPC_STATUS	Various error codes.

Remarks

The **WFDCloseSession** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is

to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

The **WFDCloseSession** function queues a future work item to close the session, so disconnection may not be immediate.

Calling the **WFDCloseSession** function while a WFDStartOpenSession call is pending will not close the session.

It is the responsibility of the caller to pass the **WFDCloseSession** function a handle in the *hSessionHandle* parameter that was returned from a successful asynchronous call to the WFDStartOpenSession function.

Calling the **WFDCloseSession** function with a handle that was valid and has become invalid will yield undefined results.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseHandle

WFDOpenHandle

WFDOpenLegacySession

WFDStartOpenSession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDOpenHandle function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDOpenHandle** function opens a handle to the Wi-Fi Direct service and negotiates a version of the Wi-FI Direct API to use.

Syntax

```
DWORD WFDOpenHandle(
  DWORD   dwClientVersion,
  PDWORD  pdwNegotiatedVersion,
  PHANDLE phClientHandle
);
```

Parameters

`dwClientVersion`

The highest version of the Wi-Fi Direct API the client supports.

For Windows 8 and Windows Server 2012, this parameter should be set to **WFD_API_VERSION**, constant defined in the *Wlanapi.h* header file.

`pdwNegotiatedVersion`

A pointer to a **DWORD** to received the negotiated version.

If the **WFDOpenHandle** function is successful, the version negotiated with the Wi-Fi Direct Service to be used by this session is returned. This value is usually the highest version supported by both the client and Wi-Fi Direct service.

`phClientHandle`

A pointer to a **HANDLE** to receive the handle to the Wi-Fi Direct service for this session.

If the **WFDOpenHandle** function is successful, a handle to the Wi-Fi Direct service to use in this session is returned.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *pdwNegotiatedVersion* parameter is **NULL** or the *phClientHandle* parameter is **NULL**. This value is also returned if the *dwClientVersion* parameter is not equal to **WFD_API_VERSION**.

ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command. This error is returned if the system was unable to allocate memory to create the client context.
ERROR_REMOTE_SESSION_LIMIT_EXCEEDED	An attempt was made to establish a session to a network server, but there are already too many sessions established to that server. This error is returned if too many handles have been issued by the Wi-Fi Direct service.
RPC_STATUS	Various error codes.

Remarks

The **WFDOpenHandle** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

In order to use Wi-Fi Direct, an application must first obtain a handle to the Wi-Fi Direct service by calling the **WFDOpenHandle** function. The Wi-Fi Direct (WFD) handle returned by the **WFDOpenHandle** function is used for subsequent calls made to the Wi-Fi Direct service. Once an application is done using the Wi-Fi Direct service, the application should call the WFDCloseHandle function to signal to the Wi-Fi Direct service that the application is done using the service. This allows the Wi-Fi Direct service to release resources used by the application.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseHandle

WFDCloseSession

WFDOpenLegacySession

WFDStartOpenSession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDOpenLegacySession function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDOpenLegacySession** function retrieves and applies a stored profile for a Wi-Fi Direct legacy device.

Syntax

```
DWORD WFDOpenLegacySession(
  HANDLE             hClientHandle,
  PDOT11_MAC_ADDRESS pLegacyMacAddress,
  HANDLE             *phSessionHandle,
  GUID               *pGuidSessionInterface
);
```

Parameters

`hClientHandle`

A **HANDLE** to the Wi-Fi Direct service for this session. This parameter is retrieved using the WFDOpenHandle function.

`pLegacyMacAddress`

A pointer to Wi-Fi Direct device address of the legacy client device.

`phSessionHandle`

A pointer to a **HANDLE** to receive the handle to the Wi-Fi Direct service for this session.

If the **WFDOpenLegacySession** function is successful, a handle to the Wi-Fi Direct service to use in this session is returned.

`pGuidSessionInterface`

A pointer to the GUID of the network interface for this session.

If the **WFDOpenLegacySession** function is successful, a GUID of the network interface on which Wi-Fi Direct session is returned.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *phClientHandle* or the *pLegacyMacAddress* parameter is **NULL**.

ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command. This error is returned if the system was unable to allocate memory to create the client context.
RPC_STATUS	Various error codes.

Remarks

The **WFDOpenLegacySession** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

In order to use Wi-Fi Direct, an application must first obtain a handle to the Wi-Fi Direct service by calling the **WFDOpenLegacySession** or WFDOpenHandle function. The Wi-Fi Direct (WFD) handle returned by the **WFDOpenHandle** function is used for subsequent calls made to the Wi-Fi Direct service. The **WFDOpenLegacySession** function is used to retrieve and apply a stored profile for a Wi-Fi Direct legacy device.

The **WFDOpenLegacySession** function retrieves the stored legacy profile for device from the profile store for the specified legacy device address. This device address must be obtained from a Device Node created as a result of the Inbox pairing experience (Legacy WPS Pairing).

Once an application is done using the Wi-Fi Direct service, the application should call the WFDCloseSession function to close the session and call the WFDCloseHandle function to signal to the Wi-Fi Direct service that the application is done using the service. This allows the Wi-Fi Direct service to release resources used by the application.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseHandle

WFDCloseSession

WFDOpenHandle

WFDStartOpenSession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDStartOpenSession function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDStartOpenSession** function starts an on-demand connection to a specific Wi-Fi Direct device, which has been previously paired through the Windows Pairing experience.

Syntax

```
DWORD WFDStartOpenSession(
  HANDLE                            hClientHandle,
  PDOT11_MAC_ADDRESS                pDeviceAddress,
  PVOID                             pvContext,
  WFD_OPEN_SESSION_COMPLETE_CALLBACK pfnCallback,
  PHANDLE                           phSessionHandle
);
```

Parameters

`hClientHandle`

A client handle to the Wi-Fi Direct service. This handle was obtained by a previous call to the WFDOpenHandle function.

`pDeviceAddress`

A pointer to the target device's Wi-Fi Direct device address. This is the MAC address of the target Wi-Fi device.

`pvContext`

An optional context pointer which is passed to the callback function specified in the *pfnCallback* parameter.

`pfnCallback`

A pointer to the callback function to be called once the **WFDStartOpenSession** request has completed.

`phSessionHandle`

A handle to this specific Wi-Fi Direct session.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.

ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *hClientHandle* parameter is **NULL** or not valid. This error is also returned if the *pDeviceAddress* parameter is **NULL**, the *pfnCallback* parameter is **NULL**, or the *phSessionHandle* parameter is **NULL**. This value is also returned if the *dwClientVersion* parameter is not equal to **WFD_API_VERSION**.
ERROR_INVALID_STATE	The group or resource is not in the correct state to perform the requested operation. This error is returned if the Wi-Fi Direct service is disabled by group policy on a domain.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
RPC_STATUS	Various error codes.

Remarks

The **WFDStartOpenSession** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

The **WFDStartOpenSession** function starts an asynchronous operation to start an on-demand connection to a specific Wi-Fi Direct device. The target Wi-Fi device must previously have been paired through the Windows Pairing experience. When the asynchronous operation completes, the callback function specified in the *pfnCallback* parameter is called.

If the application attempts to close the handle to the Wi-Fi Direct service by calling the WFDCloseHandle function before the **WFDStartOpenSession** function completes asynchronously, the **WFDCloseHandle** function will wait until the **WFDStartOpenSession** call is completed.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib

DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseHandle

WFDCloseSession

WFDOpenHandle

WFDOpenLegacySession

WFDUpdateDeviceVisibility

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WFDUpdateDeviceVisibility function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WFDUpdateDeviceVisibility** function updates device visibility for the Wi-Fi Direct device address for a given installed Wi-Fi Direct device node.

Syntax

```
DWORD WFDUpdateDeviceVisibility(
  PDOT11_MAC_ADDRESS pDeviceAddress
);
```

Parameters

`pDeviceAddress`

A pointer to the Wi-Fi Direct device address of the client device.

This device address must be obtained from a Device Node created as a result of the Inbox pairing experience.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	The parameter is incorrect. This error is returned if the *pDeviceAddress* parameter is **NULL**.
ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
RPC_STATUS	Various error codes.

Remarks

The **WFDUpdateDeviceVisibility** function is part of Wi-Fi Direct, a new feature in Windows 8 and Windows Server 2012. Wi-Fi Direct is based on the development of the Wi-Fi Peer-to-Peer Technical Specification v1.1 by the Wi-Fi Alliance (see Wi-Fi Alliance Published Specifications). The goal of the Wi-Fi Peer-to-Peer Technical Specification is to provide a solution for Wi-Fi device-to-device connectivity without the need for either a Wireless Access Point (wireless AP) to setup the connection or the use of the existing Wi-Fi adhoc (IBSS) mechanism.

The **WFDUpdateDeviceVisibility** function will perform a targeted Wi-Fi Direct discovery, and will update the DEVPKEY_WiFiDirect_IsVisibile property key on the device node for the given device.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WFDCancelOpenSession

WFDCloseHandle

WFDCloseSession

WFDOpenHandle

WFDOpenLegacySession

WFDStartOpenSession

WFD_OPEN_SESSION_COMPLETE_CALLBACK

WL_DISPLAY_PAGES enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Specifies the active tab when the wireless profile user interface dialog box appears.

Syntax

```
typedef enum _WL_DISPLAY_PAGES {
  WLConnectionPage,
  WLSecurityPage,
  WLAdvPage
} WL_DISPLAY_PAGES, *PWL_DISPLAY_PAGES;
```

Constants

`WLConnectionPage`	Displays the **Connection** tab.
`WLSecurityPage`	Displays the **Security** tab.
`WLAdvPage`	

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WlanUIEditProfile

WLAN_ASSOCIATION_ATTRIBUTES structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_ASSOCIATION_ATTRIBUTES** structure contains association attributes for a connection.

Syntax

```
typedef struct _WLAN_ASSOCIATION_ATTRIBUTES {
  DOT11_SSID          dot11Ssid;
  DOT11_BSS_TYPE      dot11BssType;
  DOT11_MAC_ADDRESS   dot11Bssid;
  DOT11_PHY_TYPE      dot11PhyType;
  ULONG               uDot11PhyIndex;
  WLAN_SIGNAL_QUALITY wlanSignalQuality;
  ULONG               ulRxRate;
  ULONG               ulTxRate;
} WLAN_ASSOCIATION_ATTRIBUTES, *PWLAN_ASSOCIATION_ATTRIBUTES;
```

Members

`dot11Ssid`

A DOT11_SSID structure that contains the SSID of the association.

`dot11BssType`

A DOT11_BSS_TYPE value that specifies whether the network is infrastructure or ad hoc.

`dot11Bssid`

A DOT11_MAC_ADDRESS that contains the BSSID of the association.

`dot11PhyType`

A DOT11_PHY_TYPE value that indicates the physical type of the association.

`uDot11PhyIndex`

The position of the DOT11_PHY_TYPE value in the structure containing the list of PHY types.

`wlanSignalQuality`

A percentage value that represents the signal quality of the network. **WLAN_SIGNAL_QUALITY** is of type **ULONG**. This member contains a value between 0 and 100. A value of 0 implies an actual RSSI signal strength of -100 dbm. A value of 100 implies an actual RSSI signal strength of -50 dbm. You can calculate the RSSI signal strength value for **wlanSignalQuality** values between 1 and 99 using linear interpolation.

`ulRxRate`

Contains the receiving rate of the association.

`ulTxRate`

Contains the transmission rate of the association.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_CONNECTION_ATTRIBUTES

WLAN_AUTH_CIPHER_PAIR_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_AUTH_CIPHER_PAIR_LIST** structure contains a list of authentication and cipher algorithm pairs.

Syntax

```
typedef struct _WLAN_AUTH_CIPHER_PAIR_LIST {
  DWORD                  dwNumberOfItems;
#if ...
  DOT11_AUTH_CIPHER_PAIR *pAuthCipherPairList[];
#else
  DOT11_AUTH_CIPHER_PAIR pAuthCipherPairList[1];
#endif
} WLAN_AUTH_CIPHER_PAIR_LIST, *PWLAN_AUTH_CIPHER_PAIR_LIST;
```

Members

`dwNumberOfItems`

Contains the number of supported auth-cipher pairs.

`pAuthCipherPairList`

A DOT11_AUTH_CIPHER_PAIR structure containing a list of auth-cipher pairs.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanQueryInterface

WLAN_AVAILABLE_NETWORK structure (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WLAN_AVAILABLE_NETWORK** structure contains information about an available wireless network.

Syntax

```
typedef struct _WLAN_AVAILABLE_NETWORK {
  WCHAR                  strProfileName[WLAN_MAX_NAME_LENGTH];
  DOT11_SSID             dot11Ssid;
  DOT11_BSS_TYPE         dot11BssType;
  ULONG                  uNumberOfBssids;
  BOOL                   bNetworkConnectable;
  WLAN_REASON_CODE       wlanNotConnectableReason;
  ULONG                  uNumberOfPhyTypes;
  DOT11_PHY_TYPE         dot11PhyTypes[WLAN_MAX_PHY_TYPE_NUMBER];
  BOOL                   bMorePhyTypes;
  WLAN_SIGNAL_QUALITY    wlanSignalQuality;
  BOOL                   bSecurityEnabled;
  DOT11_AUTH_ALGORITHM   dot11DefaultAuthAlgorithm;
  DOT11_CIPHER_ALGORITHM dot11DefaultCipherAlgorithm;
  DWORD                  dwFlags;
  DWORD                  dwReserved;
} WLAN_AVAILABLE_NETWORK, *PWLAN_AVAILABLE_NETWORK;
```

Members

`strProfileName`

Contains the profile name associated with the network. If the network does not have a profile, this member will be empty. If multiple profiles are associated with the network, there will be multiple entries with the same SSID in the visible network list. Profile names are case-sensitive. This string must be NULL-terminated.

`dot11Ssid`

A DOT11_SSID structure that contains the SSID of the visible wireless network.

`dot11BssType`

A DOT11_BSS_TYPE value that specifies whether the network is infrastructure or ad hoc.

`uNumberOfBssids`

Indicates the number of BSSIDs in the network.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: **uNumberofBssids** is at most 1, regardless of the number of access points broadcasting the SSID.

`bNetworkConnectable`

Indicates whether the network is connectable or not. If set to **TRUE**, the network is connectable, otherwise the network cannot be connected to.

`wlanNotConnectableReason`

A WLAN_REASON_CODE value that indicates why a network cannot be connected to. This member is only valid

when **bNetworkConnectable** is **FALSE**.

`uNumberOfPhyTypes`

The number of PHY types supported on available networks. The maximum value of *uNumberOfPhyTypes* is **WLAN_MAX_PHY_TYPE_NUMBER**, which has a value of 8. If more than **WLAN_MAX_PHY_TYPE_NUMBER** PHY types are supported, *bMorePhyTypes* must be set to **TRUE**.

`dot11PhyTypes`

Contains an array of DOT11_PHY_TYPE values that represent the PHY types supported by the available networks. When *uNumberOfPhyTypes* is greater than **WLAN_MAX_PHY_TYPE_NUMBER**, this array contains only the first **WLAN_MAX_PHY_TYPE_NUMBER** PHY types.

VALUE	MEANING
dot11_phy_type_unknown	Specifies an unknown or uninitialized PHY type.
dot11_phy_type_any	Specifies any PHY type.
dot11_phy_type_fhss	Specifies a frequency-hopping spread-spectrum (FHSS) PHY. Bluetooth devices can use FHSS or an adaptation of FHSS.
dot11_phy_type_dsss	Specifies a direct sequence spread spectrum (DSSS) PHY.
dot11_phy_type_irbaseband	Specifies an infrared (IR) baseband PHY.
dot11_phy_type_ofdm	Specifies an orthogonal frequency division multiplexing (OFDM) PHY. 802.11a devices can use OFDM.
dot11_phy_type_hrdsss	Specifies a high-rate DSSS (HRDSSS) PHY.
dot11_phy_type_erp	Specifies an extended rate PHY (ERP). 802.11g devices can use ERP.
dot11_phy_type_ht	Specifies an 802.11n PHY type.
dot11_phy_type_vht	Specifies the 802.11ac PHY type. This is the very high throughput PHY type specified in IEEE 802.11ac. This value is supported on Windows 8.1, Windows Server 2012 R2, and later.
dot11_phy_type_IHV_start	Specifies the start of the range that is used to define PHY types that are developed by an independent hardware vendor (IHV).

dot11_phy_type_IHV_end	Specifies the end of the range that is used to define PHY types that are developed by an independent hardware vendor (IHV).

`bMorePhyTypes`

Specifies if there are more than **WLAN_MAX_PHY_TYPE_NUMBER** PHY types supported.

When this member is set to **TRUE**, an application must call WlanGetNetworkBssList to get the complete list of PHY types. The returned WLAN_BSS_LIST structure has an array of WLAN_BSS_ENTRY structures. The *uPhyId* member of the **WLAN_BSS_ENTRY** structure contains the PHY type for an entry.

`wlanSignalQuality`

A percentage value that represents the signal quality of the network. **WLAN_SIGNAL_QUALITY** is of type **ULONG**. This member contains a value between 0 and 100. A value of 0 implies an actual RSSI signal strength of -100 dbm. A value of 100 implies an actual RSSI signal strength of -50 dbm. You can calculate the RSSI signal strength value for **wlanSignalQuality** values between 1 and 99 using linear interpolation.

`bSecurityEnabled`

Indicates whether security is enabled on the network. A value of **TRUE** indicates that security is enabled, otherwise it is not.

`dot11DefaultAuthAlgorithm`

A DOT11_AUTH_ALGORITHM value that indicates the default authentication algorithm used to join this network for the first time.

`dot11DefaultCipherAlgorithm`

A DOT11_CIPHER_ALGORITHM value that indicates the default cipher algorithm to be used when joining this network.

`dwFlags`

Contains various flags for the network.

VALUE	MEANING
WLAN_AVAILABLE_NETWORK_CONNECTED	This network is currently connected.
WLAN_AVAILABLE_NETWORK_HAS_PROFILE	There is a profile for this network.

`dwReserved`

Reserved for future use. Must be set to **NULL**.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_AVAILABLE_NETWORK_LIST

WLAN_AVAILABLE_NETWORK_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_AVAILABLE_NETWORK_LIST** structure contains an array of information about available networks.

Syntax

```
typedef struct _WLAN_AVAILABLE_NETWORK_LIST {
  DWORD                  dwNumberOfItems;
  DWORD                  dwIndex;
#if ...
  WLAN_AVAILABLE_NETWORK *Network[];
#else
  WLAN_AVAILABLE_NETWORK Network[1];
#endif
} WLAN_AVAILABLE_NETWORK_LIST, *PWLAN_AVAILABLE_NETWORK_LIST;
```

Members

`dwNumberOfItems`

Contains the number of items in the **Network** member.

`dwIndex`

The index of the current item. The index of the first item is 0. **dwIndex** must be less than **dwNumberOfItems**.

This member is not used by the wireless service. Applications can use this member when processing individual networks in the **WLAN_AVAILABLE_NETWORK_LIST** structure. When an application passes this structure from one function to another, it can set the value of **dwIndex** to the index of the item currently being processed. This can help an application maintain state.

dwIndex should always be initialized before use.

`Network`

An array of WLAN_AVAILABLE_NETWORK structures containing interface information.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanGetAvailableNetworkList

WLAN_BSS_ENTRY structure (wlanapi.h)

7/1/2021 • 6 minutes to read • Edit Online

The **WLAN_BSS_ENTRY** structure contains information about a basic service set (BSS).

Syntax

```
typedef struct _WLAN_BSS_ENTRY {
  DOT11_SSID        dot11Ssid;
  ULONG             uPhyId;
  DOT11_MAC_ADDRESS dot11Bssid;
  DOT11_BSS_TYPE    dot11BssType;
  DOT11_PHY_TYPE    dot11BssPhyType;
  LONG              lRssi;
  ULONG             uLinkQuality;
  BOOLEAN           bInRegDomain;
  USHORT            usBeaconPeriod;
  ULONGLONG         ullTimestamp;
  ULONGLONG         ullHostTimestamp;
  USHORT            usCapabilityInformation;
  ULONG             ulChCenterFrequency;
  WLAN_RATE_SET     wlanRateSet;
  ULONG             ulIeOffset;
  ULONG             ulIeSize;
} WLAN_BSS_ENTRY, *PWLAN_BSS_ENTRY;
```

Members

`dot11Ssid`

The SSID of the access point (AP) or peer station associated with the BSS. The data type for this member is a DOT11_SSID structure.

`uPhyId`

The identifier (ID) of the PHY that the wireless LAN interface used to detect the BSS network.

`dot11Bssid`

The media access control (MAC) address of the access point for infrastructure BSS networks or the peer station for independent BSS networks (ad hoc networks) that sent the 802.11 Beacon or Probe Response frame received by the wireless LAN interface while scanning. The data type for this member is a DOT11_MAC_ADDRESS structure.

`dot11BssType`

The BSS network type. The data type for this member is a DOT11_BSS_TYPE enumeration value.

This member can be one of the following values.

VALUE	MEANING
dot11_BSS_type_infrastructure 1	Specifies an infrastructure BSS network.

dot11_BSS_type_independent 2	Specifies an independent BSS (IBSS) network (an ad hoc network).

`dot11BssPhyType`

The PHY type for this network. The data type for this member is a DOT11_PHY_TYPE enumeration value.

`lRssi`

The received signal strength indicator (RSSI) value, in units of decibels referenced to 1.0 milliwatts (dBm), as detected by the wireless LAN interface driver for the AP or peer station.

`uLinkQuality`

The link quality reported by the wireless LAN interface driver. The link quality value ranges from 0 through 100. A value of 100 specifies the highest link quality.

`bInRegDomain`

A value that specifies whether the AP or peer station is operating within the regulatory domain as identified by the country/region.

If the wireless LAN interface driver does not support multiple regulatory domains, this member is set to **TRUE**.

If the 802.11 Beacon or Probe Response frame received from the AP or peer station does not include a Country information element (IE), this member is set to **TRUE**.

If the 802.11 Beacon or Probe Response frame received from the AP or peer station does include a Country IE, this member is set to **FALSE** if the value of the Country String subfield does not equal the input country string.

`usBeaconPeriod`

The value of the Beacon Interval field from the 802.11 Beacon or Probe Response frame received by the wireless LAN interface.

The interval is in 1,024 microsecond time units between target beacon transmission times. This information is retrieved from the beacon packet sent by an access point in an infrastructure BSS network or a probe response from an access point or peer station in response to a wireless LAN client sending a Probe Request.

The IEEE 802.11 standard defines a unit of time as equal to 1,024 microseconds. This unit was defined so that it could be easily implemented in hardware.

`ullTimestamp`

The value of the Timestamp field from the 802.11 Beacon or Probe Response frame received by the wireless LAN interface.

`ullHostTimestamp`

The host timestamp value that records when wireless LAN interface received the Beacon or Probe Response frame. This member is a count of 100-nanosecond intervals since January 1, 1601.

For more information, see the **NdisGetCurrentSystemTime** function documented in the WDK.

`usCapabilityInformation`

The value of the Capability Information field from the 802.11 Beacon or Probe Response frame received by the wireless LAN interface. This value is a set of bit flags defining the capability.

This member can be one or more of the following values.

VALUE	MEANING
ESS bit 0	An extended service set. A set of one or more interconnected basic service sets (BSSs) and integrated local area networks (LANs) that appears as a single BSS to the logical link control layer at any station associated with one of those BSSs. An AP set the ESS subfield to 1 and the IBSS subfield to 0 within transmitted Beacon or Probe Response frames. A peer station within an IBSS (ad hoc network) sets the ESS subfield to 0 and the IBSS subfield to 1 in transmitted Beacon or Probe Response frames.
IBSS bit 1	An independent basic service set. A BSS that forms a self-contained network, and in which no access to a distribution system (DS) is available (an ad hoc network). An AP sets the ESS subfield to 1 and the IBSS subfield to 0 within transmitted Beacon or Probe Response frames. A peer station within an IBSS (ad hoc network) sets the ESS subfield to 0 and the IBSS subfield to 1 in transmitted Beacon or Probe Response frames.
CF-Pollable bit 2	A value that indicates if the AP or peer station is pollable.
CF Poll Request bit 3	A value that indicates how the AP or peer station handles poll requests.
Privacy bit 4	A value that indicates if encryption is required for all data frames. An AP sets the Privacy subfield to 1 within transmitted Beacon and Probe Response frames if WEP, WPA, or WPA2 encryption is required for all data type frames exchanged within the BSS. If WEP, WPA, or WPA2 encryption is not required, the Privacy subfield is set to 0. A peer station within and IBSS sets the Privacy subfield to 1 within transmitted Beacon and Probe Response frames if WEP, WPA, or WPA2 encryption is required for all data type frames exchanged within the IBSS. If WEP, WPA, or WPA2 encryption is not required, the Privacy subfield is set to 0.

`ulChCenterFrequency`

The channel center frequency of the band on which the 802.11 Beacon or Probe Response frame was received. The value of **ulChCenterFrequency** is in units of kilohertz (kHz).

> **Note** This member is only valid for PHY types that are not frequency-hopping spread spectrum (FHSS).

`wlanRateSet`

A set of data transfer rates supported by the BSS. The data type for this member is a WLAN_RATE_SET structure.

`ulIeOffset`

The offset, in bytes, of the information element (IE) data blob from the beginning of the **WLAN_BSS_ENTRY** structure.

This member points to a buffer that contains variable-length information elements (IEs) from the 802.11 Beacon or Probe Response frames. For each BSS, the IEs are from the last Beacon or Probe Response frame received from that BSS network. If an IE is available in only one frame, the wireless LAN interface driver merges the IE with the other IEs from the last received Beacon or Probe Response frame.

Information elements are defined in the IEEE 802.11 specifications to have a common general format consisting of a 1-byte Element ID field, a 1-byte Length field, and a variable-length element-specific information field. Each information element is assigned a unique Element ID value as defined in this IEEE 802.11 standards. The Length field specifies the number of bytes in the information field.

`ulIeSize`

The size, in bytes, of the IE data blob in the **WLAN_BSS_ENTRY** structure.

This is the exact length of the data in the buffer pointed to by **ulIeOffset** member and does not contain any padding for alignment. The maximum value for the size of the IE data blob is 2,324 bytes.

Remarks

The WlanGetNetworkBssList function retrieves the BSS list of the wireless network or networks on a given interface and returns this information in a WLAN_BSS_LIST structure that contains an array of .**WLAN_BSS_ENTRY** structures.

When the wireless LAN interface is also operating as a Wireless Hosted Network , the BSS list will contain an entry for the BSS created for the Wireless Hosted Network.

Since the information is returned by the access point for an infrastructure BSS network or by the network peer for an independent BSS network (ad hoc network), the information returned should not be trusted. The **ulIeOffset** and **ulIeSize** members in the **WLAN_BSS_ENTRY** structure should be used to determine the maximum size of the information element data blob in the **WLAN_BSS_ENTRY** structure, not the data in the information element data blob.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_AVAILABLE_NETWORK

WLAN_AVAILABLE_NETWORK_LIST

WLAN_BSS_LIST

WlanGetAvailableNetworkList

WlanGetNetworkBssList

WLAN_BSS_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_BSS_LIST** structure contains a list of basic service set (BSS) entries.

Syntax

```
typedef struct _WLAN_BSS_LIST {
  DWORD          dwTotalSize;
  DWORD          dwNumberOfItems;
  WLAN_BSS_ENTRY wlanBssEntries[1];
} WLAN_BSS_LIST, *PWLAN_BSS_LIST;
```

Members

`dwTotalSize`

The total size of this structure, in bytes.

`dwNumberOfItems`

The number of items in the **wlanBssEntries** member.

`wlanBssEntries`

An array of WLAN_BSS_ENTRY structures that contains information about a BSS.

Remarks

The WlanGetNetworkBssList function retrieves the BSS list of the wireless network or networks on a given interface and returns this information in a **WLAN_BSS_LIST** structure.

The **WLAN_BSS_LIST** structure may contain padding for alignment between the **dwTotalSize** member, the **dwNumberOfItems** member, and the first WLAN_BSS_ENTRY array entry in the **wlanBssEntries** member. Padding for alignment may also be present between the **WLAN_BSS_ENTRY** array entries in the **wlanBssEntries** member. Any access to a **WLAN_BSS_ENTRY** array entry should assume padding may exist.

When the wireless LAN interface is also operating as a Wireless Hosted Network , the BSS list will contain an entry for the BSS created for the Wireless Hosted Network.

Since the information is returned by the access point for an infrastructure BSS network or by the network peer for an independent BSS network (ad hoc network), the information returned should not be trusted. The **ulIeOffset** and **ulIeSize** members in the WLAN_BSS_ENTRY structure should be used to determine the maximum size of the information element data blob in the **WLAN_BSS_ENTRY** structure, not the data in the information element data blob.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_AVAILABLE_NETWORK

WLAN_AVAILABLE_NETWORK_LIST

WLAN_BSS_ENTRY

WlanGetAvailableNetworkList

WlanGetNetworkBssList

WLAN_CONNECTION_ATTRIBUTES structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_CONNECTION_ATTRIBUTES** structure defines the attributes of a wireless connection.

Syntax

```
typedef struct _WLAN_CONNECTION_ATTRIBUTES {
  WLAN_INTERFACE_STATE        isState;
  WLAN_CONNECTION_MODE        wlanConnectionMode;
  WCHAR                       strProfileName[WLAN_MAX_NAME_LENGTH];
  WLAN_ASSOCIATION_ATTRIBUTES wlanAssociationAttributes;
  WLAN_SECURITY_ATTRIBUTES    wlanSecurityAttributes;
} WLAN_CONNECTION_ATTRIBUTES, *PWLAN_CONNECTION_ATTRIBUTES;
```

Members

`isState`

A WLAN_INTERFACE_STATE value that indicates the state of the interface.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_interface_state_connected**, **wlan_interface_state_disconnected**, and **wlan_interface_state_authenticating** values are supported.

`wlanConnectionMode`

A WLAN_CONNECTION_MODE value that indicates the mode of the connection.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_connection_mode_profile** value is supported.

`strProfileName`

The name of the profile used for the connection. Profile names are case-sensitive. This string must be NULL-terminated.

`wlanAssociationAttributes`

A WLAN_ASSOCIATION_ATTRIBUTES structure that contains the attributes of the association.

`wlanSecurityAttributes`

A WLAN_SECURITY_ATTRIBUTES structure that contains the security attributes of the connection.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanQueryInterface

WLAN_CONNECTION_MODE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_CONNECTION_MODE** enumerated type defines the mode of connection.**Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** Only the **wlan_connection_mode_profile** value is supported.

Syntax

```
typedef enum _WLAN_CONNECTION_MODE {
  wlan_connection_mode_profile,
  wlan_connection_mode_temporary_profile,
  wlan_connection_mode_discovery_secure,
  wlan_connection_mode_discovery_unsecure,
  wlan_connection_mode_auto,
  wlan_connection_mode_invalid
} WLAN_CONNECTION_MODE, *PWLAN_CONNECTION_MODE;
```

Constants

`wlan_connection_mode_profile` A profile will be used to make the connection.
`wlan_connection_mode_temporary_profile` A temporary profile will be used to make the connection.
`wlan_connection_mode_discovery_secure` Secure discovery will be used to make the connection.
`wlan_connection_mode_discovery_unsecure` Unsecure discovery will be used to make the connection.
`wlan_connection_mode_auto` The connection is initiated by the wireless service automatically using a persistent profile.
`wlan_connection_mode_invalid` Not used.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_CONNECTION_ATTRIBUTES

WLAN_CONNECTION_NOTIFICATION_DATA

WLAN_CONNECTION_PARAMETERS

WLAN_CONNECTION_NOTIFICATION_DATA structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_CONNECTION_NOTIFICATION_DATA** structure contains information about connection related notifications.

Syntax

```
typedef struct _WLAN_CONNECTION_NOTIFICATION_DATA {
  WLAN_CONNECTION_MODE wlanConnectionMode;
  WCHAR                strProfileName[WLAN_MAX_NAME_LENGTH];
  DOT11_SSID           dot11Ssid;
  DOT11_BSS_TYPE       dot11BssType;
  BOOL                 bSecurityEnabled;
  WLAN_REASON_CODE     wlanReasonCode;
  DWORD                dwFlags;
  WCHAR                strProfileXml[1];
} WLAN_CONNECTION_NOTIFICATION_DATA, *PWLAN_CONNECTION_NOTIFICATION_DATA;
```

Members

`wlanConnectionMode`

A WLAN_CONNECTION_MODE value that specifies the mode of the connection.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_connection_mode_profile** value is supported.

`strProfileName`

The name of the profile used for the connection. WLAN_MAX_NAME_LENGTH is 256. Profile names are case-sensitive. This string must be NULL-terminated.

`dot11Ssid`

A DOT11_SSID structure that contains the SSID of the association.

`dot11BssType`

A DOT11_BSS_TYPE value that indicates the BSS network type.

`bSecurityEnabled`

Indicates whether security is enabled for this connection. If **TRUE**, security is enabled.

`wlanReasonCode`

A WLAN_REASON_CODE that indicates the reason for an operation failure. This field has a value of **WLAN_REASON_CODE_SUCCESS** for all connection-related notifications except **wlan_notification_acm_connection_complete**. If the connection fails, this field indicates the reason for the failure.

`dwFlags`

A set of flags that provide additional information for the network connection.

This member can be one of the following values defined in the *Wlanapi.h* header file.

VALUE	MEANING
WLAN_CONNECTION_NOTIFICATION_ADHOC_NETWORK_FORMED	Indicates that an adhoc network is formed.
WLAN_CONNECTION_NOTIFICATION_CONSOLE_USER_PROFILE	Indicates that the connection uses a per-user profile owned by the console user. Non-console users will not be able to see the profile in their profile list.

`strProfileXml`

This field contains the XML presentation of the profile used for discovery, if the connection succeeds.

Remarks

The WlanRegisterNotification function is used by an application to register and unregister notifications on all wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter passed to the **WlanRegisterNotification** function. The prototype for this callback function is the WLAN_NOTIFICATION_CALLBACK. This callback function will receive notifications that have been registered in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function.

The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter that contains detailed information on the notification.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_ACM**, then the received notification is an auto configuration module notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the WLAN_NOTIFICATION_CALLBACK function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For some of these notifications, a **WLAN_CONNECTION_NOTIFICATION_DATA** structure is returned in the *pData* member of **WLAN_NOTIFICATION_DATA** structure.

For more information on these notifications, see the WLAN_NOTIFICATION_ACM enumeration reference.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_NOTIFICATION_ACM

WLAN_NOTIFICATION_CALLBACK

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

WLAN_CONNECTION_PARAMETERS structure (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WLAN_CONNECTION_PARAMETERS** structure specifies the parameters used when using the WlanConnect function.

Syntax

```
typedef struct _WLAN_CONNECTION_PARAMETERS {
  WLAN_CONNECTION_MODE wlanConnectionMode;
#if ...
  LPCWSTR              strProfile;
#else
  LPCWSTR              strProfile;
#endif
  PDOT11_SSID          pDot11Ssid;
  PDOT11_BSSID_LIST    pDesiredBssidList;
  DOT11_BSS_TYPE       dot11BssType;
  DWORD                dwFlags;
} WLAN_CONNECTION_PARAMETERS, *PWLAN_CONNECTION_PARAMETERS;
```

Members

`wlanConnectionMode`

A WLAN_CONNECTION_MODE value that specifies the mode of connection.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_connection_mode_profile** value is supported.

`strProfile`

Specifies the profile being used for the connection.

If **wlanConnectionMode** is set to **wlan_connection_mode_profile**, then **strProfile** specifies the name of the profile used for the connection. If **wlanConnectionMode** is set to **wlan_connection_mode_temporary_profile**, then **strProfile** specifies the XML representation of the profile used for the connection. If **wlanConnectionMode** is set to **wlan_connection_mode_discovery_secure** or **wlan_connection_mode_discovery_unsecure**, then **strProfile** should be set to **NULL**.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The profile must meet the compatibility criteria described in Wireless Profile Compatibility.

`pDot11Ssid`

Pointer to a DOT11_SSID structure that specifies the SSID of the network to connect to. This parameter is optional. When set to **NULL**, all SSIDs in the profile will be tried. This parameter must not be **NULL** if WLAN_CONNECTION_MODE is set to **wlan_connection_mode_discovery_secure** or **wlan_connection_mode_discovery_unsecure**.

`pDesiredBssidList`

Pointer to a DOT11_BSSID_LIST structure that contains the list of basic service set (BSS) identifiers desired for the connection.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This member must be **NULL**.

`dot11BssType`

A DOT11_BSS_TYPE value that indicates the BSS type of the network. If a profile is provided, this BSS type must be the same as the one in the profile.

`dwFlags`

The following table shows flags used to specify the connection parameters.

CONSTANT	VALUE	DESCRIPTION
WLAN_CONNECTION_HIDDEN_NETWORK	0x00000001	Connect to the destination network even if the destination is a hidden network. A hidden network does not broadcast its SSID. Do not use this flag if the destination network is an ad-hoc network.If the profile specified by **strProfile** is not **NULL**, then this flag is ignored and the nonBroadcast profile element determines whether to connect to a hidden network.
WLAN_CONNECTION_ADHOC_JOIN_ONLY	0x00000002	Do not form an ad-hoc network. Only join an ad-hoc network if the network already exists. Do not use this flag if the destination network is an infrastructure network.
WLAN_CONNECTION_IGNORE_PRIVACY_BIT	0x00000004	Ignore the privacy bit when connecting to the network. Ignoring the privacy bit has the effect of ignoring whether packets are encrypted and ignoring the method of encryption used. Only use this flag when connecting to an infrastructure network using a temporary profile.
WLAN_CONNECTION_EAPOL_PASSTHROUGH	0x00000008	Exempt EAPOL traffic from encryption and decryption. This flag is used when an application must send EAPOL traffic over an infrastructure network that uses Open authentication and WEP encryption. This flag must not be used to connect to networks that require 802.1X authentication. This flag is only valid when **wlanConnectionMode** is set to **wlan_connection_mode_temporary_profile**. Avoid using this flag whenever possible.

WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE	0x00000010	Automatically persist discovery profile on successful connection completion. This flag is only valid for wlan_connection_mode_discovery_secure or wlan_connection_mode_discovery_unsecure. The profile will be saved as an all user profile, with the name generated from the SSID using WlanUtf8SsidToDisplayName. If there is already a profile with the same name, a number will be appended to the end of the profile name. The profile will be saved with manual connection mode, unless WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE_CONNECTION_MODE_AUTO is also specified.
WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE_CONNECTION_MODE_AUTO	0x00000020	To be used in conjunction with WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE. The discovery profile will be persisted with automatic connection mode.
WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE_OVERWRITE_EXISTING	0x00000040	To be used in conjunction with WLAN_CONNECTION_PERSIST_DISCOVERY_PROFILE. The discovery profile will be persisted and attempt to overwrite an existing profile with the same name.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This member must be set to 0.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanConnect

WLAN_COUNTRY_OR_REGION_STRING_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

A **WLAN_COUNTRY_OR_REGION_STRING_LIST** structure contains a list of supported country or region strings.

Syntax

```
typedef struct _WLAN_COUNTRY_OR_REGION_STRING_LIST {
  DWORD                          dwNumberOfItems;
#if ...
  DOT11_COUNTRY_OR_REGION_STRING *pCountryOrRegionStringList[];
#else
  DOT11_COUNTRY_OR_REGION_STRING pCountryOrRegionStringList[1];
#endif
} WLAN_COUNTRY_OR_REGION_STRING_LIST, *PWLAN_COUNTRY_OR_REGION_STRING_LIST;
```

Members

`dwNumberOfItems`

Indicates the number of supported country or region strings.

`pCountryOrRegionStringList`

A list of supported country or region strings. In Windows, a **DOT11_COUNTRY_OR_REGION_STRING** is of type **char[3]**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WlanQueryInterface

WLAN_DEVICE_SERVICE_GUID_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Contains an array of device service GUIDs.

Syntax

```
typedef struct _WLAN_DEVICE_SERVICE_GUID_LIST {
  DWORD dwNumberOfItems;
  DWORD dwIndex;
#if ...
  GUID  *DeviceService[];
#else
  GUID  DeviceService[1];
#endif
} WLAN_DEVICE_SERVICE_GUID_LIST, *PWLAN_DEVICE_SERVICE_GUID_LIST;
```

Members

`dwNumberOfItems`

Type: DWORD

The number of items in the *DeviceService* argument.

`dwIndex`

Type: DWORD

The index of the current item. The index of the first item is 0. *dwIndex* must be less than *dwNumberOfItems*. This member is not used by the wireless service. You can use this member when processing individual **GUID**s in the **WLAN_DEVICE_SERVICE_GUID_LIST** structure. When your application passes this structure from one function to another, it can set the value of *dwIndex* to the index of the item currently being processed. This can help your application maintain state. You should always initialize *dwIndex* before use.

`DeviceService`

Type: GUID[1]

A pointer to an array containing **GUID**s; each corresponds to a WLAN device service that the driver supports.

Requirements

Header	wlanapi.h

WLAN_DEVICE_SERVICE_NOTIFICATION_DATA structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

A structure that represents a device service notification.

Syntax

```
typedef struct _WLAN_DEVICE_SERVICE_NOTIFICATION_DATA {
  GUID  DeviceService;
  DWORD dwOpCode;
  DWORD dwDataSize;
#if ...
  BYTE  *DataBlob[];
#else
  BYTE  DataBlob[1];
#endif
} WLAN_DEVICE_SERVICE_NOTIFICATION_DATA, *PWLAN_DEVICE_SERVICE_NOTIFICATION_DATA;
```

Members

`DeviceService`

Type: GUID

The **GUID** identifying the device service for this notification.

`dwOpCode`

Type: DWORD

The opcode that identifies the operation under the device service for this notification.

`dwDataSize`

Type: DWORD

The size, in bytes, of the *DataBlob* member. The maximum value of *dwDataSize* may be restricted by the type of data that is stored in the **WLAN_DEVICE_SERVICE_NOTIFICATION_DATA** structure.

`DataBlob`

Type: BYTE[1]

A pointer to an array containing **BYTE**s, representing the data blob. This is the data that is received from the independent hardware vendor (IHV) driver, and is passed on to the client as an unformatted byte array blob.

Requirements

Header	wlanapi.h

WLAN_FILTER_LIST_TYPE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_FILTER_LIST_TYPE** enumerated type indicates types of filter lists.

Syntax

```
typedef enum _WLAN_FILTER_LIST_TYPE {
  wlan_filter_list_type_gp_permit,
  wlan_filter_list_type_gp_deny,
  wlan_filter_list_type_user_permit,
  wlan_filter_list_type_user_deny
} WLAN_FILTER_LIST_TYPE, *PWLAN_FILTER_LIST_TYPE;
```

Constants

`wlan_filter_list_type_gp_permit`	Group policy permit list.
`wlan_filter_list_type_gp_deny`	Group policy deny list.
`wlan_filter_list_type_user_permit`	User permit list.
`wlan_filter_list_type_user_deny`	User deny list.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WlanGetFilterList

WlanSetFilterList

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS** structure contains information about the connection settings on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS {
  DOT11_SSID hostedNetworkSSID;
  DWORD      dwMaxNumberOfPeers;
} WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS, *PWLAN_HOSTED_NETWORK_CONNECTION_SETTINGS;
```

Members

`hostedNetworkSSID`

The SSID associated with the wireless Hosted Network.

`dwMaxNumberOfPeers`

The maximum number of concurrent peers allowed by the wireless Hosted Network.

Remarks

The **WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_SSID

WlanHostedNetworkQueryProperty

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE** structure contains information about a network state change for a data peer on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE {
  WLAN_HOSTED_NETWORK_PEER_STATE OldState;
  WLAN_HOSTED_NETWORK_PEER_STATE NewState;
  WLAN_HOSTED_NETWORK_REASON     PeerStateChangeReason;
} WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE, *PWLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE;
```

Members

`OldState`

The previous network state for a data peer on the wireless Hosted Network.

`NewState`

The current network state for a data peer on the wireless Hosted Network.

`PeerStateChangeReason`

The reason for the network state change for the data peer.

Remarks

The **WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_HOSTED_NETWORK_PEER_STATE

WLAN_HOSTED_NETWORK_REASON

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

WLAN_HOSTED_NETWORK_NOTIFICATION_CODE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_NOTIFICATION_CODE** enumerated type specifies the possible values of the NotificationCode parameter for received notifications on the wireless Hosted Network.

Syntax

```
typedef enum _WLAN_HOSTED_NETWORK_NOTIFICATION_CODE {
  wlan_hosted_network_state_change,
  wlan_hosted_network_peer_state_change,
  wlan_hosted_network_radio_state_change
} WLAN_HOSTED_NETWORK_NOTIFICATION_CODE, *PWLAN_HOSTED_NETWORK_NOTIFICATION_CODE;
```

Constants

`wlan_hosted_network_state_change`	The Hosted Network state has changed.
`wlan_hosted_network_peer_state_change`	The Hosted Network peer state has changed.
`wlan_hosted_network_radio_state_change`	The Hosted Network radio state has changed.

Remarks

The **WLAN_HOSTED_NETWORK_NOTIFICATION_CODE** enumerated type is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

The **WLAN_HOSTED_NETWORK_NOTIFICATION_CODE** specifies the possible values for the NotificationCode parameter for received notifications when the NotificationSource parameter is WLAN_NOTIFICATION_SOURCE_HNWK on the wireless Hosted Network.

The starting value for the **WLAN_HOSTED_NETWORK_NOTIFICATION_CODE** enumeration is defined as L2_NOTIFICATION_CODE_V2_BEGIN, which is defined in the *l2cmn.h* header file. Note that the *l2cmn.h* header is automatically included by the *wlanapi.h* header file.

The WlanRegisterNotification function is used by an application to register and unregister notifications on all wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter passed to the **WlanRegisterNotification** function. The prototype for this callback function is the WLAN_NOTIFICATION_CALLBACK. This callback function will receive notifications that have been registered in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function.

The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter

that contains detailed information on the notification. The callback function also receives a second parameter that contains a pointer to the client context passed in the *pCallbackContext* parameter to the WlanRegisterNotification function. This client context can be a **NULL** pointer if that is what was passed to the **WlanRegisterNotification** function.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_HNWK**, then the received notification is a wireless Hosted Network notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the WLAN_NOTIFICATION_CALLBACK function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure.

NOTIFICATIONCODE	DESCRIPTION
wlan_hosted_network_state_change	The *pData* member of WLAN_NOTIFICATION_DATA structure should be cast to a pointer to a WLAN_HOSTED_NETWORK_STATE_CHANGE structure and **dwDataSize** member would be at least as large as sizeof(**WLAN_HOSTED_NETWORK_STATE_CHANGE**).
wlan_hosted_network_peer_state_change	the *pData* member of WLAN_NOTIFICATION_DATA structure should be cast to a pointer to a WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE structure and **dwDataSize** member would be at least as large as sizeof(**WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE**).
wlan_hosted_network_radio_state_change	the *pData* member of WLAN_NOTIFICATION_DATA structure should be cast to a pointer to a WLAN_HOSTED_NETWORK_RADIO_STATE structure and **dwDataSize** member would be at least as large as sizeof(**WLAN_HOSTED_NETWORK_RADIO_STATE**).

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE

WLAN_HOSTED_NETWORK_RADIO_STATE

WLAN_HOSTED_NETWORK_STATE_CHANGE

WLAN_NOTIFICATION_CALLBACK

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

WLAN_HOSTED_NETWORK_OPCODE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_OPCODE** enumerated type specifies the possible values of the operation code for the properties to query or set on the wireless Hosted Network.

Syntax

```
typedef enum _WLAN_HOSTED_NETWORK_OPCODE {
  wlan_hosted_network_opcode_connection_settings,
  wlan_hosted_network_opcode_security_settings,
  wlan_hosted_network_opcode_station_profile,
  wlan_hosted_network_opcode_enable
} WLAN_HOSTED_NETWORK_OPCODE, *PWLAN_HOSTED_NETWORK_OPCODE;
```

Constants

`wlan_hosted_network_opcode_connection_settings`	The opcode used to query or set the wireless Hosted Network connection settings.
`wlan_hosted_network_opcode_security_settings`	The opcode used to query the wireless Hosted Network security settings.
`wlan_hosted_network_opcode_station_profile`	The opcode used to query the wireless Hosted Network station profile.
`wlan_hosted_network_opcode_enable`	The opcode used to query or set the wireless Hosted Network enabled flag.

Remarks

The **WLAN_HOSTED_NETWORK_OPCODE** enumerated type is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

The **WLAN_HOSTED_NETWORK_OPCODE** specifies the possible values of the operation code for the properties to query or set on the wireless Hosted Network.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]

Header	wlanapi.h (include Wlanapi.h)

See also

WlanHostedNetworkQueryProperty

WlanHostedNetworkSetProperty

WLAN_HOSTED_NETWORK_PEER_AUTH_STATE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_PEER_AUTH_STATE** enumerated type specifies the possible values for the authentication state of a peer on the wireless Hosted Network.

Syntax

```
typedef enum _WLAN_HOSTED_NETWORK_PEER_AUTH_STATE {
  wlan_hosted_network_peer_state_invalid,
  wlan_hosted_network_peer_state_authenticated
} WLAN_HOSTED_NETWORK_PEER_AUTH_STATE, *PWLAN_HOSTED_NETWORK_PEER_AUTH_STATE;
```

Constants

`wlan_hosted_network_peer_state_invalid` An invalid peer state.
`wlan_hosted_network_peer_state_authenticated` The peer is authenticated.

Remarks

The **WLAN_HOSTED_NETWORK_PEER_AUTH_STATE** enumerated type is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE

WLAN_HOSTED_NETWORK_PEER_STATE

WLAN_HOSTED_NETWORK_STATUS

WlanHostedNetworkQueryStatus

WLAN_HOSTED_NETWORK_PEER_STATE structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_PEER_STATE** structure contains information about the peer state for a peer on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_PEER_STATE {
  DOT11_MAC_ADDRESS                 PeerMacAddress;
  WLAN_HOSTED_NETWORK_PEER_AUTH_STATE PeerAuthState;
} WLAN_HOSTED_NETWORK_PEER_STATE, *PWLAN_HOSTED_NETWORK_PEER_STATE;
```

Members

`PeerMacAddress`

The MAC address of the peer being described.

`PeerAuthState`

The current authentication state of this peer.

Remarks

The **WLAN_HOSTED_NETWORK_PEER_STATE** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_MAC_ADDRESS

WLAN_HOSTED_NETWORK_DATA_PEER_STATE_CHANGE

WLAN_HOSTED_NETWORK_PEER_AUTH_STATE

WLAN_HOSTED_NETWORK_STATUS

WlanHostedNetworkQueryStatus

WLAN_HOSTED_NETWORK_RADIO_STATE structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_RADIO_STATE** structure contains information about the radio state on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_RADIO_STATE {
  DOT11_RADIO_STATE dot11SoftwareRadioState;
  DOT11_RADIO_STATE dot11HardwareRadioState;
} WLAN_HOSTED_NETWORK_RADIO_STATE, *PWLAN_HOSTED_NETWORK_RADIO_STATE;
```

Members

`dot11SoftwareRadioState`

The software radio state of the wireless Hosted Network.

`dot11HardwareRadioState`

The hardware radio state of the wireless Hosted Network.

Remarks

The **WLAN_HOSTED_NETWORK_RADIO_STATE** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_RADIO_STATE

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

WLAN_HOSTED_NETWORK_REASON enumeration (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_REASON** enumerated type specifies the possible values for the result of a wireless Hosted Network function call.

Syntax

```
typedef enum _WLAN_HOSTED_NETWORK_REASON {
  wlan_hosted_network_reason_success,
  wlan_hosted_network_reason_unspecified,
  wlan_hosted_network_reason_bad_parameters,
  wlan_hosted_network_reason_service_shutting_down,
  wlan_hosted_network_reason_insufficient_resources,
  wlan_hosted_network_reason_elevation_required,
  wlan_hosted_network_reason_read_only,
  wlan_hosted_network_reason_persistence_failed,
  wlan_hosted_network_reason_crypt_error,
  wlan_hosted_network_reason_impersonation,
  wlan_hosted_network_reason_stop_before_start,
  wlan_hosted_network_reason_interface_available,
  wlan_hosted_network_reason_interface_unavailable,
  wlan_hosted_network_reason_miniport_stopped,
  wlan_hosted_network_reason_miniport_started,
  wlan_hosted_network_reason_incompatible_connection_started,
  wlan_hosted_network_reason_incompatible_connection_stopped,
  wlan_hosted_network_reason_user_action,
  wlan_hosted_network_reason_client_abort,
  wlan_hosted_network_reason_ap_start_failed,
  wlan_hosted_network_reason_peer_arrived,
  wlan_hosted_network_reason_peer_departed,
  wlan_hosted_network_reason_peer_timeout,
  wlan_hosted_network_reason_gp_denied,
  wlan_hosted_network_reason_service_unavailable,
  wlan_hosted_network_reason_device_change,
  wlan_hosted_network_reason_properties_change,
  wlan_hosted_network_reason_virtual_station_blocking_use,
  wlan_hosted_network_reason_service_available_on_virtual_station
} WLAN_HOSTED_NETWORK_REASON, *PWLAN_HOSTED_NETWORK_REASON;
```

Constants

`wlan_hosted_network_reason_success` The operation was successful.
`wlan_hosted_network_reason_unspecified` Unknown error.

`wlan_hosted_network_reason_bad_parameters`
Bad parameters.

For example, this reason code is returned if an application failed to reference the client context from the correct handle (the handle returned by the WlanOpenHandle function).

`wlan_hosted_network_reason_service_shutting_down`
Service is shutting down.

`wlan_hosted_network_reason_insufficient_resources`
Service is out of resources.

`wlan_hosted_network_reason_elevation_required`
This operation requires elevation.

`wlan_hosted_network_reason_read_only`
An attempt was made to write read-only data.

`wlan_hosted_network_reason_persistence_failed`
Data persistence failed.

`wlan_hosted_network_reason_crypt_error`
A cryptographic error occurred.

`wlan_hosted_network_reason_impersonation`
User impersonation failed.

`wlan_hosted_network_reason_stop_before_start`
An incorrect function call sequence was made.

`wlan_hosted_network_reason_interface_available`
A wireless interface has become available.

`wlan_hosted_network_reason_interface_unavailable`
A wireless interface has become unavailable.

This reason code is returned by the wireless Hosted Network functions any time the network state of the wireless Hosted Network is **wlan_hosted_network_unavailable**. For example if the wireless Hosted Network is disabled by group policy on a domain, then the network state of the wireless Hosted Network is **wlan_hosted_network_unavailable**. In this case, any calls to the WlanHostedNetworkStartUsing or WlanHostedNetworkForceStart function would return this reason code.

`wlan_hosted_network_reason_miniport_stopped`
The wireless miniport driver stopped the Hosted Network.

`wlan_hosted_network_reason_miniport_started`
The wireless miniport driver status changed.

`wlan_hosted_network_reason_incompatible_connection_started`
An incompatible connection started.

An incompatible connection refers to one of the following cases:

- An ad hoc wireless connection is started on the primary station adapter.
- Network monitoring is started on the primary station adapter by an application (Network Monitor, for example) that calls the WlanSetInterface function with the *OpCode* parameter set to **wlan_intf_opcode_current_operation_mode** and the *pData* parameter points to a ULONG that contains **DOT11_OPERATION_MODE_NETWORK_MONITOR**.
- A wireless connection is started in FIPS safe mode on the primary station adapter. FIPS safe mode is specified in the profile of the wireless connection. For more information, see the FIPSMode Element .

Windows will stop the wireless Hosted Network on the software-based wireless access point (AP) adapter when an incompatible connection starts on the primary station adapter. The network state of the wireless Hosted Network state would become **wlan_hosted_network_unavailable**.

`wlan_hosted_network_reason_incompatible_connection_stopped`
An incompatible connection stopped.

An incompatible connection previously started on the primary station adapter (wlan_hosted_network_reason_incompatible_connection_started), but the incompatible connection has stopped. If the wireless Hosted Network was previously stopped as a result of an incompatible connection being started, Windows will not automatically restart the wireless Hosted Network. Applications can restart the wireless Hosted Network on the AP adapter by calling the WlanHostedNetworkStartUsing or WlanHostedNetworkForceStart function.

`wlan_hosted_network_reason_user_action`
A state change occurred that was caused by explicit user action.

`wlan_hosted_network_reason_client_abort`
A state change occurred that was caused by client abort.

`wlan_hosted_network_reason_ap_start_failed`
The driver for the wireless Hosted Network failed to start.

`wlan_hosted_network_reason_peer_arrived`
A peer connected to the wireless Hosted Network.

`wlan_hosted_network_reason_peer_departed`
A peer disconnected from the wireless Hosted Network.

`wlan_hosted_network_reason_peer_timeout`
A peer timed out.

`wlan_hosted_network_reason_gp_denied`
The operation was denied by group policy.

`wlan_hosted_network_reason_service_unavailable`
The Wireless LAN service is not running.

`wlan_hosted_network_reason_device_change`	The wireless adapter used by the wireless Hosted Network changed.
`wlan_hosted_network_reason_properties_change`	The properties of the wireless Hosted Network changed.
`wlan_hosted_network_reason_virtual_station_blocking_use`	A virtual station is active and blocking operation.
`wlan_hosted_network_reason_service_available_on_virtual_station`	An identical service is available on a virtual station.

Remarks

The **WLAN_HOSTED_NETWORK_REASON** enumerated type is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

The **WLAN_HOSTED_NETWORK_REASON** enumerates the possible reasons that a wireless Hosted Network function call failed or the reasons why a particular wireless Hosted Network notification was generated.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WlanEnumInterfaces

WlanHostedNetworkForceStart

WlanHostedNetworkForceStop

WlanHostedNetworkInitSettings

WlanHostedNetworkQuerySecondaryKey

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanHostedNetworkSetSecondaryKey

WlanHostedNetworkStartUsing

WlanHostedNetworkStopUsing

WLAN_HOSTED_NETWORK_SECURITY_SETTINGS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_SECURITY_SETTINGS** structure contains information about the security settings on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_SECURITY_SETTINGS {
  DOT11_AUTH_ALGORITHM   dot11AuthAlgo;
  DOT11_CIPHER_ALGORITHM dot11CipherAlgo;
} WLAN_HOSTED_NETWORK_SECURITY_SETTINGS, *PWLAN_HOSTED_NETWORK_SECURITY_SETTINGS;
```

Members

`dot11AuthAlgo`

The authentication algorithm used by the wireless Hosted Network.

`dot11CipherAlgo`

The cipher algorithm used by the wireless Hosted Network.

Remarks

The **WLAN_HOSTED_NETWORK_SECURITY_SETTINGS** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_AUTH_ALGORITHM

DOT11_CIPHER_ALGORITHM

WlanHostedNetworkQueryProperty

WLAN_HOSTED_NETWORK_STATE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_STATE** enumerated type specifies the possible values for the network state of the wireless Hosted Network.

Syntax

```
typedef enum _WLAN_HOSTED_NETWORK_STATE {
  wlan_hosted_network_unavailable,
  wlan_hosted_network_idle,
  wlan_hosted_network_active
} WLAN_HOSTED_NETWORK_STATE, *PWLAN_HOSTED_NETWORK_STATE;
```

Constants

`wlan_hosted_network_unavailable` The wireless Hosted Network is unavailable.	
`wlan_hosted_network_idle` The wireless Hosted Network is idle.	
`wlan_hosted_network_active` The wireless Hosted Network is active.	

Remarks

The **WLAN_HOSTED_NETWORK_STATE** enumerated type is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_HOSTED_NETWORK_STATE_CHANGE

WLAN_HOSTED_NETWORK_STATUS

WLAN_HOSTED_NETWORK_STATE_CHANGE structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_STATE_CHANGE** structure contains information about a network state change on the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_STATE_CHANGE {
  WLAN_HOSTED_NETWORK_STATE  OldState;
  WLAN_HOSTED_NETWORK_STATE  NewState;
  WLAN_HOSTED_NETWORK_REASON StateChangeReason;
} WLAN_HOSTED_NETWORK_STATE_CHANGE, *PWLAN_HOSTED_NETWORK_STATE_CHANGE;
```

Members

`OldState`

The previous network state on the wireless Hosted Network.

`NewState`

The current network state on the wireless Hosted Network.

`StateChangeReason`

The reason for the network state change.

Remarks

The **WLAN_HOSTED_NETWORK_STATE_CHANGE** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_HOSTED_NETWORK_REASON

WLAN_HOSTED_NETWORK_STATE

WLAN_NOTIFICATION_DATA

WlanRegisterNotification

WLAN_HOSTED_NETWORK_STATUS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_HOSTED_NETWORK_STATUS** structure contains information about the status of the wireless Hosted Network.

Syntax

```
typedef struct _WLAN_HOSTED_NETWORK_STATUS {
  WLAN_HOSTED_NETWORK_STATE      HostedNetworkState;
  GUID                           IPDeviceID;
  DOT11_MAC_ADDRESS              wlanHostedNetworkBSSID;
  DOT11_PHY_TYPE                 dot11PhyType;
  ULONG                          ulChannelFrequency;
  DWORD                          dwNumberOfPeers;
#if ...
  WLAN_HOSTED_NETWORK_PEER_STATE *PeerList[];
#else
  WLAN_HOSTED_NETWORK_PEER_STATE PeerList[1];
#endif
} WLAN_HOSTED_NETWORK_STATUS, *PWLAN_HOSTED_NETWORK_STATUS;
```

Members

`HostedNetworkState`

The current state of the wireless Hosted Network.

If the value of this member is **wlan_hosted_network_unavailable**, then the values of the other fields in this structure should not be used.

`IPDeviceID`

The actual network Device ID used for the wireless Hosted Network.

This is member is the GUID of a virtual wireless device which would not be available through calls to the WlanEnumInterfaces function. This GUID can be used for calling other higher layer networking functions that use the device GUID (IP Helper functions, for example).

`wlanHostedNetworkBSSID`

The BSSID used by the wireless Hosted Network in packets, beacons, and probe responses.

`dot11PhyType`

The physical type of the network interface used by wireless Hosted Network.

This is one of the types reported by the related physical interface. This value is correct only if the **HostedNetworkState** member is **wlan_hosted_network_active**.

`ulChannelFrequency`

The channel frequency of the network interface used by wireless Hosted Network.

This value is correct only if **HostedNetworkState** is **wlan_hosted_network_active**.

`dwNumberOfPeers`

The current number of authenticated peers on the wireless Hosted Network.

This value is correct only if **HostedNetworkState** is **wlan_hosted_network_active**.

`PeerList`

An array of WLAN_HOSTED_NETWORK_PEER_STATE structures describing each of the current peers on the wireless Hosted Network. The number of elements in the array is given by **dwNumberOfPeers** member.

This value is correct only if **HostedNetworkState** is **wlan_hosted_network_active**.

Remarks

The **WLAN_HOSTED_NETWORK_STATUS** structure is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and later.

The **WLAN_HOSTED_NETWORK_STATUS** structure is returned in a pointer in the *ppWlanHostedNetworkStatus* parameter by the WlanHostedNetworkQueryStatus function.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

DOT11_MAC_ADDRESS

DOT11_PHY_TYPE

WLAN_HOSTED_NETWORK_PEER_STATE

WLAN_HOSTED_NETWORK_STATE

WlanEnumInterfaces

WlanHostedNetworkQueryStatus

WLAN_INTERFACE_CAPABILITY structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_INTERFACE_CAPABILITY** structure contains information about the capabilities of an interface.

Syntax

```
typedef struct _WLAN_INTERFACE_CAPABILITY {
  WLAN_INTERFACE_TYPE interfaceType;
  BOOL                bDot11DSupported;
  DWORD               dwMaxDesiredSsidListSize;
  DWORD               dwMaxDesiredBssidListSize;
  DWORD               dwNumberOfSupportedPhys;
  DOT11_PHY_TYPE      dot11PhyTypes[WLAN_MAX_PHY_INDEX];
} WLAN_INTERFACE_CAPABILITY, *PWLAN_INTERFACE_CAPABILITY;
```

Members

`interfaceType`

A WLAN_INTERFACE_TYPE value that indicates the type of the interface.

`bDot11DSupported`

Indicates whether 802.11d is supported by the interface. If **TRUE**, 802.11d is supported.

`dwMaxDesiredSsidListSize`

The maximum size of the SSID list supported by this interface.

`dwMaxDesiredBssidListSize`

The maximum size of the basic service set (BSS) identifier list supported by this interface.

`dwNumberOfSupportedPhys`

Contains the number of supported PHY types.

`dot11PhyTypes`

An array of DOT11_PHY_TYPE values that specify the supported PHY types. WLAN_MAX_PHY_INDEX is set to 64.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanGetInterfaceCapability

WLAN_INTERFACE_INFO structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_INTERFACE_INFO** structure contains information about a wireless LAN interface.

Syntax

```
typedef struct _WLAN_INTERFACE_INFO {
  GUID                InterfaceGuid;
  WCHAR               strInterfaceDescription[WLAN_MAX_NAME_LENGTH];
  WLAN_INTERFACE_STATE isState;
} WLAN_INTERFACE_INFO, *PWLAN_INTERFACE_INFO;
```

Members

`InterfaceGuid`

Contains the GUID of the interface.

`strInterfaceDescription`

Contains the description of the interface.

`isState`

Contains a WLAN_INTERFACE_STATE value that indicates the current state of the interface.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_interface_state_connected**, **wlan_interface_state_disconnected**, and **wlan_interface_state_authenticating** values are supported.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_INTERFACE_INFO_LIST

WLAN_INTERFACE_INFO_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_INTERFACE_INFO_LIST** structure contains an array of NIC interface information.

Syntax

```
typedef struct _WLAN_INTERFACE_INFO_LIST {
  DWORD               dwNumberOfItems;
  DWORD               dwIndex;
#if ...
  WLAN_INTERFACE_INFO *InterfaceInfo[];
#else
  WLAN_INTERFACE_INFO InterfaceInfo[1];
#endif
} WLAN_INTERFACE_INFO_LIST, *PWLAN_INTERFACE_INFO_LIST;
```

Members

`dwNumberOfItems`

Contains the number of items in the **InterfaceInfo** member.

`dwIndex`

The index of the current item. The index of the first item is 0. **dwIndex** must be less than **dwNumberOfItems**.

This member is not used by the wireless service. Applications can use this member when processing individual interfaces in the **WLAN_INTERFACE_INFO_LIST** structure. When an application passes this structure from one function to another, it can set the value of **dwIndex** to the index of the item currently being processed. This can help an application maintain state.

dwIndex should always be initialized before use.

`InterfaceInfo`

An array of WLAN_INTERFACE_INFO structures containing interface information.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_INTERFACE_TYPE enumeration (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_INTERFACE_TYPE** enumeration specifies the wireless interface type.

Syntax

```
typedef enum _WLAN_INTERFACE_TYPE {
  wlan_interface_type_emulated_802_11,
  wlan_interface_type_native_802_11,
  wlan_interface_type_invalid
} WLAN_INTERFACE_TYPE, *PWLAN_INTERFACE_TYPE;
```

Constants

`wlan_interface_type_emulated_802_11`	Specifies an emulated 802.11 interface.
`wlan_interface_type_native_802_11`	Specifies a native 802.11 interface.
`wlan_interface_type_invalid`	The interface specified is invalid.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_INTERFACE_CAPABILITY

WLAN_MAC_FRAME_STATISTICS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_MAC_FRAME_STATISTICS** structure contains information about sent and received MAC frames.

Syntax

```
typedef struct WLAN_MAC_FRAME_STATISTICS {
  ULONGLONG ullTransmittedFrameCount;
  ULONGLONG ullReceivedFrameCount;
  ULONGLONG ullWEPExcludedCount;
  ULONGLONG ullTKIPLocalMICFailures;
  ULONGLONG ullTKIPReplays;
  ULONGLONG ullTKIPICVErrorCount;
  ULONGLONG ullCCMPReplays;
  ULONGLONG ullCCMPDecryptErrors;
  ULONGLONG ullWEPUndecryptableCount;
  ULONGLONG ullWEPICVErrorCount;
  ULONGLONG ullDecryptSuccessCount;
  ULONGLONG ullDecryptFailureCount;
} WLAN_MAC_FRAME_STATISTICS, *PWLAN_MAC_FRAME_STATISTICS;
```

Members

`ullTransmittedFrameCount`

Contains the number of successfully transmitted MSDU/MMPDUs.

`ullReceivedFrameCount`

Contains the number of successfully received MSDU/MMPDUs.

`ullWEPExcludedCount`

Contains the number of frames discarded due to having a "Protected" status indicated in the frame control field.

`ullTKIPLocalMICFailures`

Contains the number of MIC failures encountered while checking the integrity of packets received from the AP or peer station.

`ullTKIPReplays`

Contains the number of TKIP replay errors detected.

`ullTKIPICVErrorCount`

Contains the number of TKIP protected packets that the NIC failed to decrypt.

`ullCCMPReplays`

Contains the number of received unicast fragments discarded by the replay mechanism.

`ullCCMPDecryptErrors`

Contains the number of received fragments discarded by the CCMP decryption algorithm.

`ullWEPUndecryptableCount`

Contains the number of WEP protected packets received for which a decryption key was not available on the NIC.

`ullWEPICVErrorCount`

Contains the number of WEP protected packets the NIC failed to decrypt.

`ullDecryptSuccessCount`

Contains the number of encrypted packets that the NIC has successfully decrypted.

`ullDecryptFailureCount`

Contains the number of encrypted packets that the NIC has failed to decrypt.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_STATISTICS

WLAN_MSM_NOTIFICATION_DATA structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_MSM_NOTIFICATION_DATA** structure contains information about media specific module (MSM) connection related notifications.

Syntax

```
typedef struct _WLAN_MSM_NOTIFICATION_DATA {
  WLAN_CONNECTION_MODE wlanConnectionMode;
  WCHAR                strProfileName[WLAN_MAX_NAME_LENGTH];
  DOT11_SSID           dot11Ssid;
  DOT11_BSS_TYPE       dot11BssType;
  DOT11_MAC_ADDRESS    dot11MacAddr;
  BOOL                 bSecurityEnabled;
  BOOL                 bFirstPeer;
  BOOL                 bLastPeer;
  WLAN_REASON_CODE     wlanReasonCode;
} WLAN_MSM_NOTIFICATION_DATA, *PWLAN_MSM_NOTIFICATION_DATA;
```

Members

`wlanConnectionMode`

A WLAN_CONNECTION_MODE value that specifies the mode of the connection.

`strProfileName`

The name of the profile used for the connection. WLAN_MAX_NAME_LENGTH is 256. Profile names are case-sensitive. This string must be NULL-terminated.

`dot11Ssid`

A DOT11_SSID structure that contains the SSID of the association.

`dot11BssType`

A DOT11_BSS_TYPE value that indicates the BSS network type.

`dot11MacAddr`

A DOT11_MAC_ADDRESS that specifies the MAC address of the peer or access point.

`bSecurityEnabled`

Indicates whether security is enabled for this connection. If **TRUE**, security is enabled.

`bFirstPeer`

Indicates whether the peer is the first to join the ad hoc network created by the machine. If **TRUE**, the peer is the first to join.

After the first peer joins the network, the interface state of the machine that created the ad hoc network changes from wlan_interface_state_ad_hoc_network_formed to wlan_interface_state_connected.

`bLastPeer`

Indicates whether the peer is the last to leave the ad hoc network created by the machine. If **TRUE**, the peer is the last to leave. After the last peer leaves the network, the interface state of the machine that created the ad hoc network changes from wlan_interface_state_connected to wlan_interface_state_ad_hoc_network_formed.

`wlanReasonCode`

A WLAN_REASON_CODE that indicates the reason for an operation failure. If the operation succeeds, this field has a value of **WLAN_REASON_CODE_SUCCESS**. Otherwise, this field indicates the reason for the failure.

Remarks

The WlanRegisterNotification function is used by an application to register and unregister notifications on all wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter passed to the **WlanRegisterNotification** function. The prototype for this callback function is the WLAN_NOTIFICATION_CALLBACK. This callback function will receive notifications that have been registered in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function.

The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter that contains detailed information on the notification.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_MSM**, then the received notification is a media specific module (MSM) notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the WLAN_NOTIFICATION_CALLBACK function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For some of these notifications, a **WLAN_MSM_NOTIFICATION_DATA** structure is returned in the *pData* member of **WLAN_NOTIFICATION_DATA** structure.

For more information on these notifications, see the WLAN_NOTIFICATION_MSM enumeration reference.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_NOTIFICATION_CALLBACK

WLAN_NOTIFICATION_DATA

WLAN_NOTIFICATION_MSM

WlanRegisterNotification

WLAN_NOTIFICATION_CALLBACK callback function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WLAN_NOTIFICATION_CALLBACK** callback function prototype defines the type of notification callback function.

Syntax

```
WLAN_NOTIFICATION_CALLBACK WlanNotificationCallback;

void WlanNotificationCallback(
  PWLAN_NOTIFICATION_DATA unnamedParam1,
  PVOID unnamedParam2
)
{...}
```

Parameters

`unnamedParam1`

A pointer to a WLAN_NOTIFICATION_DATA structure that contains the notification information.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the wlan_notification_acm_connection_complete and wlan_notification_acm_disconnected notifications are available.

`unnamedParam2`

A pointer to the context information provided by the client when it registered for the notification.

Return value

None

Remarks

The WlanRegisterNotification function is used by an application to register and unregister notifications on all wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter passed to the **WlanRegisterNotification** function. The prototype for this callback function is the **WLAN_NOTIFICATION_CALLBACK**. This callback function will receive notifications that have been registered in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function.

The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter that contains detailed information on the notification. The callback function also receives a second parameter that contains a pointer to the client context passed in the *pCallbackContext* parameter to the WlanRegisterNotification function. This client context can be a **NULL** pointer if that is what was passed to the **WlanRegisterNotification** function.

Once registered, the callback function will be called whenever a notification is available until the client unregisters or closes the handle.

Any registration to receive notifications is automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) used to register for notifications with the WlanRegisterNotification function or if the process ends.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_ACM**, then the received notification is an auto configuration module notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the **WLAN_NOTIFICATION_CALLBACK** function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For more information on these notifications, see the WLAN_NOTIFICATION_ACM enumeration reference.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_HNWK**, then the received notification is a wireless Hosted Network notification supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the **WLAN_NOTIFICATION_CALLBACK** function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For more information on these notifications, see the WLAN_HOSTED_NETWORK_NOTIFICATION_CODE enumeration reference.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_IHV**, then the received notification is an indepent hardware vendor (IHV) notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the **WLAN_NOTIFICATION_CALLBACK** function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure, which is specific to the IHV.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_ONEX**, then the received notification is an 802.1X module notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the **WLAN_NOTIFICATION_CALLBACK** function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For more information on these notifications, see the ONEX_NOTIFICATION_TYPE enumeration reference.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_MSM**, then the received notification is a media specific module (MSM) notification. The **NotificationCode** member of the **WLAN_NOTIFICATION_DATA** structure passed to the **WLAN_NOTIFICATION_CALLBACK** function determines the interpretation of the *pData* member of **WLAN_NOTIFICATION_DATA** structure. For more information on these notifications, see the WLAN_NOTIFICATION_MSM enumeration reference.

If the **NotificationSource** member of the WLAN_NOTIFICATION_DATA structure received by the callback function is **WLAN_NOTIFICATION_SOURCE_SECURITY**, then the received notification is a security notification. No notifications are currently defined for **WLAN_NOTIFICATION_SOURCE_SECURITY**.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Notifications are handled by the Netman service. If the Netman service is disabled or unavailable, notifications will not be received. If a notification is not received within a reasonable period of time, an application should time out and query the current interface state.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

ONEX_NOTIFICATION_TYPE

WLAN_HOSTED_NETWORK_NOTIFICATION_CODE

WLAN_NOTIFICATION_ACM

WLAN_NOTIFICATION_DATA

WLAN_NOTIFICATION_MSM

WlanRegisterNotification

WLAN_PHY_FRAME_STATISTICS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_PHY_FRAME_STATISTICS** structure contains information about sent and received PHY frames

Syntax

```
typedef struct WLAN_PHY_FRAME_STATISTICS {
  ULONGLONG ullTransmittedFrameCount;
  ULONGLONG ullMulticastTransmittedFrameCount;
  ULONGLONG ullFailedCount;
  ULONGLONG ullRetryCount;
  ULONGLONG ullMultipleRetryCount;
  ULONGLONG ullMaxTXLifetimeExceededCount;
  ULONGLONG ullTransmittedFragmentCount;
  ULONGLONG ullRTSSuccessCount;
  ULONGLONG ullRTSFailureCount;
  ULONGLONG ullACKFailureCount;
  ULONGLONG ullReceivedFrameCount;
  ULONGLONG ullMulticastReceivedFrameCount;
  ULONGLONG ullPromiscuousReceivedFrameCount;
  ULONGLONG ullMaxRXLifetimeExceededCount;
  ULONGLONG ullFrameDuplicateCount;
  ULONGLONG ullReceivedFragmentCount;
  ULONGLONG ullPromiscuousReceivedFragmentCount;
  ULONGLONG ullFCSErrorCount;
} WLAN_PHY_FRAME_STATISTICS, *PWLAN_PHY_FRAME_STATISTICS;
```

Members

`ullTransmittedFrameCount`

Contains the number of successfully transmitted MSDU/MMPDUs.

`ullMulticastTransmittedFrameCount`

Contains the number of successfully transmitted MSDU/MMPDUs in which the multicast bit is set as the destination MAC address.

`ullFailedCount`

Contains the number of MSDU/MMPDUs transmission failures due to the number of transmit attempts exceeding the retry limit.

`ullRetryCount`

Contains the number of MSDU/MMPDUs successfully transmitted after one or more retransmissions.

`ullMultipleRetryCount`

Contains the number of MSDU/MMPDUs successfully transmitted after more than one retransmission.

`ullMaxTXLifetimeExceededCount`

Contains the number of fragmented MSDU/MMPDUs that failed to send due to timeout.

`ullTransmittedFragmentCount`

Contains the number of MPDUs with an individual address in the address 1 field and MPDUs that have a multicast address with types Data or Management.

`ullRTSSuccessCount`

Contains the number of times a CTS has been received in response to an RTS.

`ullRTSFailureCount`

Contains the number of times a CTS has not been received in response to an RTS.

`ullACKFailureCount`

Contains the number of times an expected ACK has not been received.

`ullReceivedFrameCount`

Contains the number of MSDU/MMPDUs successfully received.

`ullMulticastReceivedFrameCount`

Contains the number of successfully received MSDU/MMPDUs with the multicast bit set in the MAC address.

`ullPromiscuousReceivedFrameCount`

Contains the number of MSDU/MMPDUs successfully received only because promicscuous mode is enabled.

`ullMaxRXLifetimeExceededCount`

Contains the number of fragmented MSDU/MMPDUs dropped due to timeout.

`ullFrameDuplicateCount`

Contains the number of frames received that the Sequence Control field indicates as a duplicate.

`ullReceivedFragmentCount`

Contains the number of successfully received Data or Management MPDUs.

`ullPromiscuousReceivedFragmentCount`

Contains the number of MPDUs successfully received only because promiscuous mode is enabled.

`ullFCSErrorCount`

Contains the number of times an FCS error has been detected in a received MPDU.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_PHY_RADIO_STATE structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_PHY_RADIO_STATE** structure specifies the radio state on a specific physical layer (PHY) type.

Syntax

```
typedef struct _WLAN_PHY_RADIO_STATE {
  DWORD             dwPhyIndex;
  DOT11_RADIO_STATE dot11SoftwareRadioState;
  DOT11_RADIO_STATE dot11HardwareRadioState;
} WLAN_PHY_RADIO_STATE, *PWLAN_PHY_RADIO_STATE;
```

Members

`dwPhyIndex`

The index of the PHY type on which the radio state is being set or queried. The WlanGetInterfaceCapability function returns a list of valid PHY types.

`dot11SoftwareRadioState`

A DOT11_RADIO_STATE value that indicates the software radio state.

`dot11HardwareRadioState`

A DOT11_RADIO_STATE value that indicates the hardware radio state.

Remarks

The **WLAN_PHY_RADIO_STATE** structure is used with the WlanSetInterface function when the *OpCode* parameter is set to **wlan_intf_opcode_radio_state**.

The **WLAN_PHY_RADIO_STATE** structure is also used for notification by the media specific module (MSM) when the radio state changes. An application registers to receive MSM notifications by calling the WlanRegisterNotification function with the *dwNotifSource* parameter set to a value that includes **WLAN_NOTIFICATION_SOURCE_MSM**. For more information on these notifications, see the WLAN_NOTIFICATION_DATA structure and the WLAN_NOTIFICATION_MSM enumeration reference.

The radio state of a PHY is off if either **dot11SoftwareRadioState** or **dot11HardwareRadioState** member of the **WLAN_PHY_RADIO_STATE** structure is **dot11_radio_state_off**.

The hardware radio state cannot be changed by calling the WlanSetInterface function. The **dot11HardwareRadioState** member of the **WLAN_PHY_RADIO_STATE** structure is ignored when the **WlanSetInterface** function is called with the *OpCode* parameter set to **wlan_intf_opcode_radio_state** and the *pData* parameter points to a **WLAN_PHY_RADIO_STATE** structure.

The software radio state can be changed by calling the WlanSetInterface function.

Changing the software radio state of a physical network interface could cause related changes in the state of the wireless Hosted Network or virtual wireless adapter radio states. The PHYs of every virtual wireless adapter are linked. For more information, see the About the Wireless Hosted Network.

The radio state of a PHY is off if either the software radio state (**dot11SoftwareRadioState** member) or the hardware radio state (**dot11HardwareRadioState** member) is off.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

About the Wireless Hosted Network

DOT11_RADIO_STATE

WLAN_NOTIFICATION_DATA

WLAN_NOTIFICATION_MSM

WLAN_RADIO_STATE

WlanGetInterfaceCapability

WlanSetInterface

WLAN_PROFILE_INFO structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_PROFILE_INFO** structure contains basic information about a profile.

Syntax

```
typedef struct _WLAN_PROFILE_INFO {
  WCHAR strProfileName[WLAN_MAX_NAME_LENGTH];
  DWORD dwFlags;
} WLAN_PROFILE_INFO, *PWLAN_PROFILE_INFO;
```

Members

`strProfileName`

The name of the profile. This value may be the name of a domain if the profile is for provisioning. Profile names are case-sensitive. This string must be NULL-terminated.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The name of the profile is derived automatically from the SSID of the wireless network. For infrastructure network profiles, the name of the profile is the SSID of the network. For ad hoc network profiles, the name of the profile is the SSID of the ad hoc network followed by `-adhoc`.

`dwFlags`

A set of flags specifying settings for wireless profile. These values are defined in the *Wlanapi.h* header file.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: *dwFlags* must be 0. Per-user profiles are not supported.

Combinations of these flag bits are possible

VALUE	MEANING
WLAN_PROFILE_GROUP_POLICY	This flag indicates that this profile was created by group policy. A group policy profile is read-only. Neither the content nor the preference order of the profile can be changed.
WLAN_PROFILE_USER	This flag indicates that the profile is a per-user profile. If not set, this profile is an all-user profile.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_PROFILE_INFO_LIST

WlanGetProfile

WlanGetProfileList

WLAN_PROFILE_INFO_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_PROFILE_INFO_LIST** structure contains a list of wireless profile information.

Syntax

```
typedef struct _WLAN_PROFILE_INFO_LIST {
  DWORD             dwNumberOfItems;
  DWORD             dwIndex;
#if ...
  WLAN_PROFILE_INFO *ProfileInfo[];
#else
  WLAN_PROFILE_INFO ProfileInfo[1];
#endif
} WLAN_PROFILE_INFO_LIST, *PWLAN_PROFILE_INFO_LIST;
```

Members

`dwNumberOfItems`

The number of wireless profile entries in the **ProfileInfo** member.

`dwIndex`

The index of the current item. The index of the first item is 0. The **dwIndex** member must be less than the **dwNumberOfItems** member.

This member is not used by the wireless service. Applications can use this member when processing individual profiles in the **WLAN_PROFILE_INFO_LIST** structure. When an application passes this structure from one function to another, it can set the value of **dwIndex** to the index of the item currently being processed. This can help an application maintain state.

dwIndex should always be initialized before use.

`ProfileInfo`

An array of WLAN_PROFILE_INFO structures containing interface information. The number of items in the array is specified in the **dwNumberOfItems** member.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_PROFILE_INFO

WlanGetProfile

WlanGetProfileList

WLAN_RADIO_STATE structure (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WLAN_RADIO_STATE** structurespecifies the radio state on a list of physical layer (PHY) types.

Syntax

```
typedef struct _WLAN_RADIO_STATE {
  DWORD                dwNumberOfPhys;
  WLAN_PHY_RADIO_STATE PhyRadioState[WLAN_MAX_PHY_INDEX];
} WLAN_RADIO_STATE, *PWLAN_RADIO_STATE;
```

Members

`dwNumberOfPhys`

The number of valid PHY indices in the **PhyRadioState** member.

`PhyRadioState`

An array of WLAN_PHY_RADIO_STATE structures that specify the radio states of a number of PHY indices. Only the first **dwNumberOfPhys** entries in this array are valid.

Remarks

The **WLAN_RADIO_STATE** structure is used with the WlanQueryInterface function when the *OpCode* parameter is set to **wlan_intf_opcode_radio_state**. If the call is successful, the *ppData* parameter points to a **WLAN_RADIO_STATE** structure.

The WLAN_PHY_RADIO_STATE structure members in the **WLAN_RADIO_STATE** structure can be used with the WlanSetInterface function when the *OpCode* parameter is set to **wlan_intf_opcode_radio_state** to change the radio state.

The WLAN_PHY_RADIO_STATE structure is also used for notification by the media specific module (MSM) when the radio state changes. An application registers to receive MSM notifications by calling the WlanRegisterNotification function with the *dwNotifSource* parameter set to a value that includes **WLAN_NOTIFICATION_SOURCE_MSM**. For more information on these notifications, see the WLAN_NOTIFICATION_DATA structure and the WLAN_NOTIFICATION_MSM enumeration reference.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, queries each interface for the **WLAN_RADIO_STATE** on the interface, and prints values from the retrieved **WLAN_RADIO_STATE** structure.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif
```

```
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")

int wmain()
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;      //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    DWORD dwRetVal = 0;
    int iRet = 0;

    WCHAR GuidString[39] = { 0 };

    unsigned int i;

    // variables used for WlanEnumInterfaces

    PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
    PWLAN_INTERFACE_INFO pIfInfo = NULL;

    // variables used for WlanQueryInterfaces for opcode = wlan_intf_opcode_radio_state
    PWLAN_RADIO_STATE pradioStateInfo = NULL;
    DWORD radioStateInfoSize = sizeof (WLAN_RADIO_STATE);
    WLAN_OPCODE_VALUE_TYPE opCode = wlan_opcode_value_type_invalid;

    dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    }

    dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    } else {
        wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
        wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
        for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
            pIfInfo = (WLAN_INTERFACE_INFO *) & pIfList->InterfaceInfo[i];
            wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
            iRet =
                StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) & GuidString,
                                sizeof (GuidString) / sizeof (*GuidString));
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            //     sizeof(GuidString)/sizeof(*GuidString));
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"  InterfaceGUID[%d]:\t %ws\n", i, GuidString);
            }
```

```
            wprintf(L"  Interface Description[%d]: %ws", i, pIfInfo->strInterfaceDescription);
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            dwResult = WlanQueryInterface(hClient,
                                          &pIfInfo->InterfaceGuid,
                                          wlan_intf_opcode_radio_state,
                                          NULL,
                                          &radioStateInfoSize,
                                          (PVOID *) & pradioStateInfo, &opCode);

            if (dwResult != ERROR_SUCCESS) {
                wprintf(L"WlanQueryInterface failed with error: %u\n", dwResult);
                dwRetVal = 1;
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"  WLAN_RADIO_STATE for this interface\n");

                wprintf(L"  Number of valid PHYs:\t %u\n", pradioStateInfo->dwNumberOfPhys);
                wprintf(L"  Radio state:\n");
                wprintf(L"      Index of PHYs type[0]:\t %u\n",
                        pradioStateInfo->PhyRadioState[0].dwPhyIndex);

                wprintf(L"      Software radio state[0]:\t ");
                switch (pradioStateInfo->PhyRadioState[0].dot11SoftwareRadioState) {
                case dot11_radio_state_unknown:
                    wprintf(L"Unknown\n");
                    break;
                case dot11_radio_state_on:
                    wprintf(L"On\n");
                    break;
                case dot11_radio_state_off:
                    wprintf(L"Off\n");
                    break;
                default:
                    wprintf(L"Other Unknown state %ld\n", pradioStateInfo-
>PhyRadioState[0].dot11SoftwareRadioState);
                    break;
                }
```

```
                wprintf(L"      Hardware radio state[0]:\t ");
                switch (pradioStateInfo->PhyRadioState[0].dot11HardwareRadioState) {
                case dot11_radio_state_unknown:
                    wprintf(L"Unknown\n");
                    break;
                case dot11_radio_state_on:
                    wprintf(L"On\n");
                    break;
                case dot11_radio_state_off:
                    wprintf(L"Off\n");
                    break;
                default:
                    wprintf(L"Other Unknown state %ld\n", pradioStateInfo-
>PhyRadioState[0].dot11HardwareRadioState);
                    break;
                }

                wprintf(L"\n");
            }
        }
    }

if (pradioStateInfo != NULL) {
    WlanFreeMemory(pradioStateInfo);
    pradioStateInfo = NULL;
}

if (pIfList != NULL) {
    WlanFreeMemory(pIfList);
    pIfList = NULL;
}

return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_NOTIFICATION_DATA

WLAN_NOTIFICATION_MSM

WLAN_PHY_RADIO_STATE

WlanQueryInterface

WlanSetInterface

WLAN_RATE_SET structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The set of supported data rates.

Syntax

```
typedef struct _WLAN_RATE_SET {
  ULONG  uRateSetLength;
  USHORT usRateSet[DOT11_RATE_SET_MAX_LENGTH];
} WLAN_RATE_SET, *PWLAN_RATE_SET;
```

Members

`uRateSetLength`

The length, in bytes, of **usRateSet**.

`usRateSet`

An array of supported data transfer rates. DOT11_RATE_SET_MAX_LENGTH is defined in windot11.h to have a value of 126.

Each supported data transfer rate is stored as a USHORT. The first bit of the USHORT specifies whether the rate is a basic rate. A *basic rate* is the data transfer rate that all stations in a basic service set (BSS) can use to receive frames from the wireless medium. If the rate is a basic rate, the first bit of the USHORT is set to 1.

To calculate the data transfer rate in Mbps for an arbitrary array entry rateSet[i], use the following equation:

```
rate_in_mbps = (rateSet[i] & 0x7FFF) * 0.5
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_BSS_ENTRY

WLAN_RAW_DATA structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_RAW_DATA** structure contains raw data in the form of a blob that is used by some Native Wifi functions.

Syntax

```
typedef struct _WLAN_RAW_DATA {
  DWORD dwDataSize;
#if ...
  BYTE  *DataBlob[];
#else
  BYTE  DataBlob[1];
#endif
} WLAN_RAW_DATA, *PWLAN_RAW_DATA;
```

Members

`dwDataSize`

The size, in bytes, of the **DataBlob** member. The maximum value of the **dwDataSize** may be restricted by type of data that is stored in the **WLAN_RAW_DATA** structure.

`DataBlob`

The data blob.

Remarks

The **WLAN_RAW_DATA** structure is a raw data structure used to hold a data entry used by some Native Wifi functions. The data structure is in the form of a generalized blob that can contain any type of data.

The WlanScan function uses the **WLAN_RAW_DATA** structure. The *pleData* parameter passed to the **WlanScan** function points to a **WLAN_RAW_DATA** structure currently used to contain an information element to include in probe requests. This **WLAN_RAW_DATA** structure passed to the **WlanScan** function can contain a proximity service discovery (PSD) information element (IE) data entry.

When the **WLAN_RAW_DATA** structure is used to store a PSD IE, the **DOT11_PSD_IE_MAX_DATA_SIZE** constant defined in the *Wlanapi.h* header file is the maximum value of the **dwDataSize** member.

CONSTANT	VALUE	DESCRIPTION
DOT11_PSD_IE_MAX_DATA_SIZE	240	The maximum data size, in bytes, of a PSD IE data entry.

For more information about PSD IEs, including a discussion of the format of an IE, see the WlanSetPsdIEDataList function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_RAW_DATA_LIST

WlanScan

WlanSetPsdIEDataList

WLAN_RAW_DATA_LIST structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_RAW_DATA_LIST** structure contains raw data in the form of an array of data blobs that are used by some Native Wifi functions.

Syntax

```
typedef struct _WLAN_RAW_DATA_LIST {
  DWORD                   dwTotalSize;
  DWORD                   dwNumberOfItems;
  struct {
    DWORD dwDataOffset;
    DWORD dwDataSize;
  };
  __unnamed_struct_1812_1 DataList[1];
} WLAN_RAW_DATA_LIST, *PWLAN_RAW_DATA_LIST;
```

Members

`dwTotalSize`

The total size, in bytes, of the **WLAN_RAW_DATA_LIST** structure.

`dwNumberOfItems`

The number of raw data entries or blobs in the **WLAN_RAW_DATA_LIST** structure. The maximum value of the **dwNumberOfItems** may be restricted by the type of data that is stored in the **WLAN_RAW_DATA_LIST** structure.

`dwDataOffset`

`dwDataSize`

`DataList`

An array of raw data entries or blobs that make up the data list.

dwDataOffset

The offset, in bytes, of the data blob from the beginning of current blob descriptor. For details, see the example in the Remarks section below.

dwDataSize

The size, in bytes, of the data blob.

Remarks

The **WLAN_RAW_DATA_LIST** structure is used to encapsulate a list of data blobs into a flat memory block. It should be interpreted as a list of headers followed by data blobs.

To create a **WLAN_RAW_DATA_LIST**, an application needs to allocate a memory block that is large enough to hold the headers and the data blobs, and then cast the memory block to a pointer to a **WLAN_RAW_DATA_LIST** structure.

The following is the memory layout of an example **WLAN_RAW_DATA_LIST** structure that contains two data blobs.

Memory Offset	Field	Value	Comments
0	dwTotalSize	84	
4	dwNumberOfItems	2	
8	dwDataOffset	16	Offset of the first blob: 16 = 24 - 8
12	dwDataSize	20	Size of the first blob.
16	dwDataOffset	28	Offset of the second blob: 44 - 16.
20	dwDataSize	24	Size of the second blob.
24		20	Start of the first blob.
44		40	Start of the second blob.

The **WLAN_RAW_DATA_LIST** structure is currently used by the WlanSetPsdIEDataList function to set the proximity service discovery (PSD) information element (IE) data list for an application.

When used to store a PSD IE data list, the **DOT11_PSD_IE_MAX_ENTRY_NUMBER** constant defined in the *Wlanapi.h* header file is the maximum value of the **dwNumberOfItems** member for the number of blobs in the **WLAN_RAW_DATA_LIST** structure. The **DOT11_PSD_IE_MAX_DATA_SIZE** constant defined in the *Wlanapi.h* header file is the maximum value of the **dwDataSize** member for any blob.

CONSTANT	VALUE	DESCRIPTION
DOT11_PSD_IE_MAX_DATA_SIZE	240	The maximum data size, in bytes, of a PSD IE data entry.
DOT11_PSD_IE_MAX_ENTRY_NUMBER	5	The maximum number of PSD IE data entries.

For more information about PSD IEs, including a discussion of the format of an IE, see WlanSetPsdIEDataList.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h (include Wlanapi.h)

See also

WLAN_RAW_DATA

WlanExtractPsdIEDataList

WlanScan

WlanSetPsdIEDataList

WLAN_SECURABLE_OBJECT enumeration (wlanapi.h)

7/1/2021 • 7 minutes to read • Edit Online

The **WLAN_SECURABLE_OBJECT** enumerated type defines the securable objects used by Native Wifi Functions.

These objects can be secured using WlanSetSecuritySettings. The current permissions associated with these objects can be retrieved using WlanGetSecuritySettings. For more information about the use of securable objects, see How DACLs Control Access to an Object.

Syntax

```
typedef enum _WLAN_SECURABLE_OBJECT {
  wlan_secure_permit_list,
  wlan_secure_deny_list,
  wlan_secure_ac_enabled,
  wlan_secure_bc_scan_enabled,
  wlan_secure_bss_type,
  wlan_secure_show_denied,
  wlan_secure_interface_properties,
  wlan_secure_ihv_control,
  wlan_secure_all_user_profiles_order,
  wlan_secure_add_new_all_user_profiles,
  wlan_secure_add_new_per_user_profiles,
  wlan_secure_media_streaming_mode_enabled,
  wlan_secure_current_operation_mode,
  wlan_secure_get_plaintext_key,
  wlan_secure_hosted_network_elevated_access,
  wlan_secure_virtual_station_extensibility,
  wlan_secure_wfd_elevated_access,
  WLAN_SECURABLE_OBJECT_COUNT
} WLAN_SECURABLE_OBJECT, *PWLAN_SECURABLE_OBJECT;
```

Constants

`wlan_secure_permit_list`
The permissions for modifying the permit list for user profiles.

The discretionary access control lists (DACL) associated with this securable object is retrieved when either WlanGetFilterList or WlanSetFilterList is called with *wlanFilterListType* set to **wlan_filter_list_type_user_permit**. For the **WlanGetFilterList** call to succeed, the DACL must contain an access control entry (ACE) that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetFilterList** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_deny_list`
The permissions for modifying the deny list for user profiles. The auto config service will not establish a connection to a network on the deny list.

The DACL associated with this securable object is retrieved when either WlanGetFilterList or WlanSetFilterList is called with *wlanFilterListType* set to **wlan_filter_list_type_user_deny**. For the **WlanGetFilterList** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetFilterList** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_ac_enabled`
The permissions for enabling the auto config service.

The DACL associated with this securable object is retrieved when either WlanQueryInterface or WlanSetInterface is called with *OpCode* set to **wlan_intf_opcode_autoconf_enabled**. For the **WlanQueryInterface** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetInterface** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_bc_scan_enabled`
The permissions for enabling background scans.

The DACL associated with this securable object is retrieved when either WlanQueryInterface or WlanSetInterface is called with *OpCode* set to **wlan_intf_opcode_background_scan_enabled**. For the **WlanQueryInterface** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetInterface** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_bss_type`
The permissions for altering the basic service set type.

The DACL associated with this securable object is retrieved when either WlanQueryInterface or WlanSetInterface is called with *OpCode* set to **wlan_intf_opcode_bss_type**. For the **WlanQueryInterface** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetInterface** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_show_denied`
The permissions for modifying whether networks on the deny list appear in the available networks list.

The DACL associated with this securable object is retrieved when either WlanQueryAutoConfigParameter or WlanSetAutoConfigParameter is called with *OpCode* set to **wlan_autoconf_opcode_show_denied_networks**. For the **WlanQueryAutoConfigParameter** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetAutoConfigParameter** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_interface_properties`
The permissions for changing interface properties.

This is the generic securable object used by WlanQueryInterface or WlanSetInterface when another more specific securable object is not used. Its DACL is retrieved whenever **WlanQueryInterface** or **WlanSetInterface** is access token of the calling thread and the *OpCode* is set to a value other than **wlan_intf_opcode_autoconf_enabled**, **wlan_intf_opcode_background_scan_enabled**, **wlan_intf_opcode_media_streaming_mode**, **wlan_intf_opcode_bss_type**, or **wlan_intf_opcode_current_operation_mode**. The DACL is also not retrieved when *OpCode* is set to **wlan_intf_opcode_radio_state** and the caller is the console user.

For the WlanQueryInterface call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the WlanSetInterface call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_ihv_control`
The permissions for using the WlanIhvControl function for independent hardware vendor (IHV) control of WLAN drivers or services.

The DACL associated with this securable object is retrieved when WlanIhvControl is called. For the call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_all_user_profiles_order`
The permissions for modifying the order of all-user profiles.

The DACL associated with this securable object is retrieved before WlanSetProfileList or WlanSetProfilePosition performs an operation that changes the relative order of all-user profiles in the profile list or moves an all-user profile to a lower position in the profile list. For either call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_add_new_all_user_profiles`
The permissions for adding new all-user profiles.

> **Note** The security descriptor associated with this object is applied to new all-user profiles created.

The DACL associated with this securable object is retrieved when WlanSetProfile is called with *dwFlags* set to 0. For the call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_add_new_per_user_profiles`
The permissions for adding new per-user profiles.

The DACL associated with this securable object is retrieved when WlanSetProfile is called with *dwFlags* set to WLAN_PROFILE_USER. For the call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_media_streaming_mode_enabled`
The permissions for setting or querying the media streaming mode.

The DACL associated with this securable object is retrieved when either WlanQueryInterface or WlanSetInterface is called with *OpCode* set to **wlan_intf_opcode_media_streaming_mode**. For the **WlanQueryInterface** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetInterface** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_current_operation_mode`
The permissions for setting or querying the operation mode of the wireless interface.

The DACL associated with this securable object is retrieved when either WlanQueryInterface or WlanSetInterface is called with *OpCode* set to **wlan_intf_opcode_current_operation_mode**. For the **WlanQueryInterface** call to succeed, the DACL must contain an ACE that grants WLAN_READ_ACCESS permission to the access token of the calling thread. For the **WlanSetInterface** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread.

`wlan_secure_get_plaintext_key`
The permissions for retrieving the plain text key from a wireless profile.

The DACL associated with this securable object is retrieved when the WlanGetProfile function is called with the **WLAN_PROFILE_GET_PLAINTEXT_KEY** flag set in the value pointed to by the *pdwFlags* parameter on input. For the **WlanGetProfile** call to succeed, the DACL must contain an ACE that grants **WLAN_READ_ACCESS** permission to the access token of the calling thread. By default, the permissions for retrieving the plain text key is allowed only to the members of the Administrators group on a local computer.

Windows 7: This value is an extension to native wireless APIs added on Windows 7 and later.

`wlan_secure_hosted_network_elevated_access`
The permissions that have elevated access to call the privileged Hosted Network functions.

The DACL associated with this securable object is retrieved when the WlanHostedNetworkSetProperty function is called with the *OpCode* parameter set to **wlan_hosted_network_opcode_enable**. For the **WlanHostedNetworkSetProperty** call to succeed, the DACL must contain an ACE that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread. By default, the permission to set the wireless Hosted Network property to **wlan_hosted_network_opcode_enable** is allowed only to the members of the Administrators group on a local computer.

The DACL associated with this securable object is retrieved when the WlanHostedNetworkForceStart function is called. For the **WlanHostedNetworkForceStart** call to succeed, the DACL must contain an ACE that grants **WLAN_WRITE_ACCESS** permission to the access token of the calling thread. By default, the permission to force start the wireless Hosted Network is allowed only to the members of the Administrators group on a local computer.

Windows 7: This value is an extension to native wireless APIs added on Windows 7 and later.

`wlan_secure_virtual_station_extensibility`
Windows 7: This value is an extension to native wireless APIs added on Windows 7 and later.

`wlan_secure_wfd_elevated_access`
This value is reserved for internal use by the Wi-Fi Direct service.

Windows 8: This value is an extension to native wireless APIs added on Windows 8 and later.

`WLAN_SECURABLE_OBJECT_COUNT`

Remarks

These objects can be secured using WlanSetSecuritySettings. The current permissions associated with these objects can be retrieved using WlanGetSecuritySettings. For more information about the use of securable objects, see How DACLs Control Access to an Object and Native Wifi API Permissions.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

How DACLs Control Access to an Object

Native Wifi API Permissions

WlanGetFilterList

WlanGetProfile

WlanHostedNetworkForceStart

WlanHostedNetworkSetProperty

WlanIhvControl

WlanQueryAutoConfigParameter

WlanQueryInterface

WlanSetAutoConfigParameter

WlanSetFilterList

WlanSetInterface

WlanSetProfile

WlanSetProfileList

WlanSetProfilePosition

WLAN_SECURITY_ATTRIBUTES structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_SECURITY_ATTRIBUTES** structure defines the security attributes for a wireless connection.

Syntax

```
typedef struct _WLAN_SECURITY_ATTRIBUTES {
  BOOL                    bSecurityEnabled;
  BOOL                    bOneXEnabled;
  DOT11_AUTH_ALGORITHM    dot11AuthAlgorithm;
  DOT11_CIPHER_ALGORITHM dot11CipherAlgorithm;
} WLAN_SECURITY_ATTRIBUTES, *PWLAN_SECURITY_ATTRIBUTES;
```

Members

`bSecurityEnabled`

Indicates whether security is enabled for this connection.

`bOneXEnabled`

Indicates whether 802.1X is enabled for this connection.

`dot11AuthAlgorithm`

A DOT11_AUTH_ALGORITHM value that identifies the authentication algorithm.

`dot11CipherAlgorithm`

A DOT11_CIPHER_ALGORITHM value that identifies the cipher algorithm.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_CONNECTION_ATTRIBUTES

WLAN_STATISTICS structure (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WLAN_STATISTICS** structure contains assorted statistics about an interface.

Syntax

```
typedef struct WLAN_STATISTICS {
  ULONGLONG                 ullFourWayHandshakeFailures;
  ULONGLONG                 ullTKIPCounterMeasuresInvoked;
  ULONGLONG                 ullReserved;
  WLAN_MAC_FRAME_STATISTICS MacUcastCounters;
  WLAN_MAC_FRAME_STATISTICS MacMcastCounters;
  DWORD                     dwNumberOfPhys;
#if ...
  WLAN_PHY_FRAME_STATISTICS *PhyCounters[];
#else
  WLAN_PHY_FRAME_STATISTICS PhyCounters[1];
#endif
} WLAN_STATISTICS, *PWLAN_STATISTICS;
```

Members

`ullFourWayHandshakeFailures`

Indicates the number of 4-way handshake failures. This member is only valid if IHV Service is being used as the authentication service for the current network.

`ullTKIPCounterMeasuresInvoked`

Indicates the number of TKIP countermeasures performed by an IHV Miniport driver. This count does not include TKIP countermeasures invoked by the operating system.

`ullReserved`

Reserved for use by Microsoft.

`MacUcastCounters`

A WLAN_MAC_FRAME_STATISTICS structure that contains MAC layer counters for unicast packets directed to the receiver of the NIC.

`MacMcastCounters`

A WLAN_MAC_FRAME_STATISTICS structure that contains MAC layer counters for multicast packets directed to the current multicast address.

`dwNumberOfPhys`

Contains the number of **WLAN_PHY_FRAME_STATISTICS** structures in the **PhyCounters** member.

`PhyCounters`

An array of WLAN_PHY_FRAME_STATISTICS structures that contain PHY layer counters.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	wlanapi.h

See also

WLAN_MAC_FRAME_STATISTICS

WLAN_PHY_FRAME_STATISTICS

WlanQueryInterface

WlanAllocateMemory function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanAllocateMemory** function allocates memory. Any memory passed to other Native Wifi functions must be allocated with this function.

Syntax

```
PVOID WlanAllocateMemory(
  DWORD dwMemorySize
);
```

Parameters

`dwMemorySize`

Amount of memory being requested, in bytes.

Return value

If the call is successful, the function returns a pointer to the allocated memory.

If the memory could not be allocated for any reason or if the *dwMemorySize* parameter is 0, the returned pointer is **NULL**.

An application can call GetLastError to obtain extended error information.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanFreeMemory

WlanCloseHandle function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanCloseHandle** function closes a connection to the server.

Syntax

```
DWORD WlanCloseHandle(
  HANDLE hClientHandle,
  PVOID  pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, which identifies the connection to be closed. This handle was obtained by a previous call to the WlanOpenHandle function.

`pReserved`

Reserved for future use. Set this parameter to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.

Remarks

After a connection has been closed, any attempted use of the closed handle can cause unexpected errors. Upon closing, all outstanding notifications are discarded.

Do not call **WlanCloseHandle** from a callback function. If the client is in the middle of a notification callback when **WlanCloseHandle** is called, the function waits for the callback to finish before returning a value. Calling this function inside a callback function will result in the call never completing. If both the callback function and the thread that closes the handle try to acquire the same lock, a deadlock may occur. In addition, do not call **WlanCloseHandle** from the **DllMain** function in an application DLL. This could also cause a deadlock.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanOpenHandle

WlanConnect function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanConnect** function attempts to connect to a specific network.

Syntax

```
DWORD WlanConnect(
  HANDLE                           hClientHandle,
  const GUID                       *pInterfaceGuid,
  const PWLAN_CONNECTION_PARAMETERS pConnectionParameters,
  PVOID                            pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface to use for the connection.

`pConnectionParameters`

Pointer to a WLAN_CONNECTION_PARAMETERS structure that specifies the connection type, mode, network profile, SSID that identifies the network, and other parameters.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: There are some constraints on the WLAN_CONNECTION_PARAMETERS members. This means that structures that are valid for Windows Server 2008 and Windows Vista may not be valid for Windows XP with SP3 or Wireless LAN API for Windows XP with SP2. For a list of constraints, see **WLAN_CONNECTION_PARAMETERS**.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_INVALID_PARAMETER	One of the following conditions occurred: • *hClientHandle* is **NULL** or invalid. • *pInterfaceGuid* is **NULL**. • *pConnectionParameters* is **NULL**. • The **dwFlags** member of the structure pointed to by *pConnectionParameters* is not set to one of the values specified on the WLAN_CONNECTION_PARAMETERS page. • The **wlanConnectionMode** member of the structure pointed to by *pConnectionParameters* is set to **wlan_connection_mode_discovery_secure** or **wlan_connection_mode_discovery_unsecure**, and the **pDot11Ssid** member of the same structure is **NULL**. • The **wlanConnectionMode** member of the structure pointed to by *pConnectionParameters* is set to **wlan_connection_mode_discovery_secure** or **wlan_connection_mode_discovery_unsecure**, and the **dot11BssType** member of the same structure is set to **dot11_BSS_type_any**. • The **wlanConnectionMode** member of the structure pointed to by *pConnectionParameters* is set to **wlan_connection_mode_profile**, and the **strProfile** member of the same structure is **NULL** or the length of the profile exceeds WLAN_MAX_NAME_LENGTH. • The **wlanConnectionMode** member of the structure pointed to by *pConnectionParameters* is set to **wlan_connection_mode_profile**, and the **strProfile** member of the same structure is **NULL** or the length of the profile is zero. • The **wlanConnectionMode** member of the structure pointed to by *pConnectionParameters* is set to **wlan_connection_mode_invalid** or **wlan_connection_mode_auto**. • The **dot11BssType** member of the structure pointed to by *pConnectionParameters* is set to **dot11_BSS_type_infrastructure**, and the **dwFlags** member of the same structure is set to **WLAN_CONNECTION_ADHOC_JOIN_ONLY**. • The **dot11BssType** member of the structure pointed to by *pConnectionParameters* is set to **dot11_BSS_type_independent**, and the **dwFlags** member of the same structure is set to **WLAN_CONNECTION_HIDDEN_NETWORK**. • The **dwFlags** member of the structure pointed to by *pConnectionParameters* is set to **WLAN_CONNECTION_IGNORE_PRIVACY_BIT**, and either the **wlanConnectionMode** member of the same structure is not set to **wlan_connection_mode_temporary_profile** or the **dot11BssType** member of the same structure is set to **dot11_BSS_type_independent**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions.

Remarks

The **WlanConnect** function returns immediately. To be notified when a connection is established or when no further connections will be attempted, a client must register for notifications by calling WlanRegisterNotification.

The **strProfile** member of the WLAN_CONNECTION_PARAMETERS structure pointed to by *pConnectionParameters* specifies the profile to use for connection. If this profile is an all-user profile, the **WlanConnect** caller must have execute access on the profile. Otherwise, the **WlanConnect** call will fail with return value ERROR_ACCESS_DENIED. The permissions on an all-user profile are established when the profile is created or saved using WlanSetProfile or WlanSaveTemporaryProfile.

To perform a connection operation at the command line, use the **netsh wlan connect** command. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: You can only use **WlanConnect** to connect to networks on the preferred network list. To add a network to the preferred network list, call WlanSetProfile.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_CONNECTION_PARAMETERS

WlanDisconnect

WlanDeleteProfile function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanDeleteProfile** function deletes a wireless profile for a wireless interface on the local computer.

Syntax

```
DWORD WlanDeleteProfile(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface from which to delete the profile.

`strProfileName`

The name of the profile to be deleted. Profile names are case-sensitive. This string must be NULL-terminated.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The supplied name must match the profile name derived automatically from the SSID of the network. For an infrastructure network profile, the SSID must be supplied for the profile name. For an ad hoc network profile, the supplied name must be the SSID of the ad hoc network followed by `-adhoc`.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	The *hClientHandle* parameter is **NULL** or not valid, the *pInterfaceGuid* parameter is **NULL**, the *strProfileName* parameter is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle specified in the *hClientHandle* parameter was not found in the handle table.

ERROR_NOT_FOUND	The wireless profile specified by *strProfileName* was not found in the profile store.
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to delete the profile.
RPC_STATUS	Various error codes.

Remarks

The **WlanDeleteProfile** function deletes a wireless profile for a wireless interface on the local computer.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanDeleteProfile** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter for the wireless LAN profile has been removed from the system (a USB wireless adapter that has been removed, for example).

To delete a profile at the command line, use the **netsh wlan delete profile** command. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Examples

The following example enumerates the wireless LAN interfaces on the local computer and tries to delete a specific wireless profile on each wireless LAN interface.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")

int _cdecl wmain(int argc, WCHAR ** argv)
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;      //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    DWORD dwRetVal = 0;
```

```
int iRet = 0;

WCHAR GuidString[39] = { 0 };

unsigned int i;

/* variables used for WlanEnumInterfaces  */

PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
PWLAN_INTERFACE_INFO pIfInfo = NULL;

LPCWSTR pProfileName = NULL;

// Validate the parameters
if (argc < 2) {
    wprintf(L"usage: %s <profile>\n", argv[0]);
    wprintf(L"   Deletes a wireless profile\n");
    wprintf(L"   Example\n");
    wprintf(L"       %s \"Default Wireless\"\n", argv[0]);
    exit(1);
}

pProfileName = argv[1];

wprintf(L"Information for profile: %ws\n\n", pProfileName);

dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
}

dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
} else {
    wprintf(L"WLAN_INTERFACE_INFO_LIST for this system\n");

    wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
    wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
    for (i = 0; i < pIfList->dwNumberOfItems; i++) {
        pIfInfo = (WLAN_INTERFACE_INFO *) & pIfList->InterfaceInfo[i];
        wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
        iRet =
            StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) & GuidString,
                            sizeof (GuidString) / sizeof (*GuidString));
        // For c rather than C++ source code, the above line needs to be
        // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
        //     sizeof(GuidString)/sizeof(*GuidString));
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else {
            wprintf(L"  InterfaceGUID[%d]: %ws\n", i, GuidString);
        }
        wprintf(L"  Interface Description[%d]: %ws", i,
                pIfInfo->strInterfaceDescription);
        wprintf(L"\n");
        wprintf(L"  Interface State[%d]:\t ", i);
        switch (pIfInfo->isState) {
        case wlan_interface_state_not_ready:
            wprintf(L"Not ready\n");
            break;
        case wlan_interface_state_connected:
            wprintf(L"Connected\n");
            break;
        case wlan_interface_state_ad_hoc_network_formed:
```

```
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf
                    (L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            dwResult = WlanDeleteProfile(hClient,
                                         &pIfInfo->InterfaceGuid,
                                         pProfileName, NULL);

            if (dwResult != ERROR_SUCCESS) {
                wprintf
                    (L"WlanDeleteProfile failed on this interface with error: %u\n",
                     dwResult);
                if (dwResult == ERROR_NOT_FOUND)
                    wprintf
                        (L"  Error was the following profile was not found: %ws\n",
                         pProfileName);
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"Successfully deleted Profile Name: %ws\n",
                        pProfileName);
            }
            wprintf(L"\n");
        }

    }
    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

Native Wifi API Permissions

WlanGetProfile

WlanGetProfileList

WlanRenameProfile

WlanSetProfile

WlanDeviceServiceCommand function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Allows an original equipment manufacturer (OEM) or independent hardware vendor (IHV) component to communicate with a device service on a particular wireless LAN interface.

Syntax

```
DWORD WlanDeviceServiceCommand(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPGUID     pDeviceServiceGuid,
  DWORD      dwOpCode,
  DWORD      dwInBufferSize,
  PVOID      pInBuffer,
  DWORD      dwOutBufferSize,
  PVOID      pOutBuffer,
  PDWORD     pdwBytesReturned
);
```

Parameters

`hClientHandle`

Type: HANDLE

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

Type: CONST GUID*

A pointer to the **GUID** of the wireless LAN interface to be queried. You can determine the **GUID** of each wireless LAN interface enabled on a local computer by using the WlanEnumInterfaces function.

`pDeviceServiceGuid`

Type: GUID*

The **GUID** identifying the device service for this command.

`dwOpCode`

Type: DWORD

The operational code identifying the operation to be performed on the device service.

`dwInBufferSize`

Type: DWORD

The size, in bytes, of the input buffer.

`pInBuffer`

Type: PVOID

A generic buffer for command input.

`dwOutBufferSize`

Type: **DWORD**

The size, in bytes, of the output buffer.

`pOutBuffer`

Type: **PVOID**

A generic buffer for command output.

`pdwBytesReturned`

Type: **PDWORD**

The number of bytes returned.

Return value

Type: **HRESULT**

If the function succeeds, the return value is **ERROR_SUCCESS**. If the function fails with **ERROR_ACCESS_DENIED**, then the caller doesn't have sufficient permissions to perform this operation. The caller needs to either have admin privilege, or needs to be a UMDF driver.

Requirements

Header	wlanapi.h

WlanDisconnect function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanDisconnect** function disconnects an interface from its current network.

Syntax

```
DWORD WlanDisconnect(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface to be disconnected.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pInterfaceGuid* is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.
ERROR_NOT_ENOUGH_MEMORY	Failed to allocate memory for the query results.
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions.

Remarks

When the connection was established using WlanConnect, a profile was specified by the **strProfile** member of the WLAN_CONNECTION_PARAMETERS structure pointed to by *pConnectionParameters*. If that profile was an all-user profile, the **WlanDisconnect** caller must have execute access on the profile. Otherwise, the **WlanDisconnect** call will fail with return value ERROR_ACCESS_DENIED. The permissions on an all-user profile are established when the profile is created or saved using WlanSetProfile or WlanSaveTemporaryProfile.

To perform a disconnection operation at the command line, use the **netsh wlan disconnect** command. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: **WlanDisconnect** has the side effect of modifying the profile associated with the disconnected network. A network profile becomes an on-demand profile after a **WlanDisconnect** call. The Wireless Zero Configuration service will not connect automatically to a network with an on-demand profile when the network is in range. Do not call **WlanDisconnect** before calling WlanConnect unless you want to change a profile to an on-demand profile. When you call **WlanConnect** to establish a network connection, any existing network connection is dropped automatically.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanConnect

WlanEnumInterfaces function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanEnumInterfaces** function enumerates all of the wireless LAN interfaces currently enabled on the local computer.

Syntax

```
DWORD WlanEnumInterfaces(
  HANDLE                    hClientHandle,
  PVOID                     pReserved,
  PWLAN_INTERFACE_INFO_LIST *ppInterfaceList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pReserved`

Reserved for future use. This parameter must be set to **NULL**.

`ppInterfaceList`

A pointer to storage for a pointer to receive the returned list of wireless LAN interfaces in a WLAN_INTERFACE_INFO_LIST structure.

The buffer for the WLAN_INTERFACE_INFO_LIST returned is allocated by the **WlanEnumInterfaces** function if the call succeeds.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if the *hClientHandle* or *ppInterfaceList* parameter is **NULL**. This error is returned if the *pReserved* is not **NULL**. This error is also returned if the *hClientHandle* parameter is not valid.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.

ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to process this request and allocate memory for the query results.

Remarks

The **WlanEnumInterfaces** function allocates memory for the list of returned interfaces that is returned in the buffer pointed to by the *ppInterfaceList* parameter when the function succeeds. The memory used for the buffer pointed to by *ppInterfaceList* parameter should be released by calling the WlanFreeMemory function after the buffer is no longer needed.

Examples

The following example enumerates the wireless LAN interfaces on the local computer and prints values from the retrieved WLAN_INTERFACE_INFO_LIST structure and the enumerated WLAN_INTERFACE_INFO structures.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")

int wmain()
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;   //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    int iRet = 0;

    WCHAR GuidString[40] = {0};

    int i;

    /* variables used for WlanEnumInterfaces  */

    PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
    PWLAN_INTERFACE_INFO pIfInfo = NULL;


    dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
    if (dwResult != ERROR_SUCCESS)  {
        wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
        // FormatMessage can be used to find out why the function failed
        return 1;
    }
```

```
    }

    dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
    if (dwResult != ERROR_SUCCESS)  {
        wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
        // FormatMessage can be used to find out why the function failed
        return 1;
    }
    else {
        wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
        wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
        for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
            pIfInfo = (WLAN_INTERFACE_INFO *) &pIfList->InterfaceInfo[i];
            wprintf(L"  Interface Index[%d]:\t %lu\n", i, i);
            iRet = StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString, 39);
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString, 39);
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"  InterfaceGUID[%d]: %ws\n",i, GuidString);
            }
            wprintf(L"  Interface Description[%d]: %ws", i,
                pIfInfo->strInterfaceDescription);
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");
        }
    }

    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }
    return 0;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_INTERFACE_INFO

WLAN_INTERFACE_INFO_LIST

WlanFreeMemory

WlanExtractPsdIEDataList function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanExtractPsdIEDataList** function extracts the proximity service discovery (PSD) information element (IE) data list from raw IE data included in a beacon.

Syntax

```
DWORD WlanExtractPsdIEDataList(
  HANDLE              hClientHandle,
  DWORD               dwIeDataSize,
  const PBYTE         pRawIeData,
  LPCWSTR             strFormat,
  PVOID               pReserved,
  PWLAN_RAW_DATA_LIST *ppPsdIEDataList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`dwIeDataSize`

The size, in bytes, of the *pRawIeData* parameter.

`pRawIeData`

The raw IE data for all IEs in the list.

`strFormat`

Describes the format of a PSD IE. Only IEs with a matching format are returned.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`ppPsdIEDataList`

A pointer to a PWLAN_RAW_DATA_LIST structure that contains the formatted data list.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *dwIeDataSize* is 0, *pRawIeData* is **NULL**, or *pReserved* is not **NULL**.

ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

For more information about PSD IEs, including a discussion of the format of an IE, see WlanSetPsdIEDataList.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_RAW_DATA_LIST

WlanSetPsdIEDataList

WlanFreeMemory function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanFreeMemory** function frees memory. Any memory returned from Native Wifi functions must be freed.

Syntax

```
void WlanFreeMemory(
  PVOID pMemory
);
```

Parameters

`pMemory`

Pointer to the memory to be freed.

Return value

None

Remarks

If *pMemory* points to memory that has already been freed, an access violation or heap corruption may occur.

There is a hotfix available for Wireless LAN API for Windows XP with Service Pack 2 (SP2) that can help improve the performance of applications that call **WlanFreeMemory** and WlanGetAvailableNetworkList many times. For more information, see Help and Knowledge Base article 940541, entitled "FIX: The private bytes of the application continuously increase when an application calls the WlanGetAvailableNetworkList function and the WlanFreeMemory function on a Windows XP Service Pack 2-based computer", in the Help and Support Knowledge Base at https://go.microsoft.com/fwlink/p/?linkid=102216.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanAllocateMemory

WlanGetAvailableNetworkList function (wlanapi.h)

7/1/2021 • 6 minutes to read • Edit Online

The **WlanGetAvailableNetworkList** function retrieves the list of available networks on a wireless LAN interface.

Syntax

```
DWORD WlanGetAvailableNetworkList(
  HANDLE                      hClientHandle,
  const GUID                  *pInterfaceGuid,
  DWORD                       dwFlags,
  PVOID                       pReserved,
  PWLAN_AVAILABLE_NETWORK_LIST *ppAvailableNetworkList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

A pointer to the GUID of the wireless LAN interface to be queried.

The GUID of each wireless LAN interface enabled on a local computer can be determined using the WlanEnumInterfaces function.

`dwFlags`

A set of flags that control the type of networks returned in the list. This parameter can be a combination of these possible values.

VALUE	MEANING
WLAN_AVAILABLE_NETWORK_INCLUDE_ALL_ADHOC_PROFILES 0x00000001	Include all ad hoc network profiles in the available network list, including profiles that are not visible. **Note** If this flag is specified on Windows XP with SP3 and Wireless LAN API for Windows XP with SP2, it is consider considered an invalid parameter.
WLAN_AVAILABLE_NETWORK_INCLUDE_ALL_MANUAL_HIDDEN_PROFILES 0x00000002	Include all hidden network profiles in the available network list, including profiles that are not visible. **Note** If this flag is specified on Windows XP with SP3 and Wireless LAN API for Windows XP with SP2, it is consider considered an invalid parameter.

`pReserved`

Reserved for future use. This parameter must be set to **NULL**.

`ppAvailableNetworkList`

A pointer to storage for a pointer to receive the returned list of visible networks in a WLAN_AVAILABLE_NETWORK_LIST structure.

The buffer for the WLAN_AVAILABLE_NETWORK_LIST returned is allocated by the **WlanGetAvailableNetworkList** function if the call succeeds.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if the *hClientHandle*, *pInterfaceGuid*, or *ppAvailableNetworkList* parameter is **NULL**. This error is returned if the *pReserved* is not **NULL**. This error is also returned if the *dwFlags* parameter value is set to value that is not valid or the *hClientHandle* parameter is not valid.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NDIS_DOT11_POWER_STATE_INVALID	The radio associated with the interface is turned off. There are no available networks when the radio is off.
RPC_STATUS	Various error codes.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to process this request and allocate memory for the query results.

Remarks

The **WlanGetAvailableNetworkList** function allocates memory for the list of available networks returned in the buffer pointed to by the *ppAvailableNetworkList* parameter when the function succeeds. The memory used for the buffer pointed to by *ppAvailableNetworkList* parameter should be released by calling the WlanFreeMemory function after the buffer is no longer needed.

There is a hotfix available for Wireless LAN API for Windows XP with SP2 that can help improve the performance of applications that call WlanFreeMemory and **WlanGetAvailableNetworkList** many times. For more information, see Help and Knowledge Base article 940541, entitled "FIX: The private bytes of the application continuously increase when an application calls the WlanGetAvailableNetworkList function and the WlanFreeMemory function on a Windows XP Service Pack 2-based computer", in the Help and Support Knowledge Base at https://go.microsoft.com/fwlink/p/?linkid=102216.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, retrieves the list of available networks on each wireless LAN interface, and prints values from the retrieved

WLAN_AVAILABLE_NETWORK_LIST that contains the WLAN_AVAILABLE_NETWORK entries.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")


int wmain()
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;      //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    DWORD dwRetVal = 0;
    int iRet = 0;

    WCHAR GuidString[39] = {0};

    unsigned int i, j, k;

    /* variables used for WlanEnumInterfaces  */

    PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
    PWLAN_INTERFACE_INFO pIfInfo = NULL;

    PWLAN_AVAILABLE_NETWORK_LIST pBssList = NULL;
    PWLAN_AVAILABLE_NETWORK pBssEntry = NULL;

    int iRSSI = 0;

    dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    }

    dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    } else {
        wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
        wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
        for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
```

```
            pIfInfo = (WLAN_INTERFACE_INFO *) &pIfList->InterfaceInfo[i];
            wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
            iRet = StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
                sizeof(GuidString)/sizeof(*GuidString));
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            //     sizeof(GuidString)/sizeof(*GuidString));
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"  InterfaceGUID[%d]: %ws\n",i, GuidString);
            }
            wprintf(L"  Interface Description[%d]: %ws", i,
                pIfInfo->strInterfaceDescription);
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            dwResult = WlanGetAvailableNetworkList(hClient,
                                           &pIfInfo->InterfaceGuid,
                                           0,
                                           NULL,
                                           &pBssList);

            if (dwResult != ERROR_SUCCESS) {
                wprintf(L"WlanGetAvailableNetworkList failed with error: %u\n",
                        dwResult);
                dwRetVal = 1;
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"WLAN_AVAILABLE_NETWORK_LIST for this interface\n");

                wprintf(L"  Num Entries: %lu\n\n", pBssList->dwNumberOfItems);

                for (j = 0; j < pBssList->dwNumberOfItems; j++) {
                    pBssEntry =
                        (WLAN_AVAILABLE_NETWORK *) & pBssList->Network[j];

                    wprintf(L"  Profile Name[%u]:  %ws\n", j, pBssEntry->strProfileName);
```

```
                    wprintf(L"  SSID[%u]:\t\t ", j);
                    if (pBssEntry->dot11Ssid.uSSIDLength == 0)
                        wprintf(L"\n");
                    else {
                        for (k = 0; k < pBssEntry->dot11Ssid.uSSIDLength; k++) {
                            wprintf(L"%c", (int) pBssEntry->dot11Ssid.ucSSID[k]);
                        }
                        wprintf(L"\n");
                    }

                    wprintf(L"  BSS Network type[%u]:\t ", j);
                    switch (pBssEntry->dot11BssType) {
                    case dot11_BSS_type_infrastructure   :
                        wprintf(L"Infrastructure (%u)\n", pBssEntry->dot11BssType);
                        break;
                    case dot11_BSS_type_independent:
                        wprintf(L"Infrastructure (%u)\n", pBssEntry->dot11BssType);
                        break;
                    default:
                        wprintf(L"Other (%lu)\n", pBssEntry->dot11BssType);
                        break;
                    }

                    wprintf(L"  Number of BSSIDs[%u]:\t %u\n", j, pBssEntry->uNumberOfBssids);

                    wprintf(L"  Connectable[%u]:\t ", j);
                    if (pBssEntry->bNetworkConnectable)
                        wprintf(L"Yes\n");
                    else {
                        wprintf(L"No\n");
                        wprintf(L"  Not connectable WLAN_REASON_CODE value[%u]:\t %u\n", j,
                            pBssEntry->wlanNotConnectableReason);
                    }

                    wprintf(L"  Number of PHY types supported[%u]:\t %u\n", j, pBssEntry-
>uNumberOfPhyTypes);

                    if (pBssEntry->wlanSignalQuality == 0)
                        iRSSI = -100;
                    else if (pBssEntry->wlanSignalQuality == 100)
                        iRSSI = -50;
                    else
                        iRSSI = -100 + (pBssEntry->wlanSignalQuality/2);

                    wprintf(L"  Signal Quality[%u]:\t %u (RSSI: %i dBm)\n", j,
                        pBssEntry->wlanSignalQuality, iRSSI);

                    wprintf(L"  Security Enabled[%u]:\t ", j);
                    if (pBssEntry->bSecurityEnabled)
                        wprintf(L"Yes\n");
                    else
                        wprintf(L"No\n");

                    wprintf(L"  Default AuthAlgorithm[%u]: ", j);
                    switch (pBssEntry->dot11DefaultAuthAlgorithm) {
                    case DOT11_AUTH_ALGO_80211_OPEN:
                        wprintf(L"802.11 Open (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_80211_SHARED_KEY:
                        wprintf(L"802.11 Shared (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_WPA:
                        wprintf(L"WPA (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_WPA_PSK:
                        wprintf(L"WPA-PSK (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_WPA_NONE:
                        wprintf(L"WPA-None (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
```

```
                        wprintf(L"WPA NONE (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_RSNA:
                        wprintf(L"RSNA (%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    case DOT11_AUTH_ALGO_RSNA_PSK:
                        wprintf(L"RSNA with PSK(%u)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    default:
                        wprintf(L"Other (%lu)\n", pBssEntry->dot11DefaultAuthAlgorithm);
                        break;
                    }

                    wprintf(L"  Default CipherAlgorithm[%u]: ", j);
                    switch (pBssEntry->dot11DefaultCipherAlgorithm) {
                    case DOT11_CIPHER_ALGO_NONE:
                        wprintf(L"None (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    case DOT11_CIPHER_ALGO_WEP40:
                        wprintf(L"WEP-40 (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    case DOT11_CIPHER_ALGO_TKIP:
                        wprintf(L"TKIP (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    case DOT11_CIPHER_ALGO_CCMP:
                        wprintf(L"CCMP (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    case DOT11_CIPHER_ALGO_WEP104:
                        wprintf(L"WEP-104 (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    case DOT11_CIPHER_ALGO_WEP:
                        wprintf(L"WEP (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    default:
                        wprintf(L"Other (0x%x)\n", pBssEntry->dot11DefaultCipherAlgorithm);
                        break;
                    }

                    wprintf(L"  Flags[%u]:\t 0x%x", j, pBssEntry->dwFlags);
                    if (pBssEntry->dwFlags) {
                        if (pBssEntry->dwFlags & WLAN_AVAILABLE_NETWORK_CONNECTED)
                            wprintf(L" - Currently connected");
                        if (pBssEntry->dwFlags & WLAN_AVAILABLE_NETWORK_HAS_PROFILE)
                            wprintf(L" - Has profile");
                    }
                    wprintf(L"\n");

                    wprintf(L"\n");
                }
            }
        }

    }
    if (pBssList != NULL) {
        WlanFreeMemory(pBssList);
        pBssList = NULL;
    }

    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_AVAILABLE_NETWORK

WLAN_AVAILABLE_NETWORK_LIST

WLAN_BSS_ENTRY

WLAN_BSS_LIST

WlanEnumInterfaces

WlanFreeMemory

WlanGetNetworkBssList

WlanScan

WlanGetFilterList function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanGetFilterList** function retrieves a group policy or user permission list.

Syntax

```
DWORD WlanGetFilterList(
  HANDLE                hClientHandle,
  WLAN_FILTER_LIST_TYPE wlanFilterListType,
  PVOID                 pReserved,
  PDOT11_NETWORK_LIST   *ppNetworkList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`wlanFilterListType`

A WLAN_FILTER_LIST_TYPE value that specifies the type of filter list. All user defined and group policy filter lists can be queried.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`ppNetworkList`

Pointer to a DOT11_NETWORK_LIST structure that contains the list of permitted or denied networks.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to get the filter list. When called with *wlanFilterListType* set to **wlan_filter_list_type_user_permit**, WlanGetFilterList retrieves the discretionary access control list (DACL) stored with the **wlan_secure_permit_list** object. When called with *wlanFilterListType* set to **wlan_filter_list_type_user_deny**, **WlanGetFilterList** retrieves the DACL stored with the **wlan_secure_deny_list** object. In either of these cases, if the DACL does not contain an access control entry (ACE) that grants WLAN_READ_ACCESS permission to the access token of the calling thread, then **WlanGetFilterList** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *ppNetworkList* is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

User permission lists can be set by calling WlanSetFilterList.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

DOT11_NETWORK_LIST

WLAN_FILTER_LIST_TYPE

WlanGetInterfaceCapability function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanGetInterfaceCapability** function retrieves the capabilities of an interface.

Syntax

```
DWORD WlanGetInterfaceCapability(
  HANDLE                    hClientHandle,
  const GUID                *pInterfaceGuid,
  PVOID                     pReserved,
  PWLAN_INTERFACE_CAPABILITY *ppCapability
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of this interface.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`ppCapability`

A WLAN_INTERFACE_CAPABILITY structure that contains information about the capabilities of the specified interface.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pInterfaceGuid* is **NULL**, *pReserved* is not **NULL**, or *ppCapability* is **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.

RPC_STATUS	Various error codes.

Remarks

The caller is responsible for calling the WlanFreeMemory function to free the memory allocated to *ppCapability*.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

WlanGetNetworkBssList function (wlanapi.h)

7/1/2021 • 5 minutes to read • Edit Online

The **WlanGetNetworkBssList** function retrieves a list of the basic service set (BSS) entries of the wireless network or networks on a given wireless LAN interface.

Syntax

```
DWORD WlanGetNetworkBssList(
  HANDLE            hClientHandle,
  const GUID        *pInterfaceGuid,
  const PDOT11_SSID pDot11Ssid,
  DOT11_BSS_TYPE    dot11BssType,
  BOOL              bSecurityEnabled,
  PVOID             pReserved,
  PWLAN_BSS_LIST    *ppWlanBssList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

A pointer to the GUID of the wireless LAN interface to be queried.

The GUID of each wireless LAN interface enabled on a local computer can be determined using the WlanEnumInterfaces function.

`pDot11Ssid`

A pointer to a DOT11_SSID structure that specifies the SSID of the network from which the BSS list is requested. This parameter is optional. When set to **NULL**, the returned list contains all of available BSS entries on a wireless LAN interface.

If a pointer to a DOT11_SSID structure is specified, the SSID length specified in the **uSSIDLength** member of **DOT11_SSID** structure must be less than or equal to **DOT11_SSID_MAX_LENGTH** defined in the *Wlantypes.h* header file. In addition, the *dot11BssType* parameter must be set to either **dot11_BSS_type_infrastructure** or **dot11_BSS_type_independent** and the *bSecurityEnabled* parameter must be specified.

`dot11BssType`

The BSS type of the network. This parameter is ignored if the SSID of the network for the BSS list is unspecified (the *pDot11Ssid* parameter is **NULL**).

This parameter can be one of the following values defined in the DOT11_BSS_TYPE enumeration defined in the *Wlantypes.h* header file.

VALUE	MEANING

dot11_BSS_type_infrastructure	An infrastructure BSS network.
dot11_BSS_type_independent	An independent BSS (IBSS) network (an ad hoc network).
dot11_BSS_type_any	Any BSS network.

`bSecurityEnabled`

A value that indicates whether security is enabled on the network. This parameter is only valid when the SSID of the network for the BSS list is specified (the *pDot11Ssid* parameter is not **NULL**).

`pReserved`

Reserved for future use. This parameter must be set to **NULL**.

`ppWlanBssList`

A pointer to storage for a pointer to receive the returned list of of BSS entries in a WLAN_BSS_LIST structure.

The buffer for the WLAN_BSS_LIST returned is allocated by the **WlanGetNetworkBssList** function if the call succeeds.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if the *hClientHandle*, *pInterfaceGuid*, or *ppWlanBssList* parameter is **NULL**. This error is returned if the *pReserved* is not **NULL**. This error is also returned if the *hClientHandle*, the SSID specified in the *pDot11Ssid* parameter, or the BSS type specified in the *dot11BssType* parameter is not valid.
ERROR_NDIS_DOT11_POWER_STATE_INVALID	The radio associated with the interface is turned off. The BSS list is not available when the radio is off.
ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to process this request and allocate memory for the query results.
ERROR_NOT_FOUND	The element was not found. This error is returned if the GUID of the interface to be queried that was specified in the *pInterfaceGuid* parameter could not be found.

ERROR_NOT_SUPPORTED	The request is not supported. This error is returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client. This error is also returned if the WLAN AutoConfig service is disabled.
ERROR_SERVICE_NOT_ACTIVE	The WLAN AutoConfig service has not been started.
RPC_STATUS	Various error codes.

Remarks

The **WlanGetNetworkBssList** function retrieves the basic service set list for each wireless network or networks accessible on a given interface. The list of information returned for each wireless network also contains a list of information elements returned by each access point for an infrastructure BSS network or a network peer for an independent BSS network (ad hoc network). The information is returned as a pointer to an WLAN_BSS_LIST structure in the *ppWlanBssList* parameter. The **WLAN_BSS_LIST** structure contains an item count followed by an array of WLAN_BSS_ENTRY structure entries.

Since the information returned by the **WlanGetNetworkBssList** function is sent by an access point for an infrastructure BSS network or by a network peer for an independent BSS network (ad hoc network), the information returned should not be trusted. The **ulIeOffset** and **ulIeSize** members in the WLAN_BSS_ENTRY structure should be used to determine the size of the information element data blob in the **WLAN_BSS_ENTRY** structure, not the data in the information element data blob itself. The **WlanGetNetworkBssList** function does not validate that any information returned in the information element data blob pointed to by the **ulIeOffset** member is a valid information element as defined by the IEEE 802.11 standards for wireless LANs.

If the *pDot11Ssid* parameter is specified (not **NULL**), then the *dot11BssType* parameter specified must be set to either **dot11_BSS_type_infrastructure** for an infrastructure BSS network or **dot11_BSS_type_independent** for an independent BSS network (ad hoc network). If the *dot11BssType* parameter is set to **dot11_BSS_type_any**, then the **WlanGetNetworkBssList** function returns ERROR_SUCCESS but no BSS entries will be returned.

To return a list of all the infrastructure BSS networks and independent BSS networks (ad hoc networks) on a wireless LAN interface, set the *pDot11Ssid* parameter to **NULL**. When the wireless LAN interface is also operating as a Wireless Hosted Network , the BSS list will contain an entry for the BSS created for the Wireless Hosted Network.

The **WlanGetNetworkBssList** function returns ERROR_SUCCESS when an empty BSS list is returned by the WLAN AutoConfig Service. An application that calls the **WlanGetNetworkBssList** function must check that the **dwNumberOfItems** member of the WLAN_BSS_LIST pointed to by the *ppWlanBssList* parameter is not zero before accessing the **wlanBssEntries[0]** member in **WLAN_BSS_LIST** structure.

The **WlanGetNetworkBssList** function allocates memory for the basic service set list that is returned in a buffer pointed to by the *ppWlanBssList* parameter when the function succeeds. The memory used for the buffer pointed to by *ppWlanBssList* parameter should be released by calling the WlanFreeMemory function after the buffer is no longer needed.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_AVAILABLE_NETWORK

WLAN_AVAILABLE_NETWORK_LIST

WLAN_BSS_ENTRY

WLAN_BSS_LIST

WlanEnumInterfaces

WlanFreeMemory

WlanGetAvailableNetworkList

WlanScan

WlanGetProfile function (wlanapi.h)

7/1/2021 • 9 minutes to read • Edit Online

The **WlanGetProfile** function retrieves all information about a specified wireless profile.

Syntax

```
DWORD WlanGetProfile(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  PVOID      pReserved,
  LPWSTR     *pstrProfileXml,
  DWORD      *pdwFlags,
  DWORD      *pdwGrantedAccess
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the wireless interface.

A list of the GUIDs for wireless interfaces on the local computer can be retrieved using the WlanEnumInterfaces function.

`strProfileName`

The name of the profile. Profile names are case-sensitive. This string must be NULL-terminated. The maximum length of the profile name is 255 characters. This means that the maximum length of this string, including the NULL terminator, is 256 characters.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The name of the profile is derived automatically from the SSID of the network. For infrastructure network profiles, the name of the profile is the SSID of the network. For ad hoc network profiles, the name of the profile is the SSID of the ad hoc network followed by `-adhoc`.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pstrProfileXml`

A string that is the XML representation of the queried profile. There is no predefined maximum string length.

`pdwFlags`

On input, a pointer to the address location used to provide additional information about the request. If this parameter is **NULL** on input, then no information on profile flags will be returned. On output, a pointer to the address location used to receive profile flags.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Per-user profiles are not supported. Set this parameter to **NULL**.

The *pdwFlags* parameter can point to an address location that contains the following values:

VALUE	MEANING
WLAN_PROFILE_GET_PLAINTEXT_KEY	On input, this flag indicates that the caller wants to retrieve the plain text key from a wireless profile. If the calling thread has the required permissions, the **WlanGetProfile** function returns the plain text key in the keyMaterial element of the profile returned in the buffer pointed to by the *pstrProfileXml* parameter. For the **WlanGetProfile** call to return the plain text key, the **wlan_secure_get_plaintext_key** permissions from the WLAN_SECURABLE_OBJECT enumerated type must be set on the calling thread. The DACL must also contain an ACE that grants **WLAN_READ_ACCESS** permission to the access token of the calling thread. By default, the permissions for retrieving the plain text key is allowed only to the members of the Administrators group on a local machine. If the calling thread lacks the required permissions, the **WlanGetProfile** function returns the encrypted key in the keyMaterial element of the profile returned in the buffer pointed to by the *pstrProfileXml* parameter. No error is returned if the calling thread lacks the required permissions. **Windows 7:** This flag passed on input is an extension to native wireless APIs added on Windows 7 and later. The *pdwFlags* parameter is an __inout_opt parameter on Windows 7 and later.
WLAN_PROFILE_GROUP_POLICY	On output when the **WlanGetProfile** call is successful, this flag indicates that this profile was created by group policy. A group policy profile is read-only. Neither the content nor the preference order of the profile can be changed.
WLAN_PROFILE_USER	On output when the **WlanGetProfile** call is successful, this flag indicates that the profile is a user profile for the specific user in whose context the calling thread resides. If not set, this profile is an all-user profile.

`pdwGrantedAccess`

The access mask of the all-user profile.

VALUE	MEANING
WLAN_READ_ACCESS	The user can view the contents of the profile.
WLAN_EXECUTE_ACCESS	The user has read access, and the user can also connect to and disconnect from a network using the profile. If a user has WLAN_EXECUTE_ACCESS, then the user also has WLAN_READ_ACCESS.

WLAN_WRITE_ACCESS	The user has execute access and the user can also modify the content of the profile or delete the profile. If a user has WLAN_WRITE_ACCESS, then the user also has WLAN_EXECUTE_ACCESS and WLAN_READ_ACCESS.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions. This error is returned if the *pstrProfileXml* parameter specifies an all-user profile, but the caller does not have read access on the profile.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pInterfaceGuid* is **NULL**. • *pstrProfileXml* is **NULL**. • *pReserved* is not **NULL**.
ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command. This error is returned if the system was unable to allocate memory for the profile.
ERROR_NOT_FOUND	The profile specified by *strProfileName* was not found.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

If the **WlanGetProfile** function succeeds, the wireless profile is returned in the buffer pointed to by the *pstrProfileXml* parameter. The buffer contains a string that is the XML representation of the queried profile. For a description of the XML representation of the wireless profile, see WLAN_profile Schema.

The caller is responsible for calling the WlanFreeMemory function to free the memory allocated for the buffer pointer to by the *pstrProfileXml* parameter when the buffer is no longer needed.

If *pstrProfileXml* specifies an all-user profile, the **WlanGetProfile** caller must have read access on the profile. Otherwise, the **WlanGetProfile** call will fail with a return value of **ERROR_ACCESS_DENIED**. The permissions on an all-user profile are established when the profile is created or saved using WlanSetProfile or WlanSaveTemporaryProfile.

Windows 7:

The keyMaterial element returned in the profile schema pointed to by the *pstrProfileXml* may be requested as plaintext if the **WlanGetProfile** function is called with the **WLAN_PROFILE_GET_PLAINTEXT_KEY** flag set in the value pointed to by the *pdwFlags* parameter on input.

For a WEP key, both 5 ASCII characters or 10 hexadecimal characters can be used to set the plaintext key when the profile is created or updated. However, a WEP profile will be saved with 10 hexadecimal characters in the key no matter what the original input was used to create the profile. So in the profile returned by the **WlanGetProfile** function, the plaintext WEP key is always returned as 10 hexadecimal characters.

For the **WlanGetProfile** call to return the plain text key, the **wlan_secure_get_plaintext_key** permissions from the WLAN_SECURABLE_OBJECT enumerated type must be set on the calling thread. The DACL must also contain an ACE that grants **WLAN_READ_ACCESS** permission to the access token of the calling thread. By default, the permissions for retrieving the plain text key is allowed only to the members of the Administrators group on a local machine.

If the calling thread lacks the required permissions, the **WlanGetProfile** function returns the encrypted key in the keyMaterial element of the profile returned in the buffer pointed to by the *pstrProfileXml* parameter. No error is returned if the calling thread lacks the required permissions.

By default, the keyMaterial element returned in the profile pointed to by the *pstrProfileXml* is encrypted. If your process runs in the context of the LocalSystem account on the same computer, then you can unencrypt key material by calling the CryptUnprotectData function.

Windows Server 2008 and Windows Vista: The keyMaterial element returned in the profile schema pointed to by the *pstrProfileXml* is always encrypted. If your process runs in the context of the LocalSystem account, then you can unencrypt key material by calling the CryptUnprotectData function.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The key material is never encrypted.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, retrieves information for a specific wireless profile on each wireless LAN interface, and prints the values retrieved. The string that is the XML representation of the queried profile is also printed.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")


int _cdecl wmain(int argc, WCHAR **argv)
{
```

```
// Declare and initialize variables.

HANDLE hClient = NULL;
DWORD dwMaxClient = 2;      //
DWORD dwCurVersion = 0;
DWORD dwResult = 0;
DWORD dwRetVal = 0;
int iRet = 0;

WCHAR GuidString[39] = {0};

unsigned int i;

/* variables used for WlanEnumInterfaces  */

PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
PWLAN_INTERFACE_INFO pIfInfo = NULL;

LPCWSTR pProfileName = NULL;
LPWSTR pProfileXml = NULL;
DWORD dwFlags = 0;
DWORD dwGrantedAccess = 0;

    // Validate the parameters
if (argc < 2) {
    wprintf(L"usage: %s <profile>\n", argv[0]);
    wprintf(L"   Gets a wireless profile\n");
    wprintf(L"   Example\n");
    wprintf(L"       %s \"Default Wireless\"\n", argv[0]);
    exit(1);
}

pProfileName = argv[1];

wprintf(L"Information for profile: %ws\n\n", pProfileName);

dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
}

dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
} else {
    wprintf(L"WLAN_INTERFACE_INFO_LIST for this system\n");

    wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
    wprintf(L"Current Index: %lu\n\n", pIfList->dwIndex);
    for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
        pIfInfo = (WLAN_INTERFACE_INFO *) &pIfList->InterfaceInfo[i];
        wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
        iRet = StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            sizeof(GuidString)/sizeof(*GuidString));
        // For c rather than C++ source code, the above line needs to be
        // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
        //     sizeof(GuidString)/sizeof(*GuidString));
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else {
            wprintf(L"  InterfaceGUID[%d]: %ws\n",i, GuidString);
        }
        wprintf(L"  Interface Description[%d]: %ws", i,
            pIfInfo->strInterfaceDescription);
        wprintf(L"\n");
```

```
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n\n");

            dwResult = WlanGetProfile(hClient,
                                      &pIfInfo->InterfaceGuid,
                                      pProfileName,
                                      NULL,
                                      &pProfileXml,
                                      &dwFlags,
                                      &dwGrantedAccess);

            if (dwResult != ERROR_SUCCESS) {
                wprintf(L"WlanGetProfile failed with error: %u\n",
                        dwResult);
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"  Profile Name:  %ws\n", pProfileName);

                wprintf(L"  Profile XML string:\n");
                wprintf(L"%ws\n\n", pProfileXml);

                wprintf(L"  dwFlags:\t    0x%x", dwFlags);
//                    if (dwFlags & WLAN_PROFILE_GET_PLAINTEXT_KEY)
//                        wprintf(L"   Get Plain Text Key");
                    if (dwFlags & WLAN_PROFILE_GROUP_POLICY)
                        wprintf(L"  Group Policy");
                    if (dwFlags & WLAN_PROFILE_USER)
                        wprintf(L"  Per User Profile");
                    wprintf(L"\n");

                wprintf(L"  dwGrantedAccess:  0x%x", dwGrantedAccess);
                if (dwGrantedAccess & WLAN_READ_ACCESS)
                    wprintf(L"  Read access");
                if (dwGrantedAccess & WLAN_EXECUTE_ACCESS)
                    wprintf(L"  Execute access");
                if (dwGrantedAccess & WLAN_WRITE_ACCESS)
                    wprintf(L"  Write access");
                wprintf(L"\n");
```

```
                wprintf(L"\n");
            }
        }

    }
    if (pProfileXml != NULL) {
        WlanFreeMemory(pProfileXml);
        pProfileXml = NULL;
    }

    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_PROFILE_INFO

WLAN_PROFILE_INFO_LIST

WLAN_SECURABLE_OBJECT

WLAN_profile Schema

WlanDeleteProfile

WlanEnumInterfaces

WlanFreeMemory

WlanGetProfileCustomUserData

WlanGetProfileList

WlanOpenHandle

WlanRenameProfile

WlanSaveTemporaryProfile

WlanSetProfile

WlanSetProfileCustomUserData

WlanSetProfileEapUserData

WlanSetProfileEapXmlUserData

WlanSetProfileList

WlanSetProfilePosition

WlanGetProfileCustomUserData function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanGetProfileCustomUserData** function gets the custom user data associated with a wireless profile.

Syntax

```
DWORD WlanGetProfileCustomUserData(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  PVOID      pReserved,
  DWORD      *pdwDataSize,
  PBYTE      *ppData
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

A pointer to the GUID of the wireless LAN interface.

`strProfileName`

The name of the profile with which the custom user data is associated. Profile names are case-sensitive. This string must be NULL-terminated.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwDataSize`

The size, in bytes, of the user data buffer pointed to by the *ppData*parameter.

`ppData`

A pointer to the user data.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_FILE_NOT_FOUND	The system cannot find the file specified. This error is returned if no user custom data exists for the profile specified.

ERROR_INVALID_PARAMETER	The *hClientHandle* parameter is **NULL** or not valid, the *pInterfaceGuid* parameter is **NULL**, the *strProfileName* parameter is **NULL**, the *pReserved* parameter is not **NULL**, the *pdwDataSize* parameter is 0, or the *ppData* parameter is **NULL**.
ERROR_FILE_NOT_FOUND	The system cannot find the file specified. This error is returned if no custom user data exists for the profile specified.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

For every wireless WLAN profile used by the Native Wifi AutoConfig service, Windows maintains the concept of custom user data. This custom user data is initially non-existent, but can be set by calling the WlanSetProfileCustomUserData function. The custom user data gets reset to empty any time the profile is modified by calling the WlanSetProfile function.

Once custom user data has been set, this data can be accessed using the **WlanGetProfileCustomUserData** function.

The caller is responsible for freeing the memory allocated for the buffer pointed to by the *ppData* parameter using the WlanFreeMemory function.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, and then tries to retrieve any custom user data information for a specific wireless profile on each wireless LAN interface. The size of the user custom data is printed.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
```

```
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")


int _cdecl wmain(int argc, WCHAR **argv)
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;      //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    DWORD dwRetVal = 0;
    int iRet = 0;

    WCHAR GuidString[39] = {0};

    unsigned int i;

    /* variables used for WlanEnumInterfaces  */

    PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
    PWLAN_INTERFACE_INFO pIfInfo = NULL;

    LPCWSTR pProfileName = NULL;

    PBYTE pProfileData = NULL;
    DWORD dwDataSize = 0;

        // Validate the parameters
    if (argc < 2) {
        wprintf(L"usage: %s <profile>\n", argv[0]);
        wprintf(L"   Gets a wireless profile\n");
        wprintf(L"   Example\n");
        wprintf(L"       %s \"Default Wireless\"\n", argv[0]);
        exit(1);
    }

    pProfileName = argv[1];

    wprintf(L"Custom user data information for profile: %ws\n\n", pProfileName);

    dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    }

    dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    } else {
        wprintf(L"WLAN_INTERFACE_INFO_LIST for this system\n");

        wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
        wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
        for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
            pIfInfo = (WLAN_INTERFACE_INFO *) &pIfList->InterfaceInfo[i];
            wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
            iRet = StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
                sizeof(GuidString)/sizeof(*GuidString));
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            //     sizeof(GuidString)/sizeof(*GuidString));
            if (iRet == 0)
```

```
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"  InterfaceGUID[%d]: %ws\n",i, GuidString);
            }
            wprintf(L"  Interface Description[%d]: %ws", i,
                pIfInfo->strInterfaceDescription);
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            dwResult = WlanGetProfileCustomUserData(hClient,
                                        &pIfInfo->InterfaceGuid,
                                        pProfileName,
                                        NULL,
                                        &dwDataSize,
                                        &pProfileData);

            if (dwResult != ERROR_SUCCESS) {
                wprintf(L"WlanGetProfileCustomData failed with error: %u\n",
                        dwResult);
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"Profile Name:  %ws\n", pProfileName);

                wprintf(L"  dwDataSize:\t    0x%x\n", dwDataSize);
                wprintf(L"  Profile Custom Data:\n");
//                wprintf(L"%ws\n\n", pProfileXml);

                wprintf(L"\n");

                wprintf(L"\n");
            }
        }

    }
    if (pProfileData != NULL) {
        WlanFreeMemory(pProfileData);
        pProfileData = NULL;
    }
```

```
    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WlanGetProfile

WlanGetProfileList

WlanSetProfile

WlanSetProfileCustomUserData

WlanGetProfileList function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanGetProfileList** function retrieves the list of profiles in preference order.

Syntax

```
DWORD WlanGetProfileList(
  HANDLE                  hClientHandle,
  const GUID              *pInterfaceGuid,
  PVOID                   pReserved,
  PWLAN_PROFILE_INFO_LIST *ppProfileList
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the wireless interface.

A list of the GUIDs for wireless interfaces on the local computer can be retrieved using the WlanEnumInterfaces function.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`ppProfileList`

A PWLAN_PROFILE_INFO_LIST structure that contains the list of profile information.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pInterfaceGuid* is **NULL**. • *ppProfileList* is **NULL**. • *pReserved* is not **NULL**.

ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to process this request and allocate memory for the query results.
RPC_STATUS	Various error codes.

Remarks

The **WlanGetProfileList** function returns only the basic information on the wireless profiles on a wireless interface. The list of wireless profiles on a wireless interface are retrieved in the preference order. The WlanSetProfilePosition can be used to change the preference order for the wireless profiles on a wireless interface.

More detailed information for a wireless profile on a wireless interface can be retrieved by using the WlanGetProfile function. The WlanGetProfileCustomUserData function can be used to retrieve custom user data for a wireless profile on a wireless interface. A list of the wireless interfaces and associated GUIDs on the local computer can be retrieved using the WlanEnumInterfaces function.

The **WlanGetProfileList** function allocates memory for the list of profiles returned in the buffer pointed to by the *ppProfileList* parameter. The caller is responsible for freeing this memory using the WlanFreeMemory function when this buffer is no longer needed.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Guest profiles, profiles with Wireless Provisioning Service (WPS) authentication, and profiles with Wi-Fi Protected Access-None (WPA-None) authentication are not supported. These types of profiles are not returned by **WlanGetProfileList**, even if a profile of this type appears on the preferred profile list.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, retrieves the list of profiles on each wireless LAN interface, and prints values from the retrieved WLAN_PROFILE_INFO_LIST that contains the WLAN_PROFILE_INFO entries.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <objbase.h>
#include <wtypes.h>

#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")


int wmain()
{

    // Declare and initialize variables.
```

```
HANDLE hClient = NULL;
DWORD dwMaxClient = 2;      //
DWORD dwCurVersion = 0;
DWORD dwResult = 0;
DWORD dwRetVal = 0;
int iRet = 0;

WCHAR GuidString[39] = {0};

unsigned int i, j;

/* variables used for WlanEnumInterfaces  */

PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
PWLAN_INTERFACE_INFO pIfInfo = NULL;

PWLAN_PROFILE_INFO_LIST pProfileList = NULL;
PWLAN_PROFILE_INFO pProfile = NULL;

dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
}

dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
if (dwResult != ERROR_SUCCESS) {
    wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
    return 1;
    // You can use FormatMessage here to find out why the function failed
} else {
    wprintf(L"WLAN_INTERFACE_INFO_LIST for this system\n");

    wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
    wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
    for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
        pIfInfo = (WLAN_INTERFACE_INFO *) &pIfList->InterfaceInfo[i];
        wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
        iRet = StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            sizeof(GuidString)/sizeof(*GuidString));
        // For c rather than C++ source code, the above line needs to be
        // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
        //     sizeof(GuidString)/sizeof(*GuidString));
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else {
            wprintf(L"  Interface GUID[%d]: %ws\n",i, GuidString);
        }
        wprintf(L"  Interface Description[%d]: %ws", i,
            pIfInfo->strInterfaceDescription);
        wprintf(L"\n");
        wprintf(L"  Interface State[%d]:\t ", i);
        switch (pIfInfo->isState) {
        case wlan_interface_state_not_ready:
            wprintf(L"Not ready\n");
            break;
        case wlan_interface_state_connected:
            wprintf(L"Connected\n");
            break;
        case wlan_interface_state_ad_hoc_network_formed:
            wprintf(L"First node in a ad hoc network\n");
            break;
        case wlan_interface_state_disconnecting:
            wprintf(L"Disconnecting\n");
            break;
        case wlan_interface_state_disconnected:
            wprintf(L"Not connected\n");
```

```
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            dwResult = WlanGetProfileList(hClient,
                                          &pIfInfo->InterfaceGuid,
                                          NULL,
                                          &pProfileList);

            if (dwResult != ERROR_SUCCESS) {
                wprintf(L"WlanGetProfileList failed with error: %u\n",
                        dwResult);
                dwRetVal = 1;
                // You can use FormatMessage to find out why the function failed
            } else {
                wprintf(L"WLAN_PROFILE_INFO_LIST for this interface\n");

                wprintf(L"  Num Entries: %lu\n\n", pProfileList->dwNumberOfItems);

                for (j = 0; j < pProfileList->dwNumberOfItems; j++) {
                    pProfile =
                        (WLAN_PROFILE_INFO *) & pProfileList->ProfileInfo[j];

                    wprintf(L"  Profile Name[%u]:  %ws\n", j, pProfile->strProfileName);

                    wprintf(L"  Flags[%u]:\t    0x%x", j, pProfile->dwFlags);
                    if (pProfile->dwFlags & WLAN_PROFILE_GROUP_POLICY)
                        wprintf(L"   Group Policy");
                    if (pProfile->dwFlags & WLAN_PROFILE_USER)
                        wprintf(L"   Per User Profile");
                    wprintf(L"\n");

                    wprintf(L"\n");
                }
            }
        }

    }
    if (pProfileList != NULL) {
        WlanFreeMemory(pProfileList);
        pProfileList = NULL;
    }

    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_PROFILE_INFO

WLAN_PROFILE_INFO_LIST

WlanDeleteProfile

WlanFreeMemory

WlanGetProfile

WlanGetProfileCustomUserData

WlanOpenHandle

WlanRenameProfile

WlanSaveTemporaryProfile

WlanSetProfile

WlanSetProfileCustomUserData

WlanSetProfileEapUserData

WlanSetProfileEapXmlUserData

WlanSetProfileList

WlanSetProfilePosition

WlanGetSecuritySettings function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanGetSecuritySettings** function gets the security settings associated with a configurable object.

Syntax

```
DWORD WlanGetSecuritySettings(
  HANDLE                  hClientHandle,
  WLAN_SECURABLE_OBJECT   SecurableObject,
  PWLAN_OPCODE_VALUE_TYPE pValueType,
  LPWSTR                  *pstrCurrentSDDL,
  PDWORD                  pdwGrantedAccess
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`SecurableObject`

A WLAN_SECURABLE_OBJECT value that specifies the object to which the security settings apply.

`pValueType`

A pointer to a WLAN_OPCODE_VALUE_TYPE value that specifies the source of the security settings.

VALUE	MEANING
wlan_opcode_value_type_set_by_group_policy	The security settings were set by group policy.
wlan_opcode_value_type_set_by_user	The security settings were set by the user. A user can set security settings by calling WlanSetSecuritySettings.

`pstrCurrentSDDL`

On input, this parameter must be **NULL**.

On output, this parameter receives a pointer to the security descriptor string that specifies the security settings for the object if the function call succeeds. For more information about this string, see WlanSetSecuritySettings function.

`pdwGrantedAccess`

The access mask of the object.

VALUE	MEANING
WLAN_READ_ACCESS	The caller can view the object's permissions.

WLAN_EXECUTE_ACCESS	The caller can read from and execute the object. WLAN_EXECUTE_ACCESS has the same value as the bitwise OR combination WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS.
WLAN_WRITE_ACCESS	The caller can read from, execute, and write to the object. WLAN_WRITE_ACCESS has the same value as the bitwise OR combination WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS | WLAN_WRITE_ACCESS.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pstrCurrentSDDL* is **NULL**. • *pdwGrantedAccess* is **NULL**. • *SecurableObject* is set to a value greater than or equal to **WLAN_SECURABLE_OBJECT_COUNT** (12).
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.

Remarks

The caller is responsible for freeing the memory allocated to the security descriptor string pointed to by the *pstrCurrentSDDL* parameter if the function succeeds. When no longer needed, the memory for the security descriptor string should be freed by calling WlanFreeMemory function and passing in the *pstrCurrentSDDL* parameter.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

Native Wifi API Permissions

WlanFreeMemory

WlanSetSecuritySettings

WlanGetSupportedDeviceServices function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Retrieves a list of the supported device services on a given wireless LAN interface.

Syntax

```
DWORD WlanGetSupportedDeviceServices(
  HANDLE                        hClientHandle,
  const GUID                    *pInterfaceGuid,
  PWLAN_DEVICE_SERVICE_GUID_LIST *ppDevSvcGuidList
);
```

Parameters

`hClientHandle`

Type: HANDLE

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

Type: CONST GUID*

A pointer to the **GUID** of the wireless LAN interface to be queried. You can determine the **GUID** of each wireless LAN interface enabled on a local computer by using the WlanEnumInterfaces function.

`ppDevSvcGuidList`

Type: PWLAN_DEVICE_SERVICE_GUID_LIST*

A pointer to storage for a pointer to receive the returned list of device service **GUID**s in a WLAN_DEVICE_SERVICE_GUID_LIST structure. If the call succeeds, then the buffer for the **WLAN_DEVICE_SERVICE_GUID_LIST** returned is allocated by the **WlanGetSupportedDeviceServices** function.

Return value

Type: HRESULT

If the function succeeds, the return value is **ERROR_SUCCESS**. If the function fails with **ERROR_ACCESS_DENIED**, then the caller doesn't have sufficient permissions to perform this operation. The caller needs to either have admin privilege, or needs to be a UMDF driver.

Remarks

If the call succeeds, then the **WlanGetSupportedDeviceServices** function allocates memory for the device services **GUID** list that's returned in a buffer pointed to by the *ppDevSvcGuidList* parameter. When you no longer need the buffer pointed to by *ppDevSvcGuidList*, you should release the memory used for it by calling the WlanFreeMemory function.

Requirements

Header	wlanapi.h

WlanHostedNetworkForceStart function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanHostedNetworkForceStart** function transitions the wireless Hosted Network to the **wlan_hosted_network_active state** without associating the request with the application's calling handle.

Syntax

```
DWORD WlanHostedNetworkForceStart(
  HANDLE                     hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                      pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason if the call to the **WlanHostedNetworkForceStart** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.

ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This error is returned if the wireless Hosted Network is disabled by group policy on a domain.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkForceStart** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkForceStart** function to force the start of the wireless Hosted Network by transitioning the wireless Hosted Network to the **wlan_hosted_network_active state** without associating the request with the application's calling handle. A successful call to the **WlanHostedNetworkForceStart** function should eventually be matched by a call to WlanHostedNetworkForceStop function. Any Hosted Network state change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

The cost of calling the **WlanHostedNetworkForceStart** function over calling WlanHostedNetworkStartUsing is the associated privilege required. An application might call the **WlanHostedNetworkForceStart** function after ensuring that an elevated system user accepts the increased power requirements involved in running the wireless Hosted Network for extended durations.

The **WlanHostedNetworkForceStart** function could fail if Hosted Network state is **wlan_hosted_network_unavailable** or the caller does not have sufficient privileges.

This function to force the start of the Hosted Network can only be called if the user has the appropriate associated privilege. Permissions are stored in a discretionary access control list (DACL) associated with a WLAN_SECURABLE_OBJECT. To call the **WlanHostedNetworkForceStart**, the client access token of the caller must have elevated privileges exposed by the following enumeration in **WLAN_SECURABLE_OBJECT**:

- **wlan_secure_hosted_network_elevated_access**

The ability to enable the wireless Hosted Network may also be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WLAN_SECURABLE_OBJECT

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkForceStop

WlanHostedNetworkQueryStatus

WlanHostedNetworkStartUsing

WlanHostedNetworkStopUsing

WlanOpenHandle

WlanHostedNetworkForceStop function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanHostedNetworkForceStop** function transitions the wireless Hosted Network to the **wlan_hosted_network_idle** without associating the request with the application's calling handle.

Syntax

```
DWORD WlanHostedNetworkForceStop(
  HANDLE                      hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the **WlanHostedNetworkForceStop** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation.

ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkForceStop** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkForceStop** function to force the stop the Hosted Network and transition the wireless Hosted Network to the **wlan_hosted_network_idle** without associating the request with the application's calling handle. A client typically calls the **WlanHostedNetworkForceStop** function to match an earlier successful call to the WlanHostedNetworkForceStart function.

The **WlanHostedNetworkForceStop** function could fail if Hosted Network state is not **wlan_hosted_network_active**.

Any Hosted Network state change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

An application might call the **WlanHostedNetworkForceStop** function to stop the Hosted Network after a previous call to the WlanHostedNetworkForceStart by an elevated system user that accepted the increased power requirements involved in running the wireless Hosted Network for extended durations.

Any user can call the **WlanHostedNetworkForceStop** function to force the stop of the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib

DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WLAN_SECURABLE_OBJECT

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkForceStart

WlanHostedNetworkQueryStatus

WlanHostedNetworkStartUsing

WlanHostedNetworkStopUsing

WlanOpenHandle

WlanHostedNetworkInitSettings function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanHostedNetworkInitSettings** function configures and persists to storage the network connection settings (SSID and maximum number of peers, for example) on the wireless Hosted Network if these settings are not already configured.

Syntax

```
DWORD WlanHostedNetworkInitSettings(
  HANDLE                      hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason if the call to the **WlanHostedNetworkInitSettings** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation.

ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkInitSettings** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkInitSettings** function to configure and persist to storage the network connection settings (SSID and maximum number of peers, for example) on the wireless Hosted Network, if the connections settings are not already configured. If the network settings on the wireless Hosted Network settings are already configured (the WlanHostedNetworkQueryProperty function does not return **ERROR_BAD_CONFIGURATION** for the station profile or connection settings), then this function call returns **ERROR_SUCCESS** without changing the configuration of the network connection settings.

A client application should always call the **WlanHostedNetworkInitSettings** function before using other Hosted Network features on the local computer. This function initializes settings that are required when the wireless Hosted Network is used for the first time on a local computer. The **WlanHostedNetworkInitSettings** function does not change any configuration if the configuration has already been persisted. So it is safe to call the **WlanHostedNetworkInitSettings** function if the configuration has already been persisted. It is recommended that applications that use Hosted Network call the **WlanHostedNetworkInitSettings** function before using other Hosted Network functions.

The **WlanHostedNetworkInitSettings** function computes a random and readable SSID from the host name and computes a random primary key. This function also uses sets a value for the maximum number of peers allowed that defaults to 100. If an application wants to use a different SSID or a different maximum number of peers, then the application should call the WlanHostedNetworkSetProperty function to specifically set these properties used by the wireless Hosted Network.

Any Hosted Network state change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

Any user can call the **WlanHostedNetworkInitSettings** function to configure and persist to storage network connection settings on the Hosted Network. If the wireless Hosted Network has already been configured, this function does nothing and returns **ERROR_SUCCESS**.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WLAN_SECURABLE_OBJECT

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkQueryProperty

WlanHostedNetworkQuerySecondaryKey

WlanHostedNetworkQueryStatus

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanHostedNetworkSetSecondaryKey

WlanOpenHandle

WlanHostedNetworkQueryProperty function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanHostedNetworkQueryProperty** function queries the current static properties of the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkQueryProperty(
  HANDLE                     hClientHandle,
  WLAN_HOSTED_NETWORK_OPCODE OpCode,
  PDWORD                     pdwDataSize,
  PVOID                      *ppvData,
  PWLAN_OPCODE_VALUE_TYPE    pWlanOpcodeValueType,
  PVOID                      pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`OpCode`

The identifier for property to be queried. This identifier can be any of the values in the WLAN_HOSTED_NETWORK_OPCODE enumeration defined in the *Wlanapi.h* header file.

`pdwDataSize`

A pointer to a value that specifies the size, in bytes, of the buffer returned in the *ppvData* parameter, if the call to the **WlanHostedNetworkQueryProperty** function succeeds.

`ppvData`

On input, this parameter must be **NULL**.

On output, this parameter receives a pointer to a buffer returned with the static property requested, if the call to the **WlanHostedNetworkQueryProperty** function succeeds. The data type associated with this buffer depends upon the value of *OpCode* parameter.

`pWlanOpcodeValueType`

A pointer to a value that receives the value type of the wireless Hosted Network property, if the call to the **WlanHostedNetworkQueryProperty** function succeeds. The returned value is an enumerated type in the WLAN_OPCODE_VALUE_TYPE enumeration defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_BAD_CONFIGURATION	The configuration data for the wireless Hosted Network is unconfigured. This error is returned if the application calls the WlanHostedNetworkQueryProperty function with the *OpCode* parameter set to **wlan_hosted_network_opcode_station_profile** or **wlan_hosted_network_opcode_connection_settings** before a SSID is configured in the wireless Hosted Network.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *OpCode* is not one of the enumerated values defined in the WLAN_HOSTED_NETWORK_OPCODE. • *pdwDataSize* is **NULL**. • *ppvData* is **NULL**. • *pWlanOpcodeValueType* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_OUTOFMEMORY	Not enough storage is available to complete this operation.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkQueryProperty** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkQueryProperty** function to query the current static properties of the wireless Hosted Network. This function does not change the state or properties of the wireless Hosted Network.

If the function succeeds, the *ppvData* parameter points to a buffer that contains the requested property. The size of this buffer is returned in a pointer returned in the *pwdDataSize*parameter. The WLAN_OPCODE_VALUE_TYPE is returned in a pointer returned in the *pWlanOpcodeValueType* parameter. The memory used for the buffer in the *ppvData* parameter that is returned should be released by calling the WlanFreeMemory function after the buffer is no longer needed.

The data type associated with the buffer pointed to by the *ppvData* parameter depends upon the value of *OpCode* parameter as follows:

OPCODE	DESCRIPTION
wlan_hosted_network_opcode_connection_settings	A pointer to a WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS structure is returned.
wlan_hosted_network_opcode_security_settings	A pointer to a WLAN_HOSTED_NETWORK_SECURITY_SETTINGS structure is returned.
wlan_hosted_network_opcode_station_profile	A **PWSTR** to contains an XML WLAN profile for connecting to the wireless Hosted Network is returned.
wlan_hosted_network_opcode_enable	A **PBOOL** that indicates if wireless Hosted Network is enabled is returned.

If the **WlanHostedNetworkQueryProperty** function is passed any of the following values in the *OpCode* parameter before a SSID is configured in the wireless Hosted Network, the function will fail with **ERROR_BAD_CONFIGURATION**:

- wlan_hosted_network_opcode_station_profile
- wlan_hosted_network_opcode_connection_settings

Any user can call the **WlanHostedNetworkQueryProperty** function to query the Hosted Network properties.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS

WLAN_HOSTED_NETWORK_OPCODE

WLAN_HOSTED_NETWORK_SECURITY_SETTINGS

WLAN_OPCODE_VALUE_TYPE

WlanEnumInterfaces

WlanFreeMemory

WlanHostedNetworkInitSettings

WlanHostedNetworkQuerySecondaryKey

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanHostedNetworkSetSecondaryKey

WlanOpenHandle

WlanHostedNetworkQuerySecondaryKey function (wlanapi.h)

7/1/2021 • 5 minutes to read • Edit Online

The **WlanHostedNetworkQuerySecondaryKey** function queries the secondary security key that is configured to be used by the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkQuerySecondaryKey(
  HANDLE                     hClientHandle,
  PDWORD                     pdwKeyLength,
  PUCHAR                     *ppucKeyData,
  PBOOL                      pbIsPassPhrase,
  PBOOL                      pbPersistent,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                      pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pdwKeyLength`

A pointer to a value that specifies number of valid data bytes in the key data array pointed to by the *ppucKeyData* parameter, if the call to the **WlanHostedNetworkQuerySecondaryKey** function succeeds.

This key length includes the terminating '\0' if the key is a passphrase.

`ppucKeyData`

A pointer to a value that receives a pointer to the buffer returned with the secondary security key data, if the call to the **WlanHostedNetworkQuerySecondaryKey** function succeeds.

`pbIsPassPhrase`

A pointer to a Boolean value that indicates if the key data array pointed to by the *ppucKeyData* parameter is in passphrase format.

If this parameter is **TRUE**, the key data array is in passphrase format. If this parameter is **FALSE**, the key data array is not in passphrase format.

`pbPersistent`

A pointer to a Boolean value that indicates if the key data array pointed to by the *ppucKeyData* parameter is to be stored and reused later or is for one-time use only.

If this parameter is **TRUE**, the key data array is to be stored and reused later. If this parameter is **FALSE**, the key data array is for one-time use only.

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the WlanHostedNetworkSetSecondaryKey function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pdwKeyLength* is **NULL**. • *ppucKeyData* is **NULL** or invalid. • *pbIsPassPhrase* is **NULL** or invalid. • *pbPersistent* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_OUTOFMEMORY	Not enough storage is available to complete this operation.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkQuerySecondaryKey** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkQuerySecondaryKey** function to query the secondary security key that will be used by the wireless Hosted Network. This function will return the key information including key data, key length, whether it is a passphrase, and whether it is persistent or for one-time use. This function does not change the state or properties of the wireless Hosted Network.

The secondary security key is a passphrase if the value pointed to by the *pbIsPassPhrase* parameter is **TRUE**. The secondary security key is a binary key if the value pointed to by the *pbIsPassPhrase* parameter is **FALSE**.

The secondary security key returned in the buffer pointed to by the *ppucKeyData* parameter is used with WPA2-Personal authentication and is in one of the following formats:

- A key passphrase that consists of an array of ASCII characters from 8 to 63 characters. The value pointed to by the *pdwKeyLength* parameter includes the terminating '\0' in the passphrase. The value pointed to by the *pdwKeyLength* parameter should be in the range of 9 to 64.
- A binary key that conists of 32 bytes of binary key data. The value pointed to by the *pdwKeyLength* parameter should be 32 for binary key.

The secondary security key is persistent if the value pointed to by the *pbPersistent* parameter is **TRUE**. When persistent, the secondary security key would be used immediately if the Hosted Network is already started, and also reused whenever Hosted Network is started in the future.

If secondary security key is not specified as persistent, it will be used immediately if the Hosted Network is already started, or only for the next time when the Hosted Network is started. After the Hosted Network is stopped, this secondary security key will never be used again and will be removed from the system.

If there is no secondary security key currently configured, the returned value pointed to by the *pdwKeyLength* parameter will be zero, and the value returned in the *ppucKeyData* parameter will be **NULL**. In such case, the value returned in the *pbIsPassPhrase* and *pbPersistent* parameters will be meaningless.

If the **WlanHostedNetworkQuerySecondaryKey** function succeeds, the memory used for the buffer in the *ppucKeyData* parameter that is returned should be freed after use by calling the WlanFreeMemory function.

Any user can call the **WlanHostedNetworkQuerySecondaryKey** function to query the secondary security key used in the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WlanCloseHandle

WlanEnumInterfaces

WlanFreeMemory

WlanHostedNetworkInitSettings

WlanHostedNetworkQueryProperty

WlanHostedNetworkQueryStatus

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanHostedNetworkSetSecondaryKey

WlanOpenHandle

WlanHostedNetworkQueryStatus function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanHostedNetworkQueryStatus** function queries the current status of the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkQueryStatus(
  HANDLE                     hClientHandle,
  PWLAN_HOSTED_NETWORK_STATUS *ppWlanHostedNetworkStatus,
  PVOID                      pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`ppWlanHostedNetworkStatus`

On input, this parameter must be **NULL**.

On output, this parameter receives a pointer to the current status of the wireless Hosted Network, if the call to the **WlanHostedNetworkQueryStatus** function succeeds. The current status is returned in a WLAN_HOSTED_NETWORK_STATUS structure.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • ppWlanHostedNetworkStatus is **NULL**. • *pvReserved* is not **NULL**.

ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkQueryStatus** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkQueryStatus** function to query the current status of the wireless Hosted Network. This function does not change the state of the wireless Hosted Network.

If the function succeeds, the *ppWlanHostedNetworkStatus* parameter points to a WLAN_HOSTED_NETWORK_STATUS structure with the current status. The memory used for the **WLAN_HOSTED_NETWORK_STATUS** structure that is returned should be freed after use by calling the WlanFreeMemory function.

Any user can call the **WlanHostedNetworkQueryStatus** function to query the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_STATUS

WlanEnumInterfaces

WlanFreeMemory

WlanHostedNetworkQueryProperty

WlanHostedNetworkQuerySecondaryKey

WlanOpenHandle

WlanHostedNetworkRefreshSecuritySettings function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanHostedNetworkRefreshSecuritySettings** function refreshes the configurable and auto-generated parts of the wireless Hosted Network security settings.

Syntax

```
DWORD WlanHostedNetworkRefreshSecuritySettings(
  HANDLE                      hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the **WlanHostedNetworkRefreshSecuritySettings** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation.

ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkRefreshSecuritySettings** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkRefreshSecuritySettings** function to force a refresh of the configurable and auto-generated parts of the security settings (the primary key) on the wireless Hosted Network.

An application might call the **WlanHostedNetworkRefreshSecuritySettings** function after ensuring that the user accepts the impact of updating the security settings. In order to succeed, this function must persist the new settings which would require that Hosted Network state be transitioned to wlan_hosted_network_idle if it was currently running (wlan_hosted_network_active).

> **Note** Any network clients (PCs or devices) on the wireless Hosted Network would have to be re-configured after calling the **WlanHostedNetworkRefreshSecuritySettings** function if their continued usage is a goal. An application would typically call this function in situations where the user feels that the security of the previous primary key used for security by the wireless Hosted Network has been violated. Note that the **WlanHostedNetworkRefreshSecuritySettings** function does not change or reset the secondary key.

Any Hosted Network state change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

Any user can call the **WlanHostedNetworkRefreshSecuritySettings** function to refresh the security settings on the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]

Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WLAN_SECURABLE_OBJECT

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkInitSettings

WlanHostedNetworkQueryProperty

WlanHostedNetworkQueryStatus

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanHostedNetworkSetSecondaryKey

WlanOpenHandle

WlanHostedNetworkSetProperty function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanHostedNetworkSetProperty** function sets static properties of the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkSetProperty(
  HANDLE                      hClientHandle,
  WLAN_HOSTED_NETWORK_OPCODE  OpCode,
  DWORD                       dwDataSize,
  PVOID                       pvData,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`OpCode`

The identifier for the property to be set. This identifier can only be the following values in the WLAN_HOSTED_NETWORK_OPCODE enumeration defined in the *Wlanapi.h* header file:

- **wlan_hosted_network_opcode_connection_settings**

The Hosted Network connection settings.

- **wlan_hosted_network_opcode_enable**

The Hosted Network enabled flag.

`dwDataSize`

A value that specifies the size, in bytes, of the buffer pointed to by the *pvData* parameter.

`pvData`

A pointer to a buffer with the static property to set. The data type associated with this buffer depends upon the value of *OpCode* parameter.

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the **WlanHostedNetworkSetProperty** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions. This error is also returned if the *OpCode* parameter was **wlan_hosted_network_opcode_enable** and the wireless Hosted Network is disabled by group policy on a domain.
ERROR_BAD_PROFILE	The network connection profile used by the wireless Hosted Network is corrupted.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *OpCode* is not one of the enumerated values defined in the WLAN_HOSTED_NETWORK_OPCODE. • *dwDataSize* is zero. • *pvData* is **NULL**. • *pvData* does not point to a well- formed static property. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_NOT_SUPPORTED	The request is not supported. This error is returned if the application calls the WlanHostedNetworkSetProperty function with the *OpCode* parameter set to **wlan_hosted_network_opcode_station_profile** or **wlan_hosted_network_opcode_security_settings**.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkSetProperty** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkSetProperty** function to set the current static properties of the wireless Hosted Network. Any Hosted Network property change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the

hClientHandle parameter) or if the process ends.

The data type associated with the buffer pointed to by the *pvData* parameter depends upon the value of *OpCode* parameter as follows:

OPCODE	DESCRIPTION
wlan_hosted_network_opcode_connection_settings	A pointer to a WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS structure is passed in the *pvData* parameter.
wlan_hosted_network_opcode_enable	A pointer to **BOOL** is passed in the *pvData* parameter.

If the **WlanHostedNetworkSetProperty** function is called with the *OpCode* parameter set to **wlan_hosted_network_opcode_enable**, the user must have the appropriate associated privilege. Permissions are stored in a discretionary access control list (DACL) associated with a WLAN_SECURABLE_OBJECT. To call the **WlanHostedNetworkSetProperty** function with the *OpCode* parameter of **wlan_hosted_network_opcode_enable**, the client access token of the caller must have elevated privileges exposed by the following enumeration in **WLAN_SECURABLE_OBJECT**:

- **wlan_secure_hosted_network_elevated_access**

If the **WlanHostedNetworkSetProperty** function is passed any of the following values in the *OpCode* parameter, the function will fail with **ERROR_NOT_SUPPORTED**:

- wlan_hosted_network_opcode_station_profile
- wlan_hosted_network_opcode_connection_settings

In order to succeed, the **WlanHostedNetworkSetProperty** function must persist the new settings which requires that the Hosted Network state be transitioned to **wlan_hosted_network_idle** if it was currently running (wlan_hosted_network_active).

Any user can call this function to set the Hosted Network properties. However, to set the **wlan_hosted_network_opcode_enable** flag requires elevated privileges. The ability to enable the wireless Hosted Network may also be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_CONNECTION_SETTINGS

WLAN_HOSTED_NETWORK_OPCODE

WLAN_HOSTED_NETWORK_REASON

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkInitSettings

WlanHostedNetworkQueryProperty

WlanHostedNetworkQuerySecondaryKey

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetSecondaryKey

WlanOpenHandle

WlanHostedNetworkSetSecondaryKey function (wlanapi.h)

7/1/2021 • 5 minutes to read • Edit Online

The **WlanHostedNetworkSetSecondaryKey** function configures the secondary security key that will be used by the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkSetSecondaryKey(
  HANDLE                      hClientHandle,
  DWORD                       dwKeyLength,
  PUCHAR                      pucKeyData,
  BOOL                        bIsPassPhrase,
  BOOL                        bPersistent,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`dwKeyLength`

The number of valid data bytes in the key data array pointed to by the *pucKeyData* parameter. This key length should include the terminating '\0' if the key is a passphrase.

`pucKeyData`

A pointer to a buffer that contains the key data. The number of valid data bytes in the buffer must be at least the value specified in *dwKeyLength* parameter.

`bIsPassPhrase`

A Boolean value that indicates if the key data array pointed to by the *pucKeyData* parameter is in passphrase format.

If this parameter is **TRUE**, the key data array is in passphrase format. If this parameter is **FALSE**, the key data array is not in passphrase format.

`bPersistent`

A Boolean value that indicates if the key data array pointed to by the *pucKeyData* parameter is to be stored and reused later or is for one-time use only.

If this parameter is **TRUE**, the key data array is to be stored and reused later. If this parameter is **FALSE**, the key data array is to be used for one session (either the current session or the next session if the Hosted Network is not started).

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the **WlanHostedNetworkSetSecondaryKey** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pucKeyData* is **NULL**. • *pucKeyData* does not point to a well- formed valid key. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkSetSecondaryKey** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkSetSecondaryKey** function to configure the secondary security key that will be used by the wireless Hosted Network. Any Hosted Network change caused by this function would not be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

Once started, the wireless Hosted Network will allow wireless peers to associate with this secondary security key in addition to the primary security key. The secondary security key is always specified by the user as needed, while the primary security key is generated by the operating system with greater security strength.

The secondary security key passed in the buffer pointed to by the *pucKeyData* parameter is used with WPA2-Personal authentication and should be in one of the following formats:

- A key passphrase that consists of an array of ASCII characters from 8 to 63 characters. The *dwKeyLength* parameter should include the terminating '\0' in the passphrase. The value of the *dwKeyLength* parameter should be in the range of 9 to 64.
- A binary key that consists of 32 bytes of binary key data. The *dwKeyLength* parameter should be 32 for binary key.

To configure a valid secondary security key, the *dwKeyLength* parameter should be in the correct range and the *pucKeyData* parameter should point to a valid memory buffer containing the specified bytes of data. To remove the currently configured secondary security key from the system, the application should call the **WlanHostedNetworkSetSecondaryKey** function with zero in *dwKeyLength* parameter and **NULL** in the *pucKeyData* parameter.

The **WlanHostedNetworkSetSecondaryKey** function will return **ERROR_INVALID_PARAMETER** if the *pucKeyData* parameter is **NULL**, but the *dwKeyLength* parameter is not zero. The **WlanHostedNetworkSetSecondaryKey** function will also return **ERROR_INVALID_PARAMETER** if the *dwKeyLength* parameter is zero, but *pucKeyData* parameter is not **NULL**.

The secondary security key is usually set before the wireless Hosted Network is started. Then it will be used the next time when the Hosted Network is started.

A secondary security key can also be set after the Hosted Network has been started. In this case, the secondary security key will be used immediately. Any clients using the previous secondary security key will remain connected, but they will be unable to reconnect if they get disconnected for any reason or if the wireless Hosted Network is restarted.

The secondary security key can be specified as persistent if the *bPersistent* parameter is set to **TRUE**. When specified as persistent, the secondary security key would be used immediately if the Hosted Network is already started, and also reused whenever Hosted Network is started in the future.

If secondary security key is not specified as persistent, it will be used immediately if the Hosted Network is already started, or only for the next time when Hosted Network is started. After the Hosted Network is stopped, this secondary security key will never be used again and will be removed from the system.

Any user can call this function to configure the secondary security key to be used in the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkInitSettings

WlanHostedNetworkQueryProperty

WlanHostedNetworkQuerySecondaryKey

WlanHostedNetworkRefreshSecuritySettings

WlanHostedNetworkSetProperty

WlanOpenHandle

WlanHostedNetworkStartUsing function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanHostedNetworkStartUsing** function starts the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkStartUsing(
  HANDLE                     hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                      pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason, if the call to the **WlanHostedNetworkStartUsing** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This error is returned if the wireless Hosted Network is disabled by group policy on a domain.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.

Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkStartUsing** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanHostedNetworkStartUsing** function to start the wireless Hosted Network. Successful calls must be matched by calls to WlanHostedNetworkStopUsing function. This call could fail if Hosted Network state is **wlan_hosted_network_unavailable**.

Any Hosted Network state change caused by this function would be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

Any user can call the **WlanHostedNetworkStartUsing** function to start the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkForceStart

WlanHostedNetworkForceStop

WlanHostedNetworkQueryStatus

WlanHostedNetworkStopUsing

WlanOpenHandle

WlanHostedNetworkStopUsing function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanHostedNetworkStopUsing** function stops the wireless Hosted Network.

Syntax

```
DWORD WlanHostedNetworkStopUsing(
  HANDLE                      hClientHandle,
  PWLAN_HOSTED_NETWORK_REASON pFailReason,
  PVOID                       pvReserved
);
```

Parameters

`hClientHandle`

The client's session handle, returned by a previous call to the WlanOpenHandle function.

`pFailReason`

An optional pointer to a value that receives the failure reason if the call to the **WlanHostedNetworkStopUsing** function fails. Possible values for the failure reason are from the WLAN_HOSTED_NETWORK_REASON enumeration type defined in the *Wlanapi.h* header file.

`pvReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This can occur if the wireless Hosted Network was in the process of shutting down.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.

Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanHostedNetworkStopUsing** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

An application calls the **WlanHostedNetworkStopUsing** function to stop the Hosted Network. A application calls the **WlanHostedNetworkStopUsing** function to match earlier successful calls to the WlanHostedNetworkStartUsing function. The wireless Hosted Network will remain active until all applications have called the **WlanHostedNetworkStopUsing** function or the WlanHostedNetworkForceStop function is called to force a stop. When the wireless Hosted Network has stopped, the state switches to **wlan_hosted_network_idle**. This call could also fail if the Hosted Network state changed because of external events (for example, if the miniport driver for the wireless interface card becomes unavailable).

Any user can call this function to stop the Hosted Network. However, the ability to enable the wireless Hosted Network may be restricted by group policy in a domain.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WLAN_HOSTED_NETWORK_REASON

WlanCloseHandle

WlanEnumInterfaces

WlanHostedNetworkForceStart

WlanHostedNetworkForceStop

WlanHostedNetworkQueryStatus

WlanHostedNetworkStartUsing

WlanOpenHandle

WlanIhvControl function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanIhvControl** function provides a mechanism for independent hardware vendor (IHV) control of WLAN drivers or services.

Syntax

```
DWORD WlanIhvControl(
  HANDLE                hClientHandle,
  const GUID            *pInterfaceGuid,
  WLAN_IHV_CONTROL_TYPE Type,
  DWORD                 dwInBufferSize,
  PVOID                 pInBuffer,
  DWORD                 dwOutBufferSize,
  PVOID                 pOutBuffer,
  PDWORD                pdwBytesReturned
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`Type`

A WLAN_IHV_CONTROL_TYPE structure that specifies the type of software bypassed by the IHV control function.

`dwInBufferSize`

The size, in bytes, of the input buffer.

`pInBuffer`

A generic buffer for driver or service interface input.

`dwOutBufferSize`

The size, in bytes, of the output buffer.

`pOutBuffer`

A generic buffer for driver or service interface output.

`pdwBytesReturned`

The number of bytes returned.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to perform this operation. When called, WlanIhvControl retrieves the discretionary access control list (DACL) stored with the **wlan_secure_ihv_control** object. If the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanIhvControl** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pInterfaceGuid* is **NULL**, or *pdwBytesReturned* is **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_IHV_CONTROL_TYPE

WlanOpenHandle function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanOpenHandle** function opens a connection to the server.

Syntax

```
DWORD WlanOpenHandle(
  DWORD   dwClientVersion,
  PVOID   pReserved,
  PDWORD  pdwNegotiatedVersion,
  PHANDLE phClientHandle
);
```

Parameters

`dwClientVersion`

The highest version of the WLAN API that the client supports.

VALUE	MEANING
1	Client version for Windows XP with SP3 and Wireless LAN API for Windows XP with SP2.
2	Client version for Windows Vista and Windows Server 2008

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwNegotiatedVersion`

The version of the WLAN API that will be used in this session. This value is usually the highest version supported by both the client and server.

`phClientHandle`

A handle for the client to use in this session. This handle is used by other functions throughout the session.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	*pdwNegotiatedVersion* is **NULL**, *phClientHandle* is **NULL**, or *pReserved* is not **NULL**.

ERROR_NOT_ENOUGH_MEMORY	Failed to allocate memory to create the client context.
RPC_STATUS	Various error codes.
ERROR_REMOTE_SESSION_LIMIT_EXCEEDED	Too many handles have been issued by the server.

Remarks

The version number specified by *dwClientVersion* and *pdwNegotiatedVersion* is a composite version number made up of both major and minor versions. The major version is specified by the low-order word, and the minor version is specified by the high-order word. The macros `WLAN_API_VERSION_MAJOR(_v)` and `WLAN_API_VERSION_MINOR(_v)` return the major and minor version numbers respectively. You can construct a version number using the macro `WLAN_API_MAKE_VERSION(_major, _minor)`.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: **WlanOpenHandle** will return an error message if the Wireless Zero Configuration (WZC) service has not been started or if the WZC service is not responsive.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanCloseHandle

WlanQueryAutoConfigParameter function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanQueryAutoConfigParameter** function queries for the parameters of the auto configuration service.

Syntax

```
DWORD WlanQueryAutoConfigParameter(
  HANDLE                  hClientHandle,
  WLAN_AUTOCONF_OPCODE    OpCode,
  PVOID                   pReserved,
  PDWORD                  pdwDataSize,
  PVOID                   *ppData,
  PWLAN_OPCODE_VALUE_TYPE pWlanOpcodeValueType
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`OpCode`

A value that specifies the configuration parameter to be queried.

VALUE	MEANING
wlan_autoconf_opcode_show_denied_networks	When set, the *ppData* parameter will contain a **BOOL** value indicating whether user and group policy-denied networks will be included in the available networks list. If the function returns ERROR_SUCCESS and *ppData* points to **TRUE**, then user and group policy-denied networks will be included in the available networks list; if **FALSE**, user and group policy-denied networks will not be included in the available networks list.
wlan_autoconf_opcode_power_setting	When set, the *ppData* parameter will contain a WLAN_POWER_SETTING value specifying the power settings.
wlan_autoconf_opcode_only_use_gp_profiles_for_allowed_networks	When set, the *ppData* parameter will contain a **BOOL** value indicating whether profiles not created by group policy can be used to connect to an allowed network with a matching group policy profile. If the function returns ERROR_SUCCESS and *ppData* points to **TRUE**, then only profiles created by group policy can be used; if **FALSE**, any profile can be used.

wlan_autoconf_opcode_allow_explicit_creds	When set, the *ppData* parameter will contain a **BOOL** value indicating whether the current wireless interface has shared user credentials allowed. If the function returns ERROR_SUCCESS and *ppData* points to **TRUE**, then the current wireless interface has shared user credentials allowed; if **FALSE**, the current wireless interface does not allow shared user credentials.
wlan_autoconf_opcode_block_period	When set, the *ppData* parameter will contain a **DWORD** value that indicates the blocked period setting for the current wireless interface. The blocked period is the amount of time, in seconds, for which automatic connection to a wireless network will not be attempted after a previous failure.
wlan_autoconf_opcode_allow_virtual_station_extensibility	When set, the *ppData* parameter will contain a **BOOL** value indicating whether extensibility on a virtual station is allowed. By default, extensibility on a virtual station is allowed. The value for this opcode is persisted across restarts. If the function returns ERROR_SUCCESS and *ppData* points to **TRUE**, then extensibility on a virtual station is allowed; if **FALSE**, extensibility on a virtual station is not allowed.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwDataSize`

Specifies the size of the *ppData* parameter, in bytes.

`ppData`

Pointer to the memory that contains the queried value for the parameter specified in *OpCode*.

> **Note** If *OpCode* is set to **wlan_autoconf_opcode_show_denied_networks**, then the pointer referenced by *ppData* may point to an integer value. If the pointer referenced by *ppData* points to 0, then the integer value should be converted to the boolean value **FALSE**. If the pointer referenced by *ppData* points to a nonzero integer, then the integer value should be converted to the boolean value **TRUE**.

`pWlanOpcodeValueType`

A WLAN_OPCODE_VALUE_TYPE value.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to get configuration parameters. When called with *OpCode* set to **wlan_autoconf_opcode_show_denied_networks**, WlanQueryAutoConfigParameter retrieves the discretionary access control list (DACL) stored with the **wlan_secure_show_denied** object. If the DACL does not contain an access control entry (ACE) that grants WLAN_READ_ACCESS permission to the access token of the calling thread, then **WlanQueryAutoConfigParameter** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pReserved* is not **NULL**, *ppData* is **NULL**, or *pdwDataSize* is **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

The **WlanQueryAutoConfigParameter** function queries for the parameters used by Auto Configuration Module (ACM), the wireless configuration component supported on Windows Vista and later.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_AUTOCONF_OPCODE

WlanSetAutoConfigParameter

WlanQueryInterface function (wlanapi.h)

7/1/2021 • 6 minutes to read • Edit Online

The **WlanQueryInterface** function queries various parameters of a specified interface.

Syntax

```
DWORD WlanQueryInterface(
  HANDLE                 hClientHandle,
  const GUID             *pInterfaceGuid,
  WLAN_INTF_OPCODE       OpCode,
  PVOID                  pReserved,
  PDWORD                 pdwDataSize,
  PVOID                  *ppData,
  PWLAN_OPCODE_VALUE_TYPE pWlanOpcodeValueType
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface to be queried.

`OpCode`

A WLAN_INTF_OPCODE value that specifies the parameter to be queried. The following table lists the valid constants along with the data type of the parameter in *ppData*.

WLAN_INTF_OPCODE VALUE	*PPDATA* DATA TYPE
wlan_intf_opcode_autoconf_enabled	BOOL
wlan_intf_opcode_background_scan_enabled	BOOL
wlan_intf_opcode_radio_state	WLAN_RADIO_STATE
wlan_intf_opcode_bss_type	DOT11_BSS_TYPE
wlan_intf_opcode_interface_state	WLAN_INTERFACE_STATE
wlan_intf_opcode_current_connection	WLAN_CONNECTION_ATTRIBUTES
wlan_intf_opcode_channel_number	ULONG
wlan_intf_opcode_supported_infrastructure_auth_cipher_pairs	WLAN_AUTH_CIPHER_PAIR_LIST
wlan_intf_opcode_supported_adhoc_auth_cipher_pairs	WLAN_AUTH_CIPHER_PAIR_LIST

wlan_intf_opcode_supported_country_or_region_string_list	WLAN_COUNTRY_OR_REGION_STRING_LIST
wlan_intf_opcode_media_streaming_mode	BOOL
wlan_intf_opcode_statistics	WLAN_STATISTICS
wlan_intf_opcode_rssi	LONG
wlan_intf_opcode_current_operation_mode	ULONG
wlan_intf_opcode_supported_safe_mode	BOOL
wlan_intf_opcode_certified_safe_mode	BOOL

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_intf_opcode_autoconf_enabled**, **wlan_intf_opcode_bss_type**, **wlan_intf_opcode_interface_state**, and **wlan_intf_opcode_current_connection** constants are valid.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwDataSize`

The size of the *ppData* parameter, in bytes.

`ppData`

Pointer to the memory location that contains the queried value of the parameter specified by the *OpCode* parameter.

> **Note** If *OpCode* is set to **wlan_intf_opcode_autoconf_enabled**, **wlan_intf_opcode_background_scan_enabled**, or **wlan_intf_opcode_media_streaming_mode**, then the pointer referenced by *ppData* may point to an integer value. If the pointer referenced by *ppData* points to 0, then the integer value should be converted to the boolean value **FALSE**. If the pointer referenced by *ppData* points to a nonzero integer, then the integer value should be converted to the boolean value **TRUE**.

`pWlanOpcodeValueType`

If passed a non-**NULL** value, points to a WLAN_OPCODE_VALUE_TYPE value that specifies the type of opcode returned. This parameter may be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

Remarks

The caller is responsible for using WlanFreeMemory to free the memory allocated for *ppData*.

When *OpCode* is set to **wlan_intf_opcode_current_operation_mode**, **WlanQueryInterface** queries the current operation mode of the wireless interface. For more information about operation modes, see Native

802.11 Operation Modes. Two operation modes are supported: **DOT11_OPERATION_MODE_EXTENSIBLE_STATION** and **DOT11_OPERATION_MODE_NETWORK_MONITOR**. The operation mode constants are defined in the header file Windot11.h. *ppData* will point to one of these two values.

Examples

The following example enumerates the wireless LAN interfaces on the local computer, queries each interface for the WLAN_CONNECTION_ATTRIBUTES on the interface, and prints values from the retrieved **WLAN_CONNECTION_ATTRIBUTES** structure.

For another example using the **WlanQueryInterface** function, see the WLAN_RADIO_STATE structure.

> **Note** This example will fail to load on Windows Server 2008 and Windows Server 2008 R2 if the Wireless LAN Service is not installed and started.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <wlanapi.h>
#include <Windot11.h>           // for DOT11_SSID struct
#include <objbase.h>
#include <wtypes.h>

//#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>

// Need to link with Wlanapi.lib and Ole32.lib
#pragma comment(lib, "wlanapi.lib")
#pragma comment(lib, "ole32.lib")

int wmain()
{

    // Declare and initialize variables.

    HANDLE hClient = NULL;
    DWORD dwMaxClient = 2;      //
    DWORD dwCurVersion = 0;
    DWORD dwResult = 0;
    DWORD dwRetVal = 0;
    int iRet = 0;

    WCHAR GuidString[39] = { 0 };

    unsigned int i, k;

    // variables used for WlanEnumInterfaces

    PWLAN_INTERFACE_INFO_LIST pIfList = NULL;
    PWLAN_INTERFACE_INFO pIfInfo = NULL;

    // variables used for WlanQueryInterfaces for opcode = wlan_intf_opcode_current_connection
    PWLAN_CONNECTION_ATTRIBUTES pConnectInfo = NULL;
    DWORD connectInfoSize = sizeof(WLAN_CONNECTION_ATTRIBUTES);
    WLAN_OPCODE_VALUE_TYPE opCode = wlan_opcode_value_type_invalid;

    dwResult = WlanOpenHandle(dwMaxClient, NULL, &dwCurVersion, &hClient);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanOpenHandle failed with error: %u\n", dwResult);
```

```
        return 1;
        // You can use FormatMessage here to find out why the function failed
    }

    dwResult = WlanEnumInterfaces(hClient, NULL, &pIfList);
    if (dwResult != ERROR_SUCCESS) {
        wprintf(L"WlanEnumInterfaces failed with error: %u\n", dwResult);
        return 1;
        // You can use FormatMessage here to find out why the function failed
    } else {
        wprintf(L"Num Entries: %lu\n", pIfList->dwNumberOfItems);
        wprintf(L"Current Index: %lu\n", pIfList->dwIndex);
        for (i = 0; i < (int) pIfList->dwNumberOfItems; i++) {
            pIfInfo = (WLAN_INTERFACE_INFO *) & pIfList->InterfaceInfo[i];
            wprintf(L"  Interface Index[%u]:\t %lu\n", i, i);
            iRet =
                StringFromGUID2(pIfInfo->InterfaceGuid, (LPOLESTR) & GuidString,
                                sizeof (GuidString) / sizeof (*GuidString));
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&pIfInfo->InterfaceGuid, (LPOLESTR) &GuidString,
            //     sizeof(GuidString)/sizeof(*GuidString));
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"  InterfaceGUID[%d]:\t %ws\n", i, GuidString);
            }
            wprintf(L"  Interface Description[%d]: %ws", i, pIfInfo->strInterfaceDescription);
            wprintf(L"\n");
            wprintf(L"  Interface State[%d]:\t ", i);
            switch (pIfInfo->isState) {
            case wlan_interface_state_not_ready:
                wprintf(L"Not ready\n");
                break;
            case wlan_interface_state_connected:
                wprintf(L"Connected\n");
                break;
            case wlan_interface_state_ad_hoc_network_formed:
                wprintf(L"First node in a ad hoc network\n");
                break;
            case wlan_interface_state_disconnecting:
                wprintf(L"Disconnecting\n");
                break;
            case wlan_interface_state_disconnected:
                wprintf(L"Not connected\n");
                break;
            case wlan_interface_state_associating:
                wprintf(L"Attempting to associate with a network\n");
                break;
            case wlan_interface_state_discovering:
                wprintf(L"Auto configuration is discovering settings for the network\n");
                break;
            case wlan_interface_state_authenticating:
                wprintf(L"In process of authenticating\n");
                break;
            default:
                wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                break;
            }
            wprintf(L"\n");

            // If interface state is connected, call WlanQueryInterface
            // to get current connection attributes
            if (pIfInfo->isState == wlan_interface_state_connected) {
                dwResult = WlanQueryInterface(hClient,
                                              &pIfInfo->InterfaceGuid,
                                              wlan_intf_opcode_current_connection,
                                              NULL,
                                              &connectInfoSize,
                                              (PVOID *) &pConnectInfo,
```

```
                                            &opCode);

                if (dwResult != ERROR_SUCCESS) {
                    wprintf(L"WlanQueryInterface failed with error: %u\n", dwResult);
                    dwRetVal = 1;
                    // You can use FormatMessage to find out why the function failed
                } else {
                    wprintf(L"  WLAN_CONNECTION_ATTRIBUTES for this interface\n");

                    wprintf(L"  Interface State:\t ");
                    switch (pConnectInfo->isState) {
                    case wlan_interface_state_not_ready:
                        wprintf(L"Not ready\n");
                        break;
                    case wlan_interface_state_connected:
                        wprintf(L"Connected\n");
                        break;
                    case wlan_interface_state_ad_hoc_network_formed:
                        wprintf(L"First node in a ad hoc network\n");
                        break;
                    case wlan_interface_state_disconnecting:
                        wprintf(L"Disconnecting\n");
                        break;
                    case wlan_interface_state_disconnected:
                        wprintf(L"Not connected\n");
                        break;
                    case wlan_interface_state_associating:
                        wprintf(L"Attempting to associate with a network\n");
                        break;
                    case wlan_interface_state_discovering:
                        wprintf
                            (L"Auto configuration is discovering settings for the network\n");
                        break;
                    case wlan_interface_state_authenticating:
                        wprintf(L"In process of authenticating\n");
                        break;
                    default:
                        wprintf(L"Unknown state %ld\n", pIfInfo->isState);
                        break;
                    }

                    wprintf(L"  Connection Mode:\t ");
                    switch (pConnectInfo->wlanConnectionMode) {
                    case wlan_connection_mode_profile:
                        wprintf(L"A profile is used to make the connection\n");
                        break;
                    case wlan_connection_mode_temporary_profile:
                        wprintf(L"A temporary profile is used to make the connection\n");
                        break;
                    case wlan_connection_mode_discovery_secure:
                        wprintf(L"Secure discovery is used to make the connection\n");
                        break;
                    case wlan_connection_mode_discovery_unsecure:
                        wprintf(L"Unsecure discovery is used to make the connection\n");
                        break;
                    case wlan_connection_mode_auto:
                        wprintf
                            (L"connection initiated by wireless service automatically using a persistent
profile\n");
                        break;
                    case wlan_connection_mode_invalid:
                        wprintf(L"Invalid connection mode\n");
                        break;
                    default:
                        wprintf(L"Unknown connection mode %ld\n",
                                pConnectInfo->wlanConnectionMode);
                        break;
                    }
```

```
                    wprintf(L"  Profile name used:\t %ws\n", pConnectInfo->strProfileName);

                    wprintf(L"  Association Attributes for this connection\n");
                    wprintf(L"    SSID:\t\t ");
                    if (pConnectInfo->wlanAssociationAttributes.dot11Ssid.uSSIDLength == 0)
                        wprintf(L"\n");
                    else {
                        for (k = 0;
                             k < pConnectInfo->wlanAssociationAttributes.dot11Ssid.uSSIDLength;
                             k++) {
                            wprintf(L"%c",
                                    (int) pConnectInfo->wlanAssociationAttributes.dot11Ssid.
                                    ucSSID[k]);
                        }
                        wprintf(L"\n");
                    }

                    wprintf(L"    BSS Network type:\t ");
                    switch (pConnectInfo->wlanAssociationAttributes.dot11BssType) {
                    case dot11_BSS_type_infrastructure:
                        wprintf(L"Infrastructure\n");
                        break;
                    case dot11_BSS_type_independent:
                        wprintf(L"Infrastructure\n");
                        break;
                    default:
                        wprintf(L"Other = %lu\n",
                                pConnectInfo->wlanAssociationAttributes.dot11BssType);
                        break;
                    }

                    wprintf(L"    MAC address:\t ");
                    for (k = 0; k < sizeof (pConnectInfo->wlanAssociationAttributes.dot11Bssid);
                         k++) {
                        if (k == 5)
                            wprintf(L"%.2X\n",
                                    pConnectInfo->wlanAssociationAttributes.dot11Bssid[k]);
                        else
                            wprintf(L"%.2X-",
                                    pConnectInfo->wlanAssociationAttributes.dot11Bssid[k]);
                    }

                    wprintf(L"    PHY network type:\t ");
                    switch (pConnectInfo->wlanAssociationAttributes.dot11PhyType) {
                    case dot11_phy_type_fhss:
                        wprintf(L"Frequency-hopping spread-spectrum (FHSS)\n");
                        break;
                    case dot11_phy_type_dsss:
                        wprintf(L"Direct sequence spread spectrum (DSSS)\n");
                        break;
                    case dot11_phy_type_irbaseband:
                        wprintf(L"Infrared (IR) baseband\n");
                        break;
                    case dot11_phy_type_ofdm:
                        wprintf(L"Orthogonal frequency division multiplexing (OFDM)\n");
                        break;
                    case dot11_phy_type_hrdsss:
                        wprintf(L"High-rate DSSS (HRDSSS) = \n");
                        break;
                    case dot11_phy_type_erp:
                        wprintf(L"Extended rate PHY type\n");
                        break;
                    case dot11_phy_type_ht:
                        wprintf(L"802.11n PHY type\n");
                        break;
                    default:
                        wprintf(L"Unknown = %lu\n",
                                pConnectInfo->wlanAssociationAttributes.dot11PhyType);
                        break;
```

```
    break;
}

wprintf(L"    PHY index:\t\t %u\n",
        pConnectInfo->wlanAssociationAttributes.uDot11PhyIndex);

wprintf(L"    Signal Quality:\t %d\n",
        pConnectInfo->wlanAssociationAttributes.wlanSignalQuality);

wprintf(L"    Receiving Rate:\t %ld\n",
        pConnectInfo->wlanAssociationAttributes.ulRxRate);

wprintf(L"    Transmission Rate:\t %ld\n",
        pConnectInfo->wlanAssociationAttributes.ulTxRate);
wprintf(L"\n");

wprintf(L"  Security Attributes for this connection\n");

wprintf(L"    Security enabled:\t ");
if (pConnectInfo->wlanSecurityAttributes.bSecurityEnabled == 0)
    wprintf(L"No\n");
else
    wprintf(L"Yes\n");

wprintf(L"    802.1X enabled:\t ");
if (pConnectInfo->wlanSecurityAttributes.bOneXEnabled == 0)
    wprintf(L"No\n");
else
    wprintf(L"Yes\n");

wprintf(L"    Authentication Algorithm: ");
switch (pConnectInfo->wlanSecurityAttributes.dot11AuthAlgorithm) {
case DOT11_AUTH_ALGO_80211_OPEN:
    wprintf(L"802.11 Open\n");
    break;
case DOT11_AUTH_ALGO_80211_SHARED_KEY:
    wprintf(L"802.11 Shared\n");
    break;
case DOT11_AUTH_ALGO_WPA:
    wprintf(L"WPA\n");
    break;
case DOT11_AUTH_ALGO_WPA_PSK:
    wprintf(L"WPA-PSK\n");
    break;
case DOT11_AUTH_ALGO_WPA_NONE:
    wprintf(L"WPA-None\n");
    break;
case DOT11_AUTH_ALGO_RSNA:
    wprintf(L"RSNA\n");
    break;
case DOT11_AUTH_ALGO_RSNA_PSK:
    wprintf(L"RSNA with PSK\n");
    break;
default:
    wprintf(L"Other (%lu)\n", pConnectInfo->wlanSecurityAttributes.dot11AuthAlgorithm);
    break;
}

wprintf(L"    Cipher Algorithm:\t ");
switch (pConnectInfo->wlanSecurityAttributes.dot11CipherAlgorithm) {
case DOT11_CIPHER_ALGO_NONE:
    wprintf(L"None\n");
    break;
case DOT11_CIPHER_ALGO_WEP40:
    wprintf(L"WEP-40\n");
    break;
case DOT11_CIPHER_ALGO_TKIP:
    wprintf(L"TKIP\n");
    break;
case DOT11_CIPHER_ALGO_CCMP:
```

```
                    case DOT11_CIPHER_ALGO_CCMP:
                        wprintf(L"CCMP\n");
                        break;
                    case DOT11_CIPHER_ALGO_WEP104:
                        wprintf(L"WEP-104\n");
                        break;
                    case DOT11_CIPHER_ALGO_WEP:
                        wprintf(L"WEP\n");
                        break;
                    default:
                        wprintf(L"Other (0x%x)\n", pConnectInfo-
>wlanSecurityAttributes.dot11CipherAlgorithm);
                        break;
                    }
                    wprintf(L"\n");
                }
            }
        }

    }
    if (pConnectInfo != NULL) {
        WlanFreeMemory(pConnectInfo);
        pConnectInfo = NULL;
    }

    if (pIfList != NULL) {
        WlanFreeMemory(pIfList);
        pIfList = NULL;
    }

    return dwRetVal;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

DOT11_BSS_TYPE

Native 802.11 Operation Modes

WLAN_AUTH_CIPHER_PAIR_LIST

WLAN_CONNECTION_ATTRIBUTES

WLAN_COUNTRY_OR_REGION_STRING_LIST

WLAN_INTERFACE_STATE

WLAN_INTF_OPCODE

WLAN_OPCODE_VALUE_TYPE

WLAN_RADIO_STATE

WLAN_STATISTICS

WlanFreeMemory

WlanOpenHandle

WlanSetInterface

WlanReasonCodeToString function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanReasonCodeToString** function retrieves a string that describes a specified reason code.

Syntax

```
DWORD WlanReasonCodeToString(
  DWORD  dwReasonCode,
  DWORD  dwBufferSize,
  PWCHAR pStringBuffer,
  PVOID  pReserved
);
```

Parameters

`dwReasonCode`

A WLAN_REASON_CODE value of which the string description is requested.

`dwBufferSize`

The size of the buffer used to store the string, in **WCHAR**. If the reason code string is longer than the buffer, it will be truncated and NULL-terminated. If *dwBufferSize* is larger than the actual amount of memory allocated to *pStringBuffer*, then an access violation will occur in the calling program.

`pStringBuffer`

Pointer to a buffer that will receive the string. The caller must allocate memory to *pStringBuffer* before calling **WlanReasonCodeToString**.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is a pointer to a constant string.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *dwBufferSize* is 0. • *pStringBuffer* is **NULL**. • *pReserved* is not **NULL**.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WLAN_REASON_CODE

WlanRegisterDeviceServiceNotification function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Allows user mode clients with admin privileges, or User-Mode Driver Framework (UMDF) drivers, to register for unsolicited notifications corresponding to device services that they're interested in.

Syntax

```
DWORD WlanRegisterDeviceServiceNotification(
  HANDLE                              hClientHandle,
  const PWLAN_DEVICE_SERVICE_GUID_LIST pDevSvcGuidList
);
```

Parameters

`hClientHandle`

Type: HANDLE

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pDevSvcGuidList`

Type: CONST PWLAN_DEVICE_SERVICE_GUID_LIST

An optional pointer to a constant WLAN_DEVICE_SERVICE_GUID_LIST structure representing the device service GUIDs for which you're interested in receiving notifications. The *dwIndex* member of the structure must have a value less than the value of its *dwNumberOfItems* member; otherwise, an access violation may occur. Every time you call this API, the previous device services list is replaced by the new one.

To unregister, set *pDevSvcGuidList* to `nullptr`, or pass a pointer to a **WLAN_DEVICE_SERVICE_GUID_LIST** structure that has the `dwNumberOfItems` member set to 0.

Return value

Type: HRESULT

If the function succeeds, the return value is **ERROR_SUCCESS**. If the function fails with **ERROR_ACCESS_DENIED**, then the caller doesn't have sufficient permissions to perform this operation. The caller needs to either have admin privilege, or needs to be a UMDF driver.

Remarks

The **WlanRegisterDeviceServiceNotification** function is an extension to existing native Wi-Fi APIs for WLAN device services.

A client application calls this function to register and unregister notifications for device services that it is interested in.

Any registration to receive notifications for device services caused by this function would be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle*

parameter), or if the process ends.

In order to receive these notifications, a client needs to call this function with a valid *pDevSvcGuidList* parameter, and must also call the WlanRegisterNotification function with a *dwNotifSource* argument of **WLAN_NOTIFICATION_SOURCE_DEVICE_SERVICE** (which is defined in `wlanapi.h`). The registration to receive notifications for device services is in effect until the application closes the client handle (by calling WlanCloseHandle with the *hClientHandle* parameter), or the process ends, or **WlanRegisterDeviceServiceNotification** is called with a *pDevSvcGuidList* argument of `nullptr`, or else has *dwNumberOfItems* set to 0.

When the operating system (OS) receives a device service notification from an independent hardware vendor (IHV) driver, and a client has registered for these notifications using **WlanRegisterDeviceServiceNotification**, the client will receive them via the WLAN_NOTIFICATION_CALLBACK that it had registered through its call to WlanRegisterNotification. This callback will be called for every notification that the client has received (with a separate buffer for every notification).

The *NotificationSource* member of the WLAN_NOTIFICATION_DATA structure received by the callback function (that is, the *data* member) will be set to **WLAN_NOTIFICATION_SOURCE_DEVICE_SERVICE**. The data blob, the device service **GUID**, and the opcode associated with this notification will be present in the *pData* member of the **WLAN_NOTIFICATION_DATA**, which will point to a structure of type WLAN_DEVICE_SERVICE_NOTIFICATION_DATA.

> **NOTE**
> The WLAN service, or the OS, will not check to see whether the device service **GUID**s that the client registers for are actually supported by the IHV driver. It is up to the client to query for supported device services using WlanGetSupportedDeviceServices API if they need to.

Requirements

Header	wlanapi.h

WlanRegisterNotification function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanRegisterNotification** function is used to register and unregister notifications on all wireless interfaces.

Syntax

```
DWORD WlanRegisterNotification(
  HANDLE                    hClientHandle,
  DWORD                     dwNotifSource,
  BOOL                      bIgnoreDuplicate,
  WLAN_NOTIFICATION_CALLBACK funcCallback,
  PVOID                     pCallbackContext,
  PVOID                     pReserved,
  PDWORD                    pdwPrevNotifSource
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`dwNotifSource`

The notification sources to be registered. These flags may be combined. When this parameter is set to **WLAN_NOTIFICATION_SOURCE_NONE**, **WlanRegisterNotification** unregisters notifications on all wireless interfaces.

The possible values for this parameter are defined in the *Wlanapi.h* and *L2cmn.h* header files.

The following table shows possible values.

VALUE	MEANING
WLAN_NOTIFICATION_SOURCE_NONE	Unregisters notifications.
WLAN_NOTIFICATION_SOURCE_ALL	Registers for all notifications available on the version of the operating system, including those generated by the 802.1X module. For Windows XP with SP3 and Wireless LAN API for Windows XP with SP2, setting *dwNotifSource* to **WLAN_NOTIFICATION_SOURCE_ALL** is functionally equivalent to setting *dwNotifSource* to **WLAN_NOTIFICATION_SOURCE_ACM**.

WLAN_NOTIFICATION_SOURCE_ACM	Registers for notifications generated by the auto configuration module. **Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** Only the wlan_notification_acm_connection_complete and wlan_notification_acm_disconnected notifications are available.
WLAN_NOTIFICATION_SOURCE_HNWK	Registers for notifications generated by the wireless Hosted Network. This notification source is available on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.
WLAN_NOTIFICATION_SOURCE_ONEX	Registers for notifications generated by 802.1X.
WLAN_NOTIFICATION_SOURCE_MSM	Registers for notifications generated by MSM. **Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** This value is not supported.
WLAN_NOTIFICATION_SOURCE_SECURITY	Registers for notifications generated by the security module. No notifications are currently defined for **WLAN_NOTIFICATION_SOURCE_SECURITY**. **Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** This value is not supported.
WLAN_NOTIFICATION_SOURCE_IHV	Registers for notifications generated by independent hardware vendors (IHV). **Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** This value is not supported.
WLAN_NOTIFICATION_SOURCE_DEVICE_SERVICE	Registers for notifications generated by device services.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This parameter must be set to WLAN_NOTIFICATION_SOURCE_NONE, WLAN_NOTIFICATION_SOURCE_ALL, or WLAN_NOTIFICATION_SOURCE_ACM.

`bIgnoreDuplicate`

Specifies whether duplicate notifications will be ignored. If set to **TRUE**, a notification will not be sent to the client if it is identical to the previous one.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This parameter is ignored.

`funcCallback`

A WLAN_NOTIFICATION_CALLBACK type that defines the type of notification callback function.

This parameter can be **NULL** if the *dwNotifSource* parameter is set to **WLAN_NOTIFICATION_SOURCE_NONE** to unregister notifications on all wireless interfaces,

`pCallbackContext`

A pointer to the client context that will be passed to the callback function with the notification.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwPrevNotifSource`

A pointer to the previously registered notification sources.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if *hClientHandle* is **NULL** or not valid or if *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_ENOUGH_MEMORY	Failed to allocate memory for the query results.
RPC_STATUS	Various error codes.

Remarks

The **WlanRegisterNotification** is used by an application to register and unregister notifications on all wireless interfaces. When registering for notifications, an application must provide a callback function pointed to by the *funcCallback* parameter. The prototype for this callback function is the WLAN_NOTIFICATION_CALLBACK. This callback function will receive notifications that have been registered for in the *dwNotifSource* parameter passed to the **WlanRegisterNotification** function. The callback function is called with a pointer to a WLAN_NOTIFICATION_DATA structure as the first parameter that contains detailed information on the notification. The callback function also receives a second parameter that contains a pointer to the client context passed in the *pCallbackContext* parameter to the **WlanRegisterNotification** function.

The **WlanRegisterNotification** function will return an error if *dwNotifSource* is a value other than **WLAN_NOTIFICATION_SOURCE_NONE** and the client fails to provide a callback function.

Once registered, the callback function will be called whenever a notification is available until the client unregisters or closes the handle.

Any registration to receive notifications caused by this function would be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

Do not call **WlanRegisterNotification** from a callback function. If the client is in the middle of a notification callback when **WlanRegisterNotification** is called with *dwNotifSource* set to **WLAN_NOTIFICATION_SOURCE_NONE** (that is, when the client is unregistering from notifications), **WlanRegisterNotification** will wait for the callback to finish before returning a value. Calling this function inside a callback function will result in the call never completing. If both the callback function and the thread that unregisters from notifications try to acquire the same lock, a deadlock may occur. In addition, do not call

WlanRegisterNotification from the **DllMain** function in an application DLL. This could also cause a deadlock.

An application can time out and query the current interface state instead of waiting for a notification.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Notifications are handled by the Netman service. If the Netman service is disabled or unavailable, notifications will not be received. If a notification is not received within a reasonable period of time, an application should time out and query the current interface state.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

ONEX_NOTIFICATION_TYPE

WLAN_HOSTED_NETWORK_NOTIFICATION_CODE

WLAN_NOTIFICATION_ACM

WLAN_NOTIFICATION_CALLBACK

WLAN_NOTIFICATION_DATA

WLAN_NOTIFICATION_MSM

WlanCloseHandle

WlanRegisterVirtualStationNotification

WlanRegisterVirtualStationNotification function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanRegisterVirtualStationNotification** function is used to register and unregister notifications on a virtual station.

Syntax

```
DWORD WlanRegisterVirtualStationNotification(
  HANDLE hClientHandle,
  BOOL   bRegister,
  PVOID  pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`bRegister`

A value that specifies whether to receive notifications on a virtual station.

`pReserved`

Reserved for future use. This parameter must be **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pvReserved* is not **NULL**.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_INVALID_STATE	The resource is not in the correct state to perform the requested operation. This error is returned if the wireless Hosted Network is disabled by group policy on a domain.

ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This error is returned if the WLAN AutoConfig Service is not running.
Other	Various RPC and other error codes. Use FormatMessage to obtain the message string for the returned error.

Remarks

The **WlanRegisterVirtualStationNotification** function is an extension to native wireless APIs added to support the wireless Hosted Network on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

A client application calls the **WlanRegisterVirtualStationNotification** function is used to register and unregister notifications on virtual station.

Any registration to receive notifications from a virtual station caused by this function would be automatically undone if the calling application closes its calling handle (by calling WlanCloseHandle with the *hClientHandle* parameter) or if the process ends.

By default, a application client will not receive notifications on a virtual station. In order to receive these notifications, a client needs to call the **WlanRegisterVirtualStationNotification** function with the *bRegister* parameter set to **TRUE** and must also call the WlanRegisterNotification function with the *dwNotifSource* parameter set to notification sources to be registered. The registration to receive notifications from a virtual station is in effect until the application closes the client handle (by calling WlanCloseHandle with the *hClientHandle* parameter), the process ends, or the **WlanRegisterVirtualStationNotification** function is called with the *bRegister* parameter set to **FALSE**.

On Windows 7 and later, the operating system installs a virtual device if a Hosted Network capable wireless adapter is present on the machine. This virtual device normally shows up in the "Network Connections Folder" as 'Wireless Network Connection 2' with a Device Name of 'Microsoft Virtual WiFi Miniport adapter' if the computer has a single wireless network adapter. This virtual device is used exclusively for performing software access point (SoftAP) connections and is not present in the list returned by the WlanEnumInterfaces function. The lifetime of this virtual device is tied to the physical wireless adapter. If the physical wireless adapter is disabled, this virtual device will be removed as well. This feature is also available on Windows Server 2008 R2 with the Wireless LAN Service installed.

Requirements

Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

Using Wireless Hosted Network and Internet Connection Sharing

WlanCloseHandle

WlanRegisterNotification

WlanRenameProfile function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanRenameProfile** function renames the specified profile.

Syntax

```
DWORD WlanRenameProfile(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strOldProfileName,
  LPCWSTR    strNewProfileName,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strOldProfileName`

The profile name to be changed.

`strNewProfileName`

The new name of the profile.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID PARAMETER	*hClientHandle* is **NULL** or not valid, *pInterfaceGuid* is **NULL**, *strOldProfileName* is **NULL**, *strNewProfileName* is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.

ERROR_NOT_FOUND	The profile specified by *strOldProfileName* was not found in the profile store.
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to rename the profile.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WlanDeleteProfile

WlanGetProfile

WlanSetProfile

WlanSaveTemporaryProfile function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanSaveTemporaryProfile** function saves a temporary profile to the profile store.

Syntax

```
DWORD WlanSaveTemporaryProfile(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  LPCWSTR    strAllUserProfileSecurity,
  DWORD      dwFlags,
  BOOL       bOverWrite,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strProfileName`

The name of the profile to be saved. Profile names are case-sensitive. This string must be NULL-terminated.

`strAllUserProfileSecurity`

Sets the security descriptor string on the all-user profile. By default, for a new all-user profile, all users have write access on the profile. For more information about profile permissions, see the Remarks section.

If *dwFlags* is set to WLAN_PROFILE_USER, this parameter is ignored.

If this parameter is set to **NULL** for an all-user profile, the default permissions are used.

If this parameter is not **NULL** for an all-user profile, the security descriptor string associated with the profile is created or modified after the security descriptor object is created and parsed as a string.

`dwFlags`

Specifies the flags to set on the profile. The flags can be combined.

VALUE	MEANING
0	The profile is an all-user profile.

WLAN_PROFILE_USER 0x00000002	The profile is a per-user profile.
WLAN_PROFILE_CONNECTION_MODE_SET_BY_CLIENT 0x00010000	The profile was created by the client.
WLAN_PROFILE_CONNECTION_MODE_AUTO 0x00020000	The profile was created by the automatic configuration module.

`bOverWrite`

Specifies whether this profile is overwriting an existing profile. If this parameter is **FALSE** and the profile already exists, the existing profile will not be overwritten and an error will be returned.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	One of the following conditions occurred: • *hClientHandle* is **NULL** or invalid. • *pInterfaceGuid* is **NULL**. • *pReserved* is not **NULL**. • *dwFlags* is not set to a combination of one or more of the values specified in the table above. • *dwFlags* is set to WLAN_PROFILE_CONNECTION_MODE_AUTO and *strProfileName* is **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.
ERROR_INVALID_STATE	The interface is not currently connected using a temporary profile.

Remarks

A temporary profile is the one passed to WlanConnect or generated by the discovery engine. A network connection can be established using a temporary profile. Using this API saves the temporary profile and associated user data to the profile store.

A new profile is added at the top of the list after the group policy profiles. A profile's position in the list is not changed if an existing profile is overwritten.

All-user profiles have three associated permissions: read, write, and execute. If a user has read access, the user can view profile permissions. If a user has execute access, the user has read access and the user can also connect to and disconnect from a network using the profile. If a user has write access, the user has execute access and the user can also modify and delete permissions associated with a profile.

The following describes the procedure for creating a security descriptor object and parsing it as a string.

1. Call InitializeSecurityDescriptor to create a security descriptor in memory.
2. Call SetSecurityDescriptorOwner.
3. Call InitializeAcl to create a discretionary access control list (DACL) in memory.
4. Call AddAccessAllowedAce or AddAccessDeniedAce to add access control entries (ACEs) to the DACL. Set the *AccessMask* parameter to one of the following bitwise OR combinations as appropriate:
   - WLAN_READ_ACCESS
   - WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS
   - WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS | WLAN_WRITE_ACCESS
5. Call SetSecurityDescriptorDacl to add the DACL to the security descriptor.
6. Call ConvertSecurityDescriptorToStringSecurityDescriptor to convert the descriptor to string.

The string returned by ConvertSecurityDescriptorToStringSecurityDescriptor can then be used as the *strAllUserProfileSecurity* parameter value when calling **WlanSaveTemporaryProfile**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

Native Wifi API Permissions

WlanScan function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanScan** function requests a scan for available networks on the indicated interface.

Syntax

```
DWORD WlanScan(
  HANDLE               hClientHandle,
  const GUID           *pInterfaceGuid,
  const PDOT11_SSID    pDot11Ssid,
  const PWLAN_RAW_DATA pIeData,
  PVOID                pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface to be queried.

The GUID of each wireless LAN interface enabled on a local computer can be determined using the WlanEnumInterfaces function.

`pDot11Ssid`

A pointer to a DOT11_SSID structure that specifies the SSID of the network to be scanned. This parameter is optional. When set to **NULL**, the returned list contains all available networks. **Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** This parameter must be **NULL**.

`pIeData`

A pointer to an information element to include in probe requests. This parameter points to a WLAN_RAW_DATA structure that may include client provisioning availability information and 802.1X authentication requirements.**Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** This parameter must be **NULL**.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pInterfaceGuid* is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.
ERROR_NOT_ENOUGH_MEMORY	Failed to allocate memory for the query results.

Remarks

The **WlanScan** function requests that the native 802.11 Wireless LAN driver scan for available wireless networks. The driver may or may not send probe requests (an active scan) depending on its implementation and the values passed in the *pDot11Ssid* and *pIeData* parameters.

If the *pIeData* parameter is not **NULL**, the driver will send probe requests during the scan. The probe requests include the information element (IE) pointed to by the *pIeData* parameter. For instance, the Wi-Fi Protected Setup (WPS) IE can be included in the probe requests to discover WPS-capable access points. The buffer pointed to by the *pIeData* parameter must contain the complete IE starting from the Element ID.

The *pIeData* parameter passed to the **WlanScan** function can contain a pointer to an optional WLAN_RAW_DATA structure that contains a proximity service discovery (PSD) IE data entry.

When used to store a PSD IE, the **DOT11_PSD_IE_MAX_DATA_SIZE** constant defined in the *Wlanapi.h* header file is the maximum value of the **dwDataSize** member.

CONSTANT	VALUE	DESCRIPTION
DOT11_PSD_IE_MAX_DATA_SIZE	240	The maximum data size, in bytes, of a PSD IE data entry.

For more information about PSD IEs, including a discussion of the format of a PSD IE, see the WlanSetPsdIEDataList function.

When the **WlanScan** function is called, the native 802.11 Wireless LAN driver may flush the current list of available wireless networks before the scan is initiated. Applications should not assume that calling the **WlanScan** function will add to the existing list of available wireless networks returned by the WlanGetNetworkBssList or WlanGetAvailableNetworkList functions from previous scans.

The **WlanScan** function returns immediately. To be notified when the network scan is complete, a client on Windows Vista and later must register for notifications by calling WlanRegisterNotification. The *dwNotifSource* parameter passed to the **WlanRegisterNotification** function must have the WLAN_NOTIFICATION_SOURCE_ACM bit set to register for notifications generated by the auto configuration module. Wireless network drivers that meet Windows logo requirements are required to complete a **WlanScan** function request in 4 seconds.

The Wireless LAN Service does not send notifications when available wireless networks change. The Wireless LAN Service does not track changes to the list of available networks across multiple scans. The current default behavior is that the Wireless LAN Service only asks the wireless interface driver to scan for wireless networks every 60 seconds, and in some cases (when already connected to wireless network) the Wireless LAN Service

does not ask for scans at all. The **WlanScan** function can be used by an application to track wireless network changes. The application should first register for WLAN_NOTIFICATION_SOURCE_ACM notifications. The **WlanScan** function can then be called to initiate a scan. The application should then wait to receive the wlan_notification_acm_scan_complete notification or timeout after 4 seconds. Then the application can call the WlanGetNetworkBssList or WlanGetAvailableNetworkList function to retrieve a list of available wireless networks. This process can be repeated periodically with the application keeping tracking of changes to available wireless networks.

The **WlanScan** function returns immediately and does not provide a notification when the scan is complete on Windows XP with SP3 or the Wireless LAN API for Windows XP with SP2.

Since it becomes more difficult for a wireless interface to send and receive data packets while a scan is occurring, the **WlanScan** function may increase latency until the network scan is complete.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

DOT11_SSID

WLAN_RAW_DATA

WlanEnumInterfaces

WlanGetAvailableNetworkList

WlanGetNetworkBssList

WlanRegisterNotification

WlanSetPsdIEDataList

WlanSetAutoConfigParameter function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanSetAutoConfigParameter** function sets parameters for the automatic configuration service.

Syntax

```
DWORD WlanSetAutoConfigParameter(
  HANDLE               hClientHandle,
  WLAN_AUTOCONF_OPCODE OpCode,
  DWORD                dwDataSize,
  const PVOID          pData,
  PVOID                pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`OpCode`

A WLAN_AUTOCONF_OPCODE value that specifies the parameter to be set. Only some of the opcodes in the **WLAN_AUTOCONF_OPCODE** enumeration support set operations.

VALUE	MEANING
wlan_autoconf_opcode_show_denied_networks	When set, the *pData* parameter will contain a **BOOL** value indicating whether user and group policy-denied networks will be included in the available networks list.
wlan_autoconf_opcode_allow_explicit_creds	When set, the *pData* parameter will contain a **BOOL** value indicating whether the current wireless interface has shared user credentials allowed.
wlan_autoconf_opcode_block_period	When set, the *pData* parameter will contain a **DWORD** value for the blocked period setting for the current wireless interface. The blocked period is the amount of time, in seconds, for which automatic connection to a wireless network will not be attempted after a previous failure.
wlan_autoconf_opcode_allow_virtual_station_extensibility	When set, the *pData* parameter will contain a **BOOL** value indicating whether extensibility on a virtual station is allowed. By default, extensibility on a virtual station is allowed. The value for this opcode is persisted across restarts. This enumeration value is supported on Windows 7 and on Windows Server 2008 R2 with the Wireless LAN Service installed.

`dwDataSize`

The size of the *pData* parameter, in bytes. This parameter must be set to `sizeof(BOOL)` for a BOOL or `sizeof(DWORD)` for a DWORD, depending on the value of the *OpCode* parameter.

`pData`

The value to be set for the parameter specified in *OpCode* parameter. The *pData* parameter must point to a boolean or DWORD value, depending on the value of the *OpCode* parameter. The *pData* parameter must not be **NULL**.

> **Note** The *pData* parameter may point to an integer value when a boolean is required. If *pData* points to 0, then the value is converted to **FALSE**. If *pData* points to a nonzero integer, then the value is converted to **TRUE**.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	Access is denied. This error is returned if the caller does not have sufficient permissions to set the configuration parameter when the *OpCode* parameter is wlan_autoconf_opcode_show_denied_networks or wlan_autoconf_opcode_allow_virtual_station_extensibility. When the *OpCode* parameter is set to one of these values, the WlanSetAutoConfigParameter function retrieves the discretionary access control list (DACL) stored for opcode object. If the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanSetAutoConfigParameter** returns **ERROR_ACCESS_DENIED**. This error is also returned if the configuration parameter is set by group policy on a domain. When group policy is set for an opcode, applications are prevented from making changes. For the following *OpCode* parameters may be set by group policy: wlan_autoconf_opcode_show_denied_networks, wlan_autoconf_opcode_allow_explicit_creds, and wlan_autoconf_opcode_block_period
ERROR_INVALID_PARAMETER	A parameter was bad. This error is returned if the *hClientHandle* parameter is **NULL**, the *pData* parameter is **NULL**, or the *pReserved* parameter is not **NULL**. This error is also returned if *OpCode* parameter specified is not one of the WLAN_AUTOCONF_OPCODE values for a configuration parameter that can be set. This error is also returned if the *dwDataSize* parameter is not set to `sizeof(BOOL)`, or the *dwDataSize* is not set to `sizeof(BOOL)` depending on the value of the *OpCode* parameter.

ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

The **WlanSetAutoConfigParameter** function sets parameters used by Auto Configuration Module (ACM), the wireless configuration component supported on Windows Vista and later.

Depending on the value of the *OpCode* parameter, the data pointed to by *pData* will be converted to a boolean value before the automatic configuration parameter is set. If *pData* points to 0, then the parameter is set to **FALSE**; otherwise, the parameter is set to **TRUE**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_AUTOCONF_OPCODE

WlanQueryAutoConfigParameter

WlanSetFilterList function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanSetFilterList** function sets the permit/deny list.

Syntax

```
DWORD WlanSetFilterList(
  HANDLE                   hClientHandle,
  WLAN_FILTER_LIST_TYPE    wlanFilterListType,
  const PDOT11_NETWORK_LIST pNetworkList,
  PVOID                    pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`wlanFilterListType`

A WLAN_FILTER_LIST_TYPE value that specifies the type of filter list. The value must be either **wlan_filter_list_type_user_permit** or **wlan_filter_list_type_user_deny**. Group policy-defined lists cannot be set using this function.

`pNetworkList`

Pointer to a DOT11_NETWORK_LIST structure that contains the list of networks to permit or deny. The **dwIndex** member of the structure must have a value less than the value of the **dwNumberOfItems** member of the structure; otherwise, an access violation may occur.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to set the filter list. When called with *wlanFilterListType* set to **wlan_filter_list_type_user_permit**, WlanSetFilterList retrieves the discretionary access control list (DACL) stored with the **wlan_secure_permit_list** object. When called with *wlanFilterListType* set to **wlan_filter_list_type_user_deny**, **WlanSetFilterList** retrieves the DACL stored with the **wlan_secure_deny_list** object. In either of these cases, if the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanSetFilterList** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

The group policy permit and deny lists take precedence over the user's permit and deny lists. That means access to a network on the user's permit list will be denied if the network appears on the group policy deny list. Similarly, access to a network on the user's deny list will be permitted if the network appears on the group policy permit list. Networks that are not on a user list or a group policy list will be permitted.

Denied networks cannot be connected by means of auto config and will not be included on the visible networks list. New user permit and deny lists overwrite previous versions of the user lists.

To clear a filter list, set the *pNetworkList* parameter to **NULL**, or pass a pointer to a DOT11_NETWORK_LIST structure that has the **dwNumberOfItems** member set to 0.

To add all SSIDs to a filter list, pass a pointer to a DOT11_NETWORK_LIST structure with an associated DOT11_NETWORK structure that has the **uSSIDLength** member of its DOT11_SSID structure set to 0.

To add all BSS types to a filter list, pass a pointer to a DOT11_NETWORK_LIST with an associated DOT11_NETWORK structure that has its **dot11BssType** member set to **dot11_BSS_type_any**.

The **netsh wlan add filter** and **netsh wlan delete filter** commands provide similar functionality at the command line. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WlanGetFilterList

WlanSetInterface function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanSetInterface** function sets user-configurable parameters for a specified interface.

Syntax

```
DWORD WlanSetInterface(
  HANDLE           hClientHandle,
  const GUID       *pInterfaceGuid,
  WLAN_INTF_OPCODE OpCode,
  DWORD            dwDataSize,
  const PVOID      pData,
  PVOID            pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface to be configured.

`OpCode`

A WLAN_INTF_OPCODE value that specifies the parameter to be set. The following table lists the valid constants along with the data type of the parameter in *pData*.

WLAN_INTF_OPCODE VALUE	*PDATA* DATA TYPE	DESCRIPTION
wlan_intf_opcode_autoconf_enabled	**BOOL**	Enables or disables auto config for the indicated interface.
wlan_intf_opcode_background_scan_enabled	**BOOL**	Enables or disables background scan for the indicated interface.
wlan_intf_opcode_radio_state	WLAN_PHY_RADIO_STATE	Sets the software radio state of a specific physical layer (PHY) for the interface.
wlan_intf_opcode_bss_type	DOT11_BSS_TYPE	Sets the BSS type.
wlan_intf_opcode_media_streaming_mode	**BOOL**	Sets media streaming mode for the driver.
wlan_intf_opcode_current_operation_mode	**ULONG**	Sets the current operation mode for the interface. For more information, see Remarks.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Only the **wlan_intf_opcode_autoconf_enabled** and **wlan_intf_opcode_bss_type** constants are valid.

`dwDataSize`

The size of the *pData* parameter, in bytes. If *dwDataSize* is larger than the actual amount of memory allocated to *pData*, then an access violation will occur in the calling program.

`pData`

The value to be set as specified by the *OpCode* parameter. The type of data pointed to by *pData* must be appropriate for the specified *OpCode*. Use the table above to determine the type of data to use.

> **Note** If *OpCode* is set to **wlan_intf_opcode_autoconf_enabled**, **wlan_intf_opcode_background_scan_enabled**, or **wlan_intf_opcode_media_streaming_mode**, then *pData* may point to an integer value. If *pData* points to 0, then the value is converted to **FALSE**. If *pData* points to a nonzero integer, then the value is converted to **TRUE**.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

Remarks

When *OpCode* is set to **wlan_intf_opcode_current_operation_mode**, the **WlanSetInterface** function sets the current operation mode of the wireless interface. For more information about operation modes, see Native 802.11 Operation Modes. Two operation modes are supported: **DOT11_OPERATION_MODE_EXTENSIBLE_STATION** and **DOT11_OPERATION_MODE_NETWORK_MONITOR**. The operation mode constants are defined in the header file Windot11.h. If *pData* does not point to one of these values when *OpCode* is set to **wlan_intf_opcode_current_operation_mode**, the **WlanSetInterface** function will fail with an error.

To enable or disable the automatic configuration service at the command line, which is functionally equivalent to calling **WlanSetInterface** with *OpCode* set to **wlan_intf_opcode_autoconf_enabled**, use the **netsh wlan setautoconfig** command. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

The software radio state can be changed by calling the **WlanSetInterface** function. The hardware radio state cannot be changed by calling the **WlanSetInterface** function. When the *OpCode* parameter is set to **wlan_intf_opcode_radio_state**, the **WlanSetInterface** function sets the software radio state of a specific PHY. The *pData* parameter must point to a WLAN_PHY_RADIO_STATE structure with the new radio state values to use. The **dot11HardwareRadioState** member of the **WLAN_PHY_RADIO_STATE** structure is ignored when the **WlanSetInterface** function is called with the *OpCode* parameter set to **wlan_intf_opcode_radio_state** and the *pData* parameter points to a **WLAN_PHY_RADIO_STATE** structure. The radio state of a PHY is off if either the software radio state (**dot11SoftwareRadioState** member of the **WLAN_PHY_RADIO_STATE** structure) or the hardware radio state (**dot11HardwareRadioState** member of the **WLAN_PHY_RADIO_STATE** structure) is off.

Changing the software radio state of a physical network interface could cause related changes in the state of the wireless Hosted Network or virtual wireless adapter radio states. The PHYs of every virtual wireless adapter are linked. For more information, see the About the Wireless Hosted Network.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

About the Wireless Hosted Network

DOT11_BSS_TYPE

WLAN_INTF_OPCODE

WLAN_PHY_RADIO_STATE

WlanQueryInterface

WlanSetProfile function (wlanapi.h)

7/1/2021 • 6 minutes to read • Edit Online

The **WlanSetProfile** function sets the content of a specific profile.

Syntax

```
DWORD WlanSetProfile(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  DWORD      dwFlags,
  LPCWSTR    strProfileXml,
  LPCWSTR    strAllUserProfileSecurity,
  BOOL       bOverwrite,
  PVOID      pReserved,
  DWORD      *pdwReasonCode
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`dwFlags`

The flags to set on the profile.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: *dwFlags* must be 0. Per-user profiles are not supported.

VALUE	MEANING
0	The profile is an all-user profile.
WLAN_PROFILE_GROUP_POLICY 0x00000001	The profile is a group policy profile.
WLAN_PROFILE_USER 0x00000002	The profile is a per-user profile.

`strProfileXml`

Contains the XML representation of the profile. The WLANProfile element is the root profile element. To view sample profiles, see Wireless Profile Samples. There is no predefined maximum string length.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The supplied profile must

meet the compatibility criteria described in Wireless Profile Compatibility.

`strAllUserProfileSecurity`

Sets the security descriptor string on the all-user profile. For more information about profile permissions, see the Remarks section.

If *dwFlags* is set to WLAN_PROFILE_USER, this parameter is ignored.

If this parameter is set to **NULL** for a new all-user profile, the security descriptor associated with the wlan_secure_add_new_all_user_profiles object is used. If the security descriptor has not been modified by a WlanSetSecuritySettings call, all users have default permissions on a new all-user profile. Call WlanGetSecuritySettings to get the default permissions associated with the wlan_secure_add_new_all_user_profiles object.

If this parameter is set to **NULL** for an existing all-user profile, the permissions of the profile are not changed.

If this parameter is not **NULL** for an all-user profile, the security descriptor string associated with the profile is created or modified after the security descriptor object is created and parsed as a string.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This parameter must be **NULL**.

`bOverwrite`

Specifies whether this profile is overwriting an existing profile. If this parameter is **FALSE** and the profile already exists, the existing profile will not be overwritten and an error will be returned.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pdwReasonCode`

A WLAN_REASON_CODE value that indicates why the profile is not valid.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to set the profile. When called with *dwFlags* set to 0 - that is, when setting an all-user profile - WlanSetProfile retrieves the discretionary access control list (DACL) stored with the **wlan_secure_add_new_all_user_profiles** object. When called with *dwFlags* set to **WLAN_PROFILE_USER** - that is, when setting a per-user profile - **WlanSetProfile** retrieves the discretionary access control list (DACL) stored with the **wlan_secure_add_new_per_user_profiles** object. In either case, if the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanSetProfile** returns **ERROR_ACCESS_DENIED**.

ERROR_ALREADY_EXISTS	*strProfileXml* specifies a network that already exists. Typically, this return value is used when *bOverwrite* is **FALSE**; however, if *bOverwrite* is **TRUE** and *dwFlags* specifies a different profile type than the one used by the existing profile, then the existing profile will not be overwritten and **ERROR_ALREADY_EXISTS** will be returned.
ERROR_BAD_PROFILE	The profile specified by *strProfileXml* is not valid. If this value is returned, *pdwReasonCode* specifies the reason the profile is invalid.
ERROR_INVALID_PARAMETER	One of the following conditions occurred: • *hClientHandle* is **NULL** or invalid. • *pInterfaceGuid* is **NULL**. • *pReserved* is not **NULL**. • *strProfileXml* is **NULL**. [ConfigBlob](/windows/desktop/eaphost/eaphostconfigschema-configblob-eaphostconfig-element). If the profile must have an empty **ConfigBlob**, use `<ConfigBlob>00</ConfigBlob>` in the profile. • *pdwReasonCode* is **NULL**. • *dwFlags* is not set to one of the specified values. • *dwFlags* is set to WLAN_PROFILE_GROUP_POLICY and *bOverwrite* is set to **FALSE**.
ERROR_NO_MATCH	The interface does not support one or more of the capabilities specified in the profile. For example, if a profile specifies the use of WPA2 when the NIC only supports WPA, then this error code is returned. Also, if a profile specifies the use of FIPS mode when the NIC does not support FIPS mode, then this error code is returned.
RPC_STATUS	Various error codes.

Remarks

The **WlanSetProfile** function can be used to add a new wireless LAN profile or replace an existing wireless LAN profile.

A new profile is added at the top of the list after the group policy profiles. A profile's position in the list is not changed if an existing profile is overwritten.**Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:**

Ad hoc profiles appear after the infrastructure profiles in the profile list. If you create a new ad hoc profile, it is placed at the top of the ad hoc list, after the group policy and infrastructure profiles.

802.1X guest profiles, Wireless Provisioning Service (WPS) profiles, and profiles with Wi-Fi Protected Access-None (WPA-None) authentication are not supported. That means such a profile cannot be created, deleted, enumerated, or accessed using Native Wifi functions. Any such profile already in the preferred profile list will remain in the list, and its position in the list relative to other profiles is fixed unless the position of the other profiles change.

You can call **WlanSetProfile** on a profile that contains a plaintext key (that is, a profile with the protected element present and set to **FALSE**). Before the profile is saved in the profile store, the key material is automatically encrypted. When the profile is subsequently retrieved from the profile store by calling

WlanGetProfile, the encrypted key material is returned.**Windows XP with SP3 and Wireless LAN API for Windows XP with SP2:** The key material is never encrypted.

All-user profiles have three associated permissions: read, write, and execute. If a user has read access, the user can view profile permissions. If a user has execute access, the user has read access and the user can also connect to and disconnect from a network using the profile. If a user has write access, the user has execute access and the user can also modify and delete permissions associated with a profile.

The following describes the procedure for creating a security descriptor object and parsing it as a string.

1. Call InitializeSecurityDescriptor to create a security descriptor in memory.
2. Call SetSecurityDescriptorOwner.
3. Call InitializeAcl to create a discretionary access control list (DACL) in memory.
4. Call AddAccessAllowedAce or AddAccessDeniedAce to add access control entries (ACEs) to the DACL. Set the *AccessMask* parameter to one of the following as appropriate:
   - WLAN_READ_ACCESS
   - WLAN_EXECUTE_ACCESS
   - WLAN_WRITE_ACCESS
5. Call SetSecurityDescriptorDacl to add the DACL to the security descriptor.
6. Call ConvertSecurityDescriptorToStringSecurityDescriptor to convert the descriptor to string.

The string returned by ConvertSecurityDescriptorToStringSecurityDescriptor can then be used as the *strAllUserProfileSecurity* parameter value when calling **WlanSetProfile**.

For every wireless LAN profile used by the Native Wifi AutoConfig service, Windows maintains the concept of custom user data. This custom user data is initially non-existent, but can be set by calling the WlanSetProfileCustomUserData function. The custom user data gets reset to empty any time the profile is modified by calling the **WlanSetProfile** function. Once custom user data has been set, this data can be accessed using the WlanGetProfileCustomUserData function.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfile** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

The **netsh wlan add profile** command provides similar functionality at the command line. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib

DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

ConvertSecurityDescriptorToStringSecurityDescriptor

InitializeAcl

InitializeSecurityDescriptor

Native Wifi API Permissions

SetSecurityDescriptorDacl

WlanGetProfile

WlanGetProfileCustomUserData

WlanGetProfileList

WlanQueryInterface

WlanSetProfileCustomUserData

WlanSetProfileEapUserData

WlanSetProfileEapXmlUserData

WlanSetProfileCustomUserData function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanSetProfileCustomUserData** function sets the custom user data associated with a profile.

Syntax

```
DWORD WlanSetProfileCustomUserData(
  HANDLE      hClientHandle,
  const GUID  *pInterfaceGuid,
  LPCWSTR     strProfileName,
  DWORD       dwDataSize,
  const PBYTE pData,
  PVOID       pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strProfileName`

The name of the profile associated with the custom user data. Profile names are case-sensitive. This string must be NULL-terminated.

`dwDataSize`

The size of *pData*, in bytes.

`pData`

A pointer to the user data to be set.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_INVALID_PARAMETER	One of the following conditions occurred: • *hClientHandle* is **NULL** or invalid. • *pInterfaceGuid* is **NULL**. • *strProfileName* is **NULL**. • *pReserved* is not **NULL**. • *dwDataSize* is not 0 and *pData* is **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

For every wireless WLAN profile used by the Native Wifi AutoConfig service, Windows maintains the concept of custom user data. This custom user data is initially non-existent, but can be set by calling the **WlanSetProfileCustomUserData** function. The custom user data gets reset to empty any time the profile is modified by calling the WlanSetProfile function.

Once custom user data has been set, this data can be accessed using the WlanGetProfileCustomUserData function.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfileCustomUserData** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

WLAN_profile Schema

WlanGetProfile

WlanGetProfileCustomUserData

WlanGetProfileList

WlanSetProfile

WlanSetProfileEapUserData

WlanSetProfileEapXmlUserData

WlanSetProfileEapUserData function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanSetProfileEapUserData** function sets the Extensible Authentication Protocol (EAP) user credentials as specified by raw EAP data. The user credentials apply to a profile on an interface.

Syntax

```
DWORD WlanSetProfileEapUserData(
  HANDLE          hClientHandle,
  const GUID      *pInterfaceGuid,
  LPCWSTR         strProfileName,
  EAP_METHOD_TYPE eapType,
  DWORD           dwFlags,
  DWORD           dwEapUserDataSize,
  const LPBYTE    pbEapUserData,
  PVOID           pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strProfileName`

The name of the profile associated with the EAP user data. Profile names are case-sensitive. This string must be NULL-terminated.

`eapType`

An EAP_METHOD_TYPE structure that contains the method for which the caller is supplying EAP user credentials.

`dwFlags`

A set of flags that modify the behavior of the function.

On Windows Vista and Windows Server 2008, this parameter is reserved and should be set to zero.

On Windows 7, Windows Server 2008 R2, and later, this parameter can be one of the following values.

VALUE	MEANING
WLAN_SET_EAPHOST_DATA_ALL_USERS 0x00000001	Set EAP host data for all users of this profile.

`dwEapUserDataSize`

The size, in bytes, of the data pointed to by *pbEapUserData*.

`pbEapUserData`

A pointer to the raw EAP data used to set the user credentials.

On Windows Vista and Windows Server 2008, this parameter must not be **NULL**.

On Windows 7, Windows Server 2008 R2, and later, this parameter can be set to **NULL** to delete the stored credentials for this profile if the *dwFlags* parameter contains **WLAN_SET_EAPHOST_DATA_ALL_USERS** and the *dwEapUserDataSize* parameter is 0.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	Access is denied. This value is returned if the caller does not have write access to the profile.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This value is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pInterfaceGuid* is **NULL**. • *strProfileName* is **NULL** • *pvReserved* is not **NULL**. On Windows Vista and Windows Server 2008, this value is returned if the *pbEapUserData* parameter is **NULL**. On Windows 7, Windows Server 2008 R2 , and later, this error is returned if the *pbEapUserData* parameter is **NULL**, but the *dwEapUserDataSize* parameter is not 0 or the *dwFlags* parameter does not contain **WLAN_SET_EAPHOST_DATA_ALL_USERS**.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle *hClientHandle* was not found in the handle table.
ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.

ERROR_NOT_SUPPORTED	The request is not supported. This value is returned when profile settings do not permit storage of user data. This can occur when single signon (SSO) is enabled or when the request was to delete the stored credentials for this profile (the *pbEapUserData* parameter was **NULL**, the *dwFlags* parameter contains **WLAN_SET_EAPHOST_DATA_ALL_USERS**, and the *dwEapUserDataSize* parameter is 0). On Windows 10, Windows Server 2016 , and later, this value is returned if the WlanSetProfileEapUserData function was called on a profile that uses a method other than 802.1X for authentication. This value is also returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This value is returned if the Wireless LAN service is not running.
RPC_STATUS	Various error codes.

Remarks

The **WlanSetProfileEapUserData** function sets the EAP user credentials to use on a profile. On Windows Vista and Windows Server 2008, these credentials can only be used by the caller.

The *eapType* parameter is an EAP_METHOD_TYPE structure that contains type, identification, and author information about an EAP method. The **eapType** member of the **EAP_METHOD_TYPE** structure is an EAP_TYPE structure that contains the type and vendor identification information for an EAP method.

For more information on the allocation of EAP method types, see section 6.2 of RFC 3748 published by the IETF.

On Windows 7, Windows Server 2008 R2, and later, the **WlanSetProfileEapUserData** function is enhanced. EAP user credentials can be set for all users of a profile if the *dwFlags* parameter contains **WLAN_SET_EAPHOST_DATA_ALL_USERS**. The EAP user credentials on a profile can also be deleted. To delete the EAP user credentials on a profile, the *pbEapUserData* parameter must be **NULL**, the *dwFlags* parameter must equal **WLAN_SET_EAPHOST_DATA_ALL_USERS**, and the *dwEapUserDataSize* parameter must be 0.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfileEapUserData** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

EAP_METHOD_TYPE

EAP_TYPE

WlanGetProfile

WlanGetProfileCustomUserData

WlanGetProfileList

WlanSetProfile

WlanSetProfileEapXmlUserData

WlanSetProfileEapXmlUserData function (wlanapi.h)

7/1/2021 • 4 minutes to read • Edit Online

The **WlanSetProfileEapXmlUserData** function sets the Extensible Authentication Protocol (EAP) user credentials as specified by an XML string. The user credentials apply to a profile on an adapter. These credentials can be used only by the caller.

Syntax

```
DWORD WlanSetProfileEapXmlUserData(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  DWORD      dwFlags,
  LPCWSTR    strEapXmlUserData,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strProfileName`

The name of the profile associated with the EAP user data. Profile names are case-sensitive. This string must be NULL-terminated.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The supplied name must match the profile name derived automatically from the SSID of the network. For an infrastructure network profile, the SSID must be supplied for the profile name. For an ad hoc network profile, the supplied name must be the SSID of the ad hoc network followed by `-adhoc`.

`dwFlags`

A set of flags that modify the behavior of the function.

On Wireless LAN API for Windows XP with SP2, Windows XP with SP3,Windows Vista, and Windows Server 2008, this parameter is reserved and should be set to zero.

On Windows 7, Windows Server 2008 R2, and later, this parameter can be one of the following values.

VALUE	MEANING
WLAN_SET_EAPHOST_DATA_ALL_USERS 0x00000001	Set EAP host data for all users of this profile.

`strEapXmlUserData`

A pointer to XML data used to set the user credentials.

The XML data must be based on the EAPHost User Credentials schema. To view sample user credential XML data, see EAPHost User Properties.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_ACCESS_DENIED	Access is denied. This value is returned if the caller does not have write access to the profile.
ERROR_BAD_PROFILE	The network connection profile is corrupted. This error is returned if the profile specified in the *strProfileName* parameter could not be parsed.
ERROR_INVALID_PARAMETER	A parameter is incorrect. This value is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *pInterfaceGuid* is **NULL**. • *strProfileName* is **NULL**. • *strEapXmlUserData* is **NULL**. • *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle *hClientHandle* was not found in the handle table.
ERROR_NOT_ENOUGH_MEMORY	Not enough storage is available to process this command.
ERROR_NOT_SUPPORTED	The request is not supported. This value is returned when profile settings do not permit storage of user data. This can occur when single signon (SSO) is enabled. On Windows 7, Windows Server 2008 R2 , and later, this value is returned if the WlanSetProfileEapXmlUserData function was called on a profile that uses a method other than 802.1X for authentication.
ERROR_SERVICE_NOT_ACTIVE	The service has not been started. This value is returned if the Wireless LAN service is not running.
RPC_STATUS	Various error codes.

Remarks

The **WlanSetProfileEapXmlUserData** function sets the EAP user credentials to use on a profile. This function can be called only on a profile that uses 802.1X for authentication. On Windows Vista and Windows Server 2008, these credentials can only be used by the caller.

The *eapType* parameter is an EAP_METHOD_TYPE structure that contains type, identification, and author information about an EAP method. The **eapType** member of the **EAP_METHOD_TYPE** structure is an EAP_TYPE structure that contains the type and vendor identification information for an EAP method.

For more information on the allocation of EAP method types, see section 6.2 of RFC 3748 published by the IETF.

On Windows 10, Windows Server 2016, and later, the **WlanSetProfileEapXmlUserData** function is enhanced. EAP user credentials can be set for all users of a profile if the *dwFlags* parameter contains **WLAN_SET_EAPHOST_DATA_ALL_USERS**.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfileEapXmlUserData** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

The **WlanSetProfileEapXmlUserData** might cause wireless connection failure when you use **EAP-TTLS** and the API is called from a 32-bit application running on a 64-bit operating system (OS). Your application should be built for the same CPU architecture as the target OS.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: This function can only be used for Protected EAP (PEAP) credentials. It can't be used for other EAP types.

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

EAP_METHOD_TYPE

EAP_TYPE

WlanGetProfile

WlanGetProfileCustomUserData

WlanGetProfileList

WlanSetProfile

WlanSetProfileCustomUserData

WlanSetProfileEapUserData

WlanSetProfileList function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The **WlanSetProfileList** function sets the preference order of profiles for a given interface.

Syntax

```
DWORD WlanSetProfileList(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  DWORD      dwItems,
  LPCWSTR    *strProfileNames,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`dwItems`

The number of profiles in the *strProfileNames* parameter.

`strProfileNames`

The names of the profiles in the desired order. Profile names are case-sensitive. This string must be NULL-terminated.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The supplied names must match the profile names derived automatically from the SSID of the network. For infrastructure network profiles, the SSID must be supplied for the profile name. For ad hoc network profiles, the supplied name must be the SSID of the ad hoc network followed by `-adhoc`.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to change the profile list. Before WlanSetProfileList performs an operation that changes the relative order of all-user profiles in the profile list or moves an all-user profile to a lower position in the profile list, **WlanSetProfileList** retrieves the discretionary access control list (DACL) stored with the **wlan_secure_all_user_profiles_order** object. If the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanSetProfileList** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_INVALID_PARAMETER	One of the following conditions occurred: • *hClientHandle* is **NULL** or invalid. • *pInterfaceGuid* is **NULL**. • *dwItems* is 0. • *strProfileNames* is **NULL**. • The same profile name appears more than once in *strProfileNames*. • *pReserved* is not **NULL**.
ERROR_NOT_FOUND	*strProfileNames* contains the name of a profile that is not present in the profile store.
RPC_STATUS	Various error codes.

Remarks

The **WlanSetProfileList** function sets the preference order of wireless LAN profiles for a given wireless interface.

The profiles in the list must be a one-to-one match with the current profiles returned by the WlanGetProfileList function. The position of group policy profiles cannot be changed.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfileList** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanGetProfile

WlanGetProfileList

WlanSetProfile

WlanSetProfilePosition function (wlanapi.h)

7/1/2021 • 3 minutes to read • Edit Online

The **WlanSetProfilePosition** function sets the position of a single, specified profile in the preference list.

Syntax

```
DWORD WlanSetProfilePosition(
  HANDLE     hClientHandle,
  const GUID *pInterfaceGuid,
  LPCWSTR    strProfileName,
  DWORD      dwPosition,
  PVOID      pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`pInterfaceGuid`

The GUID of the interface.

`strProfileName`

The name of the profile. Profile names are case-sensitive. This string must be NULL-terminated.

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: The supplied name must match the profile name derived automatically from the SSID of the network. For an infrastructure network profile, the SSID must be supplied for the profile name. For an ad hoc network profile, the supplied name must be the SSID of the ad hoc network followed by `-adhoc`.

`dwPosition`

Indicates the position in the preference list that the profile should be shifted to. 0 (zero) corresponds to the first profile in the list that is returned by the WlanGetProfileList function.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION

ERROR_ACCESS_DENIED	The caller does not have sufficient permissions to change the profile position. Before WlanSetProfilePosition performs an operation that changes the relative order of all-user profiles in the profile list or moves an all-user profile to a lower position in the profile list, **WlanSetProfilePosition** retrieves the discretionary access control list (DACL) stored with the **wlan_secure_all_user_profiles_order** object. If the DACL does not contain an access control entry (ACE) that grants WLAN_WRITE_ACCESS permission to the access token of the calling thread, then **WlanSetProfilePosition** returns **ERROR_ACCESS_DENIED**.
ERROR_INVALID_PARAMETER	*hClientHandle* is **NULL** or invalid, *pInterfaceGuid* is **NULL**, *strProfileName* is **NULL**, or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
RPC_STATUS	Various error codes.

Remarks

The position of group policy profiles cannot be changed.

By default, only a user logged on as a member of the Administrators group can change the position of an all-user profile. Call WlanGetSecuritySettings to determine the actual user rights required to change the position of an all-user profile.

To set the profile position at the command line, use the **netsh wlan set profileorder** command. For more information, see Netsh Commands for Wireless Local Area Network (wlan).

Windows XP with SP3 and Wireless LAN API for Windows XP with SP2: Ad hoc profiles appear after the infrastructure profiles in the profile list. If you try to position an ad hoc profile before an infrastructure profile using **WlanSetProfilePosition**, the **WlanSetProfilePosition** call will succeed but the Wireless Zero Configuration service will reorder the profile list such that the ad hoc profile is positioned after all infrastructure network profiles.

Guest profiles, profiles with Wireless Provisioning Service (WPS) authentication, and profiles with Wi-Fi Protected Access-None (WPA-None) authentication are not supported. Any such profile that appears in the preferred profile list has a fixed position in the profile list. That means its position cannot be changed using **WlanSetProfilePosition** and that its position is not affected by position changes of other profiles.

All wireless LAN functions require an interface GUID for the wireless interface when performing profile operations. When a wireless interface is removed, its state is cleared from Wireless LAN Service (WLANSVC) and no profile operations are possible.

The **WlanSetProfilePosition** function can fail with **ERROR_INVALID_PARAMETER** if the wireless interface specified in the *pInterfaceGuid* parameter has been removed from the system (a USB wireless adapter that has been removed, for example).

Requirements

Minimum supported client	Windows Vista, Windows XP with SP3 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll
Redistributable	Wireless LAN API for Windows XP with SP2

See also

WlanGetProfile

WlanGetProfileList

WlanSetProfile

WlanSetProfileList

WlanSetPsdIEDataList function (wlanapi.h)

7/1/2021 • 6 minutes to read • Edit Online

The **WlanSetPsdIeDataList** function sets the proximity service discovery (PSD) information element (IE) data list.

Syntax

```
DWORD WlanSetPsdIEDataList(
  HANDLE                   hClientHandle,
  LPCWSTR                  strFormat,
  const PWLAN_RAW_DATA_LIST pPsdIEDataList,
  PVOID                    pReserved
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`strFormat`

The format of a PSD IE in the PSD IE data list passed in the *pPsdIEDataList* parameter. This is a NULL-terminated URI string that specifies the namespace of the protocol used for discovery.

`pPsdIEDataList`

A pointer to a WLAN_RAW_DATA_LIST structure that contains the PSD IE data list to be set.

`pReserved`

Reserved for future use. Must be set to **NULL**.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if the *hClientHandle* is **NULL** or not valid or *pReserved* is not **NULL**.
ERROR_INVALID_HANDLE	The handle *hClientHandle* was not found in the handle table.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value is returned if the function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.

RPC_STATUS	Various error codes.

Remarks

The Proximity Service Discovery Protocol is a Microsoft proprietary protocol that allows a client to discover services in its physical proximity, which is defined by the radio range. The purpose of the Proximity Service Discovery Protocol is to convey service discovery information, such as service advertisements, as part of Beacon frames. Access points (APs) and stations (STAs) that operate in ad hoc mode periodically broadcast beacon frames. The beacon frame can contain single or multiple proprietary information elements that carry discovery information pertaining to the services that the device offers.

A PSD IE is used to transmit compressed information provided by higher-level discovery protocols for the purpose of passive discovery. One such higher-level protocol used for discovery is the WS-Discovery protocol. Any protocol can be used for discovery.

Windows Vista and Windows Server 2008 with the Wireless LAN Service installed support passive discovery for ad hoc clients, ad hoc services, and infrastructure clients. This means an ad hoc service can advertise an available resource or service by transmitting a PSD IE in one or more beacons. There is no guarantee that this beacon is received by an ad hoc or infrastructure client.

Windows 7 and Windows Server 2008 R2 with the Wireless LAN Service installed support passive discovery for ad hoc clients, ad hoc services, and infrastructure clients in the same way as in Windows Vista. In addition, the PSD IE is also supported for the wireless Hosted Network, a software-based wireless access point (AP). Applications on the local computer where the wireless Hosted Network is to be run may use the **WlanSetPsdIeDataList** function to set the PSD IE before starting the wireless Hosted Network. Once set, the PSD IE will be included in the beacon and probe response after the wireless Hosted Network is started.

Each application sending or receiving beacons maintains its own PSD IE data list. The *pPsdIEDataList* parameter points to a list of PSD IEs generated by the application. Each PSD IE has the following format.

FIELD	DESCRIPTION AND VALUE
Element ID (1 byte)	221
Length (1 byte)	The length, in bytes, of Data field plus 8.
OUI (3 bytes)	The Organizational Unique Identifier (OUI) must contain a value of 00-50-F2. This public OUI is registered to Microsoft.
OUI Type (1 byte)	For the Proximity Service Discovery Protocol, the OUI Type must contain a value of 6.
Format identifier hash(4 bytes)	Bits 31-0 of the HMAC computed from the *strFormat* parameter.
Data (variable)	Contains user-defined data for discovery. This field must not exceed 240 bytes in length.

Element ID 221 specifies the Vendor-Specific information element defined in the IEEE 802.11 standards. The Organizational Unique Identifier (OUI) contains a 3-byte, IEEE-assigned OUI of the vendor that defined the content of the information element in the same order that the OUI would be transmitted in an IEEE 802.11 address field. The Element ID, Length, OUI, and OUI Type fields are controlled by the automatic configuration service, while the

application controls the rest of the fields.

The Format identifier hash field describes the format of the information carried in the PSD IE. To ensure uniqueness while circumventing the need for central administration of format identifiers, a string in the form of a Uniform Resource Identifier (URI), as specified in RFC 3986, is used to distinguish the format. However, because the transmission must be efficient and space in the information element is limited, the string is not actually transmitted, but, instead, its hash is transmitted. On the client, which is the receiving side of the beacon, the hash is matched against a known set of format identifiers.

The Format identifier hash field is represented by bits 0...31 of a hash-based message authentication code (HMAC) over the format identifier string specified in the *strFormat* parameter. The HMAC is used to specify the format of the Data field of the PSD IE. The formula used to calculate the HMAC is described in RFC 2104. Sample code for the calculation of the HMAC is as specified in RFC 4634. When calculating the HMAC, use SHA-256 for the hash function. The key used is the "null" key (**NULL** pointer to the authentication key, and zero length authentication key per the source code in RFC 4634). Use the value of *strFormat* parameter (including any spaces but excluding the NULL-termination character) as the input text encoded as Unicode UTF-16 in little-endian format.

For example, if the *strFormat* parameter is `http://schemas.xmlsoaps.org/ws/2004/10/discovery`, then the first four octets of the corresponding HMAC is `0xF8 0xCB 0x35 0x15`.

If the *strFormat* parameter is `http://schemas.microsoft.com/networking/discoveryformat/v2`, then the four octets of the corresponding HMAC are `0xCF 0xF1 0x64 0x17`.

When sending the first 4 octets of an HMAC over the network, send the first (left-most) octet first.

Note that there may be collisions in the truncated HMACs, which means that it may be impossible to uniquely determine the discovery protocol corresponding to the payload of a PSD IE from the given bits of an HMAC. An application receiving a PSD IE must take a best guess at the discovery protocol used from a given HMAC, then re-run the higher-level discovery protocol once a connection has been established.

At most, five PSD IEs can be passed in a list. Also, the total length, in bytes, of the PSD IE list may be restricted by hardware limitations on the length of a beacon.

An application can call **WlanSetPsdIeDataList** many times. When **WlanSetPsdIeDataList** is called twice with the same *strFormat*, the contents of the WLAN_RAW_DATA_LIST populated by the first function call are overwritten by the second call's **WLAN_RAW_DATA_LIST** payload. When **WlanSetPsdIeDataList** is called with the *pPsdIEDataList* parameter set to **NULL**, the PSD IE list associated with *strFormat* is cleared. When **WlanSetPsdIeDataList** is called with both the *pPsdIEDataList* and *strFormat* parameters set to **NULL**, all PSD IE lists set by the application are cleared.

The wireless service processes PSD IE data lists set by different applications and generates raw IE data blobs. When a machine creates or joins an ad-hoc network on any wireless adapter, it sends beacons that include a PSD IE data blob associated with the network to other machines.

Stations can call WlanExtractPsdIEDataList function to get the PSD IE data list after receiving a beacon from a machine.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

About the Wireless Hosted Network

WLAN_RAW_DATA_LIST

WlanExtractPsdIEDataList

WlanScan

WlanSetSecuritySettings function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

The WlanGetProfileList function sets the security settings for a configurable object.

Syntax

```
DWORD WlanSetSecuritySettings(
  HANDLE                hClientHandle,
  WLAN_SECURABLE_OBJECT SecurableObject,
  LPCWSTR               strModifiedSDDL
);
```

Parameters

`hClientHandle`

The client's session handle, obtained by a previous call to the WlanOpenHandle function.

`SecurableObject`

A WLAN_SECURABLE_OBJECT value that specifies the object to which the security settings will be applied.

`strModifiedSDDL`

A security descriptor string that specifies the new security settings for the object. This string must be NULL-terminated. For more information, see the Remarks section.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	A parameter is incorrect. This error is returned if any of the following conditions occur: • *hClientHandle* is **NULL**. • *strModifiedSDDL* is **NULL**. • *SecurableObject* is set to a value greater than or equal to **WLAN_SECURABLE_OBJECT_COUNT** (12).
ERROR_INVALID_HANDLE	A handle is invalid. This error is returned if the handle specified in the *hClientHandle* parameter was not found in the handle table.
ERROR_ACCESS_DENIED	The caller does not have sufficient permissions.

ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.

Remarks

A successful call to the **WlanSetSecuritySettings** function overrides the default permissions associated with an object. For more information about default permissions, see Native Wifi API Permissions.

The following describes the procedure for creating a security descriptor object and parsing it as a string.

1. Call InitializeSecurityDescriptor to create a security descriptor in memory.
2. Call SetSecurityDescriptorOwner to set the owner information for the security descriptor.
3. Call InitializeAcl to create a discretionary access control list (DACL) in memory.
4. Call AddAccessAllowedAce or AddAccessDeniedAce to add access control entries (ACEs) to the DACL. Set the *AccessMask* parameter to one of the following bitwise OR combinations as appropriate:
   - WLAN_READ_ACCESS
   - WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS
   - WLAN_READ_ACCESS | WLAN_EXECUTE_ACCESS | WLAN_WRITE_ACCESS
5. Call SetSecurityDescriptorDacl to add the DACL to the security descriptor.
6. Call ConvertSecurityDescriptorToStringSecurityDescriptor to convert the descriptor to string.

The string returned by ConvertSecurityDescriptorToStringSecurityDescriptor can then be used as the *strModifiedSDDL* parameter value when calling **WlanSetSecuritySettings**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanapi.lib
DLL	Wlanapi.dll

See also

Native Wifi API Permissions

WlanGetSecuritySettings

WlanUIEditProfile function (wlanapi.h)

7/1/2021 • 2 minutes to read • Edit Online

Displays the wireless profile user interface (UI). This UI is used to view and edit advanced settings of a wireless network profile.

Syntax

```
DWORD WlanUIEditProfile(
  DWORD             dwClientVersion,
  LPCWSTR           wstrProfileName,
  GUID              *pInterfaceGuid,
  HWND              hWnd,
  WL_DISPLAY_PAGES  wlStartPage,
  PVOID             pReserved,
  PWLAN_REASON_CODE pWlanReasonCode
);
```

Parameters

`dwClientVersion`

Specifies the highest version of the WLAN API that the client supports. Values other than WLAN_UI_API_VERSION will be ignored.

`wstrProfileName`

Contains the name of the profile to be viewed or edited. Profile names are case-sensitive. This string must be NULL-terminated.

The supplied profile must be present on the interface *pInterfaceGuid*. That means the profile must have been previously created and saved in the profile store and that the profile must be valid for the supplied interface.

`pInterfaceGuid`

The GUID of the interface.

`hWnd`

The handle of the application window requesting the UI display.

`wlStartPage`

A WL_DISPLAY_PAGES value that specifies the active tab when the UI dialog box appears.

`pReserved`

Reserved for future use. Must be set to **NULL**.

`pWlanReasonCode`

A pointer to a WLAN_REASON_CODE value that indicates why the UI display failed.

Return value

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value may be one of the following return codes.

RETURN CODE	DESCRIPTION
ERROR_INVALID_PARAMETER	One of the supplied parameters is not valid.
ERROR_NOT_SUPPORTED	This function was called from an unsupported platform. This value will be returned if this function was called from a Windows XP with SP3 or Wireless LAN API for Windows XP with SP2 client.
RPC_STATUS	Various error codes.

Remarks

If **WlanUIEditProfile** returns ERROR_SUCCESS, any changes to the profile made in the UI will be saved in the profile store.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	wlanapi.h (include Wlanapi.h)
Library	Wlanui.lib
DLL	Wlanui.dll