



Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of Atmel may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

Here is a list of all modules:

[detail level [1](#) [2](#) [3](#) [4](#)]

▼ SSL	
Enumeration/Typedefs	
Functions	
▼ WLAN	
▶ Defines	
▶ Function	
DataTypes	
▼ BSP	
Defines	
DataTypes	
▶ Function	
▼ Socket	
▶ Defines	
Error Codes	
▶ DataTypes	
▶ Function	

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by 1.8.13



SSL

Modules

Modules

[Enumeration/Typedefs](#)

[Functions](#)

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

TypeDefs

Enumeration/TypeDefs

SSL

TypeDefs

```
typedef void(* tpfAppSSLCb) (uint8 u8MsgType, void *pvMsg)
```

Detailed Description

TypeDef Documentation

◆ tpfAppSSLCb

```
typedef void(* tpfAppSSLCb) (uint8 u8MsgType, void *pvMsg)
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by 1.8.13



Functions

SSL

Functions

Functions

NMI_API sint8 m2m_ssl_init (tpfAppSSLCb pfAppSSLCb)

NMI_API sint8 m2m_ssl_handshake_rsp (tstrEccReqInfo *strECCResp, uint8 *pu8RspDataBuff, uint16 u16RspDataSz)

NMI_API sint8 m2m_ssl_send_certs_to_winc (uint8 *pu8Buffer, uint32 u32BufferSz)

NMI_API sint8 m2m_ssl_retrieve_cert (uint16 *pu16CurveType, uint8 *pu8Hash, uint8 *pu8Sig, tstrECPoint *pu8Key)

NMI_API sint8 m2m_ssl_retrieve_hash (uint8 *pu8Hash, uint16 u16HashSz)

NMI_API void m2m_ssl_stop_processing_certs (void)

NMI_API void m2m_ssl_ecc_process_done (void)

sint8 m2m_ssl_set_active_ciphersuites (uint32 u32SslCsBMP)

Detailed Description

Function Documentation

- ◆ m2m_ssl_init()

```
NMI_API sint8 m2m_ssl_init ( tpfAppSSLCb pfAppSSLCb )
```

- ◆ m2m_ssl_handshake_rsp()

```
NMI_API sint8 m2m_ssl_handshake_rsp ( tstrEccReqInfo * strECCResp,  
                                         uint8 * pu8RspDataBuff,  
                                         uint16 u16RspDataSz  
                                       )
```

Sends ECC responses to the WINC.

Parameters

[in] **strECCResp** ECC Response struct.
[in] **pu8RspDataBuff** Pointer of the response data to be sent.
[in] **u16RspDataSz** Response data size.

Returns

The function SHALL return 0 for success and a negative value otherwise.

- ◆ m2m_ssl_send_certs_to_winc()

```
NMI_API sint8 m2m_ssl_send_certs_to_winc ( uint8 * pu8Buffer,  
                                            uint32 u32BufferSz  
                                          )
```

Sends certificates to the WINC.

Parameters

[in] **pu8Buffer** Pointer to the certificates.
[in] **u32BufferSz** Size of the certificates.

Returns

The function SHALL return 0 for success and a negative value otherwise.

- ◆ m2m_ssl_retrieve_cert()

```
NMI_API sint8 m2m_ssl_retrieve_cert ( uint16 * pu16CurveType,  
                                     uint8 * pu8Hash,  
                                     uint8 * pu8Sig,  
                                     tstrECPoint * pu8Key  
                                     )
```

Retrieve the certificate to be verified from the WINC.

Parameters

[in] **pu16CurveType** Pointer to the certificate curve type.
[in] **pu8Hash** Pointer to the certificate hash.
[in] **pu8Sig** Pointer to the certificate signature.
[in] **pu8Key** Pointer to the certificate Key.

Returns

The function SHALL return 0 for success and a negative value otherwise.

◆ m2m_ssl_retrieve_hash()

```
NMI_API sint8 m2m_ssl_retrieve_hash ( uint8 * pu8Hash,  
                                       uint16 u16HashSz  
                                       )
```

Retrieve the certificate hash.

Parameters

[in] **pu8Hash** Pointer to the certificate hash.
[in] **u16HashSz** Hash size.

Returns

The function SHALL return 0 for success and a negative value otherwise.

◆ m2m_ssl_stop_processing_certs()

```
NMI_API void m2m_ssl_stop_processing_certs ( void )
```

Allow ssl driver to tidy up in case application does not read all available certificates.

Warning

This API must only be called if some certificates are left unread.

Returns

None.

◆ m2m_ssl_ecc_process_done()

```
NMI_API void m2m_ssl_ecc_process_done ( void )
```

Allow ssl driver to tidy up after application has finished processing ecc message.

Warning

This API must be called after receiving a SSL callback with type **M2M_SSL_REQ_ECC**

Returns

None.

◆ m2m_ssl_set_active_ciphersuites()

```
NMI_API sint8 m2m_ssl_set_active_ciphersuites ( uint32 u32SslCsBMP )
```

Override the default Active SSL ciphers in the SSL module with a certain combination selected by the caller in the form of a bitmap containing the required ciphers to be on. There is no need to call this function if the application will not change the default ciphersuites.

Parameters

[in] **u32SslCsBMP** Bitmap containing the desired ciphers to be enabled for the SSL module. The ciphersuites are defined in [TLS Cipher Suite IDs](#). The default ciphersuites are all ciphersuites supported by the firmware with the exception of ECC ciphersuites. The caller can override the default with any desired combination, except for combinations involving both RSA and ECC; if any RSA ciphersuite is enabled, then firmware will disable all ECC ciphersuites. If u32SslCsBMP does not contain any ciphersuites supported by firmware, then the current active list will not be changed.

Returns

- [SOCK_ERR_NO_ERROR](#)
- [SOCK_ERR_INVALID_ARG](#)



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

WLAN

Modules

Defines

Function

DataTypes

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

doxygen 1.8.13



Defines

WLAN

[Modules](#) | [Macros](#) | [Enumerations](#)

Modules

[CommonDefines](#)

Macros

```
#define M2M_MAJOR_SHIFT (8)

#define M2M_MINOR_SHIFT (4)

#define M2M_PATCH_SHIFT (0)

#define M2M_DRV_VERSION_SHIFT (16)

#define M2M_FW_VERSION_SHIFT (0)

#define M2M_GET_MAJOR(ver_info_hword) ((uint8)((ver_info_hword) >> M2M_MAJOR_SHIFT))

#define M2M_GET_MINOR(ver_info_hword) ((uint8)((ver_info_hword) >> M2M_MINOR_SHIFT))

#define M2M_GET_PATCH(ver_info_hword) ((uint8)((ver_info_hword) >> M2M_PATCH_SHIFT))

#define M2M_GET_FW_VER(ver_info_word) ((uint16)((ver_info_word) >> M2M_FW_VERSION_SHIFT))

#define M2M_GET_DRV_VER(ver_info_word) ((uint16)((ver_info_word) >> M2M_DRV_VERSION_SHIFT))

#define M2M_GET_DRV_MAJOR(ver_info_word) M2M_GET_MAJOR(M2M_GET_DRV_VER(ver_info_word))

#define M2M_GET_DRV_MINOR(ver_info_word) M2M_GET_MINOR(M2M_GET_DRV_VER(ver_info_word))
```

```

#define M2M_GET_DRV_PATCH(ver_info_word) M2M_GET_PATCH(M2M_GET_DRV_VER(ver_info_word))

#define M2M_GET_FW_MAJOR(ver_info_word) M2M_GET_MAJOR(M2M_GET_FW_VER(ver_info_word))

#define M2M_GET_FW_MINOR(ver_info_word) M2M_GET_MINOR(M2M_GET_FW_VER(ver_info_word))

#define M2M_GET_FW_PATCH(ver_info_word) M2M_GET_PATCH(M2M_GET_FW_VER(ver_info_word))

#define M2M_MAKE_VERSION(major, minor, patch)

#define M2M_MAKE_VERSION_INFO(fw_major, fw_minor, fw_patch, drv_major, drv_minor, drv_patch)
    M2M_MAKE_VERSION_INFO(19,5,2,19,3,0)

#define REL_19_5_1_VER M2M_MAKE_VERSION_INFO(19,5,1,19,3,0)

#define REL_19_5_0_VER M2M_MAKE_VERSION_INFO(19,5,0,19,3,0)

#define REL_19_4_6_VER M2M_MAKE_VERSION_INFO(19,4,6,19,3,0)

#define REL_19_4_5_VER M2M_MAKE_VERSION_INFO(19,4,5,19,3,0)

#define REL_19_4_4_VER M2M_MAKE_VERSION_INFO(19,4,4,19,3,0)

#define REL_19_4_3_VER M2M_MAKE_VERSION_INFO(19,4,3,19,3,0)

#define REL_19_4_2_VER M2M_MAKE_VERSION_INFO(19,4,2,19,3,0)

#define REL_19_4_1_VER M2M_MAKE_VERSION_INFO(19,4,1,19,3,0)

#define REL_19_4_0_VER M2M_MAKE_VERSION_INFO(19,4,0,19,3,0)

#define REL_19_3_1_VER M2M_MAKE_VERSION_INFO(19,3,1,19,3,0)

#define REL_19_3_0_VER M2M_MAKE_VERSION_INFO(19,3,0,19,3,0)

#define REL_19_2_2_VER M2M_MAKE_VERSION_INFO(19,2,2,19,2,0)

#define REL_19_2_1_VER M2M_MAKE_VERSION_INFO(19,2,1,19,2,0)

#define REL_19_2_0_VER M2M_MAKE_VERSION_INFO(19,2,0,19,2,0)

#define REL_19_1_0_VER M2M_MAKE_VERSION_INFO(19,1,0,18,2,0)

#define REL_19_0_0_VER M2M_MAKE_VERSION_INFO(19,0,0,18,1,1)

#define M2M_RELEASE_VERSION_MAJOR_NO (19)

```

```
#define M2M_RELEASE_VERSION_MINOR_NO (5)

#define M2M_RELEASE_VERSION_PATCH_NO (2)

#define M2M_RELEASE_VERSION SVN_VERSION (SVN_REVISION)

#define M2M_MIN_REQ_DRV_VERSION_MAJOR_NO (19)

#define M2M_MIN_REQ_DRV_VERSION_MINOR_NO (3)

#define M2M_MIN_REQ_DRV_VERSION_PATCH_NO (0)

#define M2M_MIN_REQ_DRV SVN_VERSION (0)

#define M2M_BUFFER_MAX_SIZE (1600UL - 4)

#define M2M_MAC_ADDRES_LEN 6

#define M2M_ETHERNET_HDR_OFFSET 34

#define M2M_ETHERNET_HDR_LEN 14

#define M2M_MAX_SSID_LEN 33

#define M2M_MAX_PSK_LEN 65

#define M2M_MIN_PSK_LEN 9

#define M2M_DEVICE_NAME_MAX 48

#define M2M_LISTEN_INTERVAL 1

#define MAX_HIDDEN_SITES 4

#define M2M_1X_USR_NAME_MAX 21

#define M2M_1X_PWD_MAX 41

#define M2M_CUST_IE_LEN_MAX 252

#define PWR_DEFAULT PWR_HIGH

#define M2M_CONFIG_CMD_BASE 1

#define M2M_STA_CMD_BASE 40

#define M2M_AP_CMD_BASE 70
```

```
#define M2M_P2P_CMD_BASE 90

#define M2M_SERVER_CMD_BASE 100

#define M2M_OTA_CMD_BASE 100

#define M2M_CRYPTO_CMD_BASE 1

#define M2M_MAX_GRP_NUM_REQ (127)

#define WEP_40_KEY_STRING_SIZE ((uint8)10)

#define WEP_104_KEY_STRING_SIZE ((uint8)26)

#define WEP_KEY_MAX_INDEX ((uint8)4)

#define M2M_SHA256_CONTEXT_BUFF_LEN (128)

#define M2M_SCAN_DEFAULT_NUM_SLOTS (2)

#define M2M_SCAN_DEFAULT_SLOT_TIME (30)

#define M2M_SCAN_DEFAULT_NUM_PROBE (2)

#define TLS_FILE_NAME_MAX 48

#define TLS_SRV_SEC_MAX_FILES 8

#define TLS_SRV_SEC_START_PATTERN_LEN 8

#define OTA_STATUS_VALID (0x12526285)

#define OTA_STATUS_INVALID (0x23987718)

#define OTA_MAGIC_VALUE (0x1ABCDEF9)

#define M2M_MAGIC_APP (0xef522f61UL)

#define OTA_FORMAT_VER_0 (0) /*Till 19.2.2 format*/

#define OTA_FORMAT_VER_1 (1) /*starting from 19.3.0 CRC is used and sequence number is

#define OTA_SHA256_DIGEST_SIZE (32)

#define TLS_CRL_DATA_MAX_LEN 64

#define TLS_CRL_MAX_ENTRIES 10
```

```
#define TLS_CRL_TYPE_NONE 0

#define TLS_CRL_TYPE_CERT_HASH 1
```

Enumerations

```
enum tenuM2mDefaultConnErrcode { M2M_DEFAULT_CONN_INPROGRESS =
    ((sint8)-23), M2M_DEFAULT_CONN_FAIL,
    M2M_DEFAULT_CONN_SCAN_MISMATCH, M2M_DEFAULT_CONN_EMPTY_LIST }
```

Detailed Description

Macro Definition Documentation

- ◆ **M2M_MAJOR_SHIFT**

```
#define M2M_MAJOR_SHIFT (8)
```

- ◆ **M2M_MINOR_SHIFT**

```
#define M2M_MINOR_SHIFT (4)
```

- ◆ **M2M_PATCH_SHIFT**

```
#define M2M_PATCH_SHIFT (0)
```

- ◆ **M2M_DRV_VERSION_SHIFT**

```
#define M2M_DRV_VERSION_SHIFT (16)
```

- ◆ **M2M_FW_VERSION_SHIFT**

```
#define M2M_FW_VERSION_SHIFT (0)
```

- ◆ M2M_GET_MAJOR

```
#define ((uint8)((ver_info_hword) >>  
M2M_GET_MAJOR     ( ver_info_hword ) M2M_MAJOR_SHIFT) & 0xff)
```

- ◆ M2M_GET_MINOR

```
#define ((uint8)((ver_info_hword) >>  
M2M_GET_MINOR     ( ver_info_hword ) M2M_MINOR_SHIFT) & 0x0f)
```

- ◆ M2M_GET_PATCH

```
#define ((uint8)((ver_info_hword) >>  
M2M_GET_PATCH     ( ver_info_hword ) M2M_PATCH_SHIFT) & 0x0f)
```

- ◆ M2M_GET_FW_VER

```
#define ((uint16) ((ver_info_word) >>  
M2M_GET_FW_VER     ( ver_info_word ) M2M_FW_VERSION_SHIFT))
```

- ◆ M2M_GET_DRV_VER

```
#define ((uint16) ((ver_info_word) >>  
M2M_GET_DRV_VER     ( ver_info_word ) M2M_DRV_VERSION_SHIFT))
```

- ◆ M2M_GET_DRV_MAJOR

```
#define  
M2M_GET_DRV_MAJOR ( ver_info_word ) M2M_GET_MAJOR(M2M_GET_DRV_VER(ver_i
```

- ◆ M2M_GET_DRV_MINOR

```
#define  
M2M_GET_DRV_MINOR ( ver_info_word ) M2M_GET_MINOR(M2M_GET_DRV_VER(ver_in
```

- ◆ M2M_GET_DRV_PATCH

```
#define  
M2M_GET_DRV_PATCH ( ver_info_word ) M2M_GET_PATCH(M2M_GET_DRV_VER(ver_in
```

- ◆ M2M_GET_FW_MAJOR

```
#define  
M2M_GET_FW_MAJOR ( ver_info_word ) M2M_GET_MAJOR(M2M_GET_FW_VER(ver_info
```

- ◆ M2M_GET_FW_MINOR

```
#define  
M2M_GET_FW_MINOR ( ver_info_word ) M2M_GET_MINOR(M2M_GET_FW_VER(ver_info
```

- ◆ M2M_GET_FW_PATCH

```
#define  
M2M_GET_FW_PATCH ( ver_info_word ) M2M_GET_PATCH(M2M_GET_FW_VER(ver_info
```

- ◆ M2M_MAKE_VERSION

```
#define M2M_MAKE_VERSION ( major,  
                           minor,  
                           patch  
                         )
```

Value:

```
( \  
  ((uint16)((major) & 0xff) << M2M_MAJOR_SHIFT) | \  
  ((uint16)((minor) & 0x0f) << M2M_MINOR_SHIFT) | \  
  ((uint16)((patch) & 0x0f) << M2M_PATCH_SHIFT))
```

- ◆ M2M_MAKE_VERSION_INFO

```
#define M2M_MAKE_VERSION_INFO( fw_major,  
                               fw_minor,  
                               fw_patch,  
                               drv_major,  
                               drv_minor,  
                               drv_patch  
)
```

Value:

```
( \  
  ( ((uint32)M2M_MAKE_VERSION((fw_major), (fw_minor), (fw_patch)))  
    << M2M_FW_VERSION_SHIFT) | \  
  ( ((uint32)M2M_MAKE_VERSION((drv_major), (drv_minor), (drv_patch)))  
    << M2M_DRV_VERSION_SHIFT))
```

- ◆ REL_19_5_2_VER

```
#define REL_19_5_2_VER M2M_MAKE_VERSION_INFO(19,5,2,19,3,0)
```

- ◆ REL_19_5_1_VER

```
#define REL_19_5_1_VER M2M_MAKE_VERSION_INFO(19,5,1,19,3,0)
```

- ◆ REL_19_5_0_VER

```
#define REL_19_5_0_VER M2M_MAKE_VERSION_INFO(19,5,0,19,3,0)
```

- ◆ REL_19_4_6_VER

```
#define REL_19_4_6_VER M2M_MAKE_VERSION_INFO(19,4,6,19,3,0)
```

- ◆ REL_19_4_5_VER

```
#define REL_19_4_5_VER M2M_MAKE_VERSION_INFO(19,4,5,19,3,0)
```

- ◆ REL_19_4_4_VER

```
#define REL_19_4_4_VER M2M_MAKE_VERSION_INFO(19,4,4,19,3,0)
```

- ◆ REL_19_4_3_VER

```
#define REL_19_4_3_VER M2M_MAKE_VERSION_INFO(19,4,3,19,3,0)
```

- ◆ REL_19_4_2_VER

```
#define REL_19_4_2_VER M2M_MAKE_VERSION_INFO(19,4,2,19,3,0)
```

- ◆ REL_19_4_1_VER

```
#define REL_19_4_1_VER M2M_MAKE_VERSION_INFO(19,4,1,19,3,0)
```

- ◆ REL_19_4_0_VER

```
#define REL_19_4_0_VER M2M_MAKE_VERSION_INFO(19,4,0,19,3,0)
```

- ◆ REL_19_3_1_VER

```
#define REL_19_3_1_VER M2M_MAKE_VERSION_INFO(19,3,1,19,3,0)
```

- ◆ REL_19_3_0_VER

```
#define REL_19_3_0_VER M2M_MAKE_VERSION_INFO(19,3,0,19,3,0)
```

- ◆ REL_19_2_2_VER

```
#define REL_19_2_2_VER M2M_MAKE_VERSION_INFO(19,2,2,19,2,0)
```

- ◆ REL_19_2_1_VER

```
#define REL_19_2_1_VER M2M_MAKE_VERSION_INFO(19,2,1,19,2,0)
```

- ◆ REL_19_2_0_VER

```
#define REL_19_2_0_VER M2M_MAKE_VERSION_INFO(19,2,0,19,2,0)
```

- ◆ REL_19_1_0_VER

```
#define REL_19_1_0_VER M2M_MAKE_VERSION_INFO(19,1,0,18,2,0)
```

- ◆ REL_19_0_0_VER

```
#define REL_19_0_0_VER M2M_MAKE_VERSION_INFO(19,0,0,18,1,1)
```

- ◆ M2M_RELEASE_VERSION_MAJOR_NO

```
#define M2M_RELEASE_VERSION_MAJOR_NO (19)
```

Firmware Major release version number.

- ◆ M2M_RELEASE_VERSION_MINOR_NO

```
#define M2M_RELEASE_VERSION_MINOR_NO (5)
```

Firmware Minor release version number.

- ◆ M2M_RELEASE_VERSION_PATCH_NO

```
#define M2M_RELEASE_VERSION_PATCH_NO (2)
```

Firmware patch release version number.

- ◆ M2M_RELEASE_VERSION SVN_VERSION

```
#define M2M_RELEASE_VERSION SVN_VERSION (SVN_REVISION)
```

Firmware SVN release version number.

- ◆ M2M_MIN_REQ_DRV_VERSION_MAJOR_NO

```
#define M2M_MIN_REQ_DRV_VERSION_MAJOR_NO (19)
```

Driver Major release version number.

- ◆ M2M_MIN_REQ_DRV_VERSION_MINOR_NO

```
#define M2M_MIN_REQ_DRV_VERSION_MINOR_NO (3)
```

Driver Minor release version number.

- ◆ M2M_MIN_REQ_DRV_VERSION_PATCH_NO

```
#define M2M_MIN_REQ_DRV_VERSION_PATCH_NO (0)
```

Driver patch release version number.

- ◆ M2M_MIN_REQ_DRV SVN_VERSION

```
#define M2M_MIN_REQ_DRV SVN_VERSION (0)
```

Driver svn version.

- ◆ M2M_BUFFER_MAX_SIZE

```
#define M2M_BUFFER_MAX_SIZE (1600UL - 4)
```

Maximum size for the shared packet buffer.

- ◆ **M2M_MAC_ADDRES_LEN**

```
#define M2M_MAC_ADDRES_LEN 6
```

The size fo 802 MAC address.

- ◆ **M2M_ETHERNET_HDR_OFFSET**

```
#define M2M_ETHERNET_HDR_OFFSET 34
```

The offset of the Ethernet header within the WLAN Tx Buffer.

- ◆ **M2M_ETHERNET_HDR_LEN**

```
#define M2M_ETHERNET_HDR_LEN 14
```

Length of the Etherenet header in bytes.

- ◆ **M2M_MAX_SSID_LEN**

```
#define M2M_MAX_SSID_LEN 33
```

Maximum size for the Wi-Fi SSID including the NULL termination.

- ◆ **M2M_MAX_PSK_LEN**

```
#define M2M_MAX_PSK_LEN 65
```

Maximum size for the WPA PSK including the NULL termination.

- ◆ **M2M_MIN_PSK_LEN**

```
#define M2M_MIN_PSK_LEN 9
```

Maximum size for the WPA PSK including the NULL termination.

- ◆ **M2M_DEVICE_NAME_MAX**

```
#define M2M_DEVICE_NAME_MAX 48
```

Maximum Size for the device name including the NULL termination.

- ◆ **M2M_LISTEN_INTERVAL**

```
#define M2M_LISTEN_INTERVAL 1
```

The STA uses the Listen Interval parameter to indicate to the AP how many beacon intervals it shall sleep before it retrieves the queued frames from the AP.

- ◆ **MAX_HIDDEN_SITES**

```
#define MAX_HIDDEN_SITES 4
```

max number of hidden SSID supported by scan request

- ◆ **M2M_1X_USR_NAME_MAX**

```
#define M2M_1X_USR_NAME_MAX 21
```

The maximum size of the user name including the NULL termination. It is used for RADIUS authentication in case of connecting the device to an AP secured with WPA-Enterprise.

- ◆ **M2M_1X_PWD_MAX**

```
#define M2M_1X_PWD_MAX 41
```

The maximum size of the password including the NULL termination. It is used for RADIUS authentication in case of connecting the device to an AP secured with WPA-Enterprise.

- ◆ **M2M_CUST_IE_LEN_MAX**

```
#define M2M_CUST_IE_LEN_MAX 252
```

The maximum size of IE (Information Element).

- ◆ **PWR_DEFAULT**

```
#define PWR_DEFAULT PWR_HIGH
```

- ◆ **M2M_CONFIG_CMD_BASE**

```
#define M2M_CONFIG_CMD_BASE 1
```

The base value of all the host configuration commands opcodes.

- ◆ **M2M_STA_CMD_BASE**

```
#define M2M_STA_CMD_BASE 40
```

The base value of all the station mode host commands opcodes.

- ◆ **M2M_AP_CMD_BASE**

```
#define M2M_AP_CMD_BASE 70
```

The base value of all the Access Point mode host commands opcodes.

- ◆ **M2M_P2P_CMD_BASE**

```
#define M2M_P2P_CMD_BASE 90
```

The base value of all the P2P mode host commands opcodes.

- ◆ **M2M_SERVER_CMD_BASE**

```
#define M2M_SERVER_CMD_BASE 100
```

The base value of all the power save mode host commands codes.

- ◆ **M2M_OTA_CMD_BASE**

```
#define M2M_OTA_CMD_BASE 100
```

The base value of all the OTA mode host commands opcodes. The OTA Have special group so can extended from 1-M2M_MAX_GRP_NUM_REQ

- ◆ **M2M_CRYPTO_CMD_BASE**

```
#define M2M_CRYPTO_CMD_BASE 1
```

The base value of all the crypto mode host commands opcodes. The crypto Have special group so can extended from 1-M2M_MAX_GRP_NUM_REQ

- ◆ **M2M_MAX_GRP_NUM_REQ**

```
#define M2M_MAX_GRP_NUM_REQ (127)
```

max number of request in one group equal to 127 as the last bit reserved for config or data pkt

- ◆ **WEP_40_KEY_STRING_SIZE**

```
#define WEP_40_KEY_STRING_SIZE ((uint8)10)
```

Indicate the wep key size in bytes for 40 bit string passphrase.

- ◆ **WEP_104_KEY_STRING_SIZE**

```
#define WEP_104_KEY_STRING_SIZE ((uint8)26)
```

Indicate the wep key size in bytes for 104 bit string passphrase.

- ◆ **WEP_KEY_MAX_INDEX**

```
#define WEP_KEY_MAX_INDEX ((uint8)4)
```

Indicate the max key index value for WEP authentication

- ◆ **M2M_SHA256_CONTEXT_BUFF_LEN**

```
#define M2M_SHA256_CONTEXT_BUFF_LEN (128)
```

sha256 context size

- ◆ **M2M_SCAN_DEFAULT_NUM_SLOTS**

```
#define M2M_SCAN_DEFAULT_NUM_SLOTS (2)
```

The default. number of scan slots performed by the WINC board.

- ◆ **M2M_SCAN_DEFAULT_SLOT_TIME**

```
#define M2M_SCAN_DEFAULT_SLOT_TIME (30)
```

The default. duration in miliseconds of a scan slots performed by the WINC board.

- ◆ **M2M_SCAN_DEFAULT_NUM_PROBE**

```
#define M2M_SCAN_DEFAULT_NUM_PROBE (2)
```

The default. number of scan slots performed by the WINC board.

- ◆ **TLS_FILE_NAME_MAX**

```
#define TLS_FILE_NAME_MAX 48
```

Maximum length for each TLS certificate file name including null terminator.

- ◆ **TLS_SRV_SEC_MAX_FILES**

```
#define TLS_SRV_SEC_MAX_FILES 8
```

Maximum number of certificates allowed in TLS_SRV section.

- ◆ **TLS_SRV_SEC_START_PATTERN_LEN**

```
#define TLS_SRV_SEC_START_PATTERN_LEN 8
```

Length of certificate struct start pattern.

- ◆ **OTA_STATUS_VALID**

```
#define OTA_STATUS_VALID (0x12526285)
```

Magic value updated in the Control structure in case of ROLLACK image Valid

- ◆ **OTA_STATUS_INVALID**

```
#define OTA_STATUS_INVALID (0x23987718)
```

Magic value updated in the Control structure in case of ROLLACK image InValid

- ◆ **OTA_MAGIC_VALUE**

```
#define OTA_MAGIC_VALUE (0x1ABCDEF9)
```

Magic value set at the beginning of the OTA image header

- ◆ **M2M_MAGIC_APP**

```
#define M2M_MAGIC_APP (0xef522f61UL)
```

Magic value set at the beginning of the Cortus OTA image header

- ◆ **OTA_FORMAT_VER_0**

```
#define OTA_FORMAT_VER_0 (0) /*Till 19.2.2 format*/
```

- ◆ **OTA_FORMAT_VER_1**

```
#define OTA_FORMAT_VER_1 (1) /*starting from 19.3.0 CRC is used and sequence number  
is used*/
```

Control structure format version

- ◆ **OTA_SHA256_DIGEST_SIZE**

```
#define OTA_SHA256_DIGEST_SIZE (32)
```

Sha256 digest size in the OTA image, the sha256 digest is set at the beginning of image before the OTA header

- ◆ **TLS_CRL_DATA_MAX_LEN**

```
#define TLS_CRL_DATA_MAX_LEN 64
```

- ◆ **TLS_CRL_MAX_ENTRIES**

```
#define TLS_CRL_MAX_ENTRIES 10
```

- ◆ **TLS_CRL_TYPE_NONE**

```
#define TLS_CRL_TYPE_NONE 0
```

- ◆ **TLS_CRL_TYPE_CERT_HASH**

```
#define TLS_CRL_TYPE_CERT_HASH 1
```

Enumeration Type Documentation

- ◆ **tenuM2mDefaultConnErrcode**

enum **tenuM2mDefaultConnErrcode**

Enumerator	Description
M2M_DEFAULT_CONN_INPROGRESS	A failure that indicates that a default connection or forced connection is in progress
M2M_DEFAULT_CONN_FAIL	A failure response that indicates that the winc failed to connect to the cached network
M2M_DEFAULT_CONN_SCAN_MISMATCH	A failure response that indicates that no one of the cached networks was found in the scan results, as a result to the function call m2m_default_connect.
M2M_DEFAULT_CONN_EMPTY_LIST	A failure response that indicates an empty network list as a result to the function call m2m_default_connect.



CommonDefines

[WLAN](#) » [Defines](#)

Macros

```
#define M2M_TIME_OUT_DELAY 10000  
  
#define M2M_SUCCESS ((sint8)0)  
  
#define M2M_ERR_SEND ((sint8)-1)  
  
#define M2M_ERR_RCV ((sint8)-2)  
  
#define M2M_ERR_MEM_ALLOC ((sint8)-3)  
  
#define M2M_ERR_TIME_OUT ((sint8)-4)  
  
#define M2M_ERR_INIT ((sint8)-5)  
  
#define M2M_ERR_BUS_FAIL ((sint8)-6)  
  
#define M2M_NOT_YET ((sint8)-7)  
  
#define M2M_ERR_FIRMWARE ((sint8)-8)  
  
#define M2M_SPI_FAIL ((sint8)-9)  
  
#define M2M_ERR_FIRMWARE_bURN ((sint8)-10)  
  
#define M2M_ACK ((sint8)-11)  
  
#define M2M_ERR_FAIL ((sint8)-12)
```

```
#define M2M_ERR_FW_VER_MISMATCH ((sint8)-13)

#define M2M_ERR_SCAN_IN_PROGRESS ((sint8)-14)

#define M2M_ERR_INVALID_ARG ((sint8)-15)

#define M2M_ERR_INVALID ((sint8)-16)

#define I2C_ERR_LARGE_ADDRESS 0xE1UL /*the address exceed the max addressing mode in i2c flash*/

#define I2C_ERR_TX_ABRT 0xE2UL /*NO ACK from slave*/

#define I2C_ERR_OVER_SIZE 0xE3UL

#define ERR_PREFIX_NMIS 0xE4UL /*wrong first four byte in flash NMIS*/

#define ERR_FIRMWARE_EXCEED_SIZE 0xE5UL /*Total size of firmware exceed the max size 256k*/

#define PROGRAM_START 0x26961735UL

#define BOOT_SUCCESS 0x10add09eUL

#define BOOT_START 0x12345678UL

#define NBIT31 (0x80000000)

#define NBIT30 (0x40000000)

#define NBIT29 (0x20000000)

#define NBIT28 (0x10000000)

#define NBIT27 (0x08000000)

#define NBIT26 (0x04000000)

#define NBIT25 (0x02000000)

#define NBIT24 (0x01000000)

#define NBIT23 (0x00800000)

#define NBIT22 (0x00400000)

#define NBIT21 (0x00200000)
```

```

#define NBIT20 (0x00100000)

#define NBIT19 (0x00080000)

#define NBIT18 (0x00040000)

#define NBIT17 (0x00020000)

#define NBIT16 (0x00010000)

#define NBIT15 (0x00008000)

#define NBIT14 (0x00004000)

#define NBIT13 (0x00002000)

#define NBIT12 (0x00001000)

#define NBIT11 (0x00000800)

#define NBIT10 (0x00000400)

#define NBIT9 (0x00000200)

#define NBIT8 (0x00000100)

#define NBIT7 (0x00000080)

#define NBIT6 (0x00000040)

#define NBIT5 (0x00000020)

#define NBIT4 (0x00000010)

#define NBIT3 (0x00000008)

#define NBIT2 (0x00000004)

#define NBIT1 (0x00000002)

#define NBIT0 (0x00000001)

#define M2M_MAX(A, B) ((A) > (B) ? (A) : (B))

#define M2M_SEL(x, m1, m2, m3) ((x>1)?((x>2)?(m3):(m2)):(m1))

#define WORD_ALIGN(val) (((val) & 0x03) ? ((val) + 4 - ((val) & 0x03)) : (val))

```

```
#define DATA_PKT_OFFSET 4

#define BYTE_0(word) ((uint8)((word) >> 0 ) & 0x000000FFUL)

#define BYTE_1(word) ((uint8)((word) >> 8 ) & 0x000000FFUL)

#define BYTE_2(word) ((uint8)((word) >> 16) & 0x000000FFUL)

#define BYTE_3(word) ((uint8)((word) >> 24) & 0x000000FFUL)
```

Detailed Description

Macro Definition Documentation

- ◆ M2M_TIME_OUT_DELAY

```
#define M2M_TIME_OUT_DELAY 10000
```

- ◆ M2M_SUCCESS

```
#define M2M_SUCCESS ((sint8)0)
```

- ◆ M2M_ERR_SEND

```
#define M2M_ERR_SEND ((sint8)-1)
```

- ◆ M2M_ERR_RCV

```
#define M2M_ERR_RCV ((sint8)-2)
```

- ◆ M2M_ERR_MEM_ALLOC

```
#define M2M_ERR_MEM_ALLOC ((sint8)-3)
```

- ◆ M2M_ERR_TIME_OUT

```
#define M2M_ERR_TIME_OUT ((sint8)-4)
```

- ◆ M2M_ERR_INIT

```
#define M2M_ERR_INIT ((sint8)-5)
```

- ◆ M2M_ERR_BUS_FAIL

```
#define M2M_ERR_BUS_FAIL ((sint8)-6)
```

- ◆ M2M_NOT_YET

```
#define M2M_NOT_YET ((sint8)-7)
```

- ◆ M2M_ERR_FIRMWARE

```
#define M2M_ERR_FIRMWARE ((sint8)-8)
```

- ◆ M2M_SPI_FAIL

```
#define M2M_SPI_FAIL ((sint8)-9)
```

- ◆ M2M_ERR_FIRMWARE_bURN

```
#define M2M_ERR_FIRMWARE_bURN ((sint8)-10)
```

- ◆ M2M_ACK

```
#define M2M_ACK ((sint8)-11)
```

- ◆ M2M_ERR_FAIL

```
#define M2M_ERR_FAIL ((sint8)-12)
```

- ◆ M2M_ERR_FW_VER_MISMATCH

```
#define M2M_ERR_FW_VER_MISMATCH ((sint8)-13)
```

- ◆ M2M_ERR_SCAN_IN_PROGRESS

```
#define M2M_ERR_SCAN_IN_PROGRESS ((sint8)-14)
```

- ◆ M2M_ERR_INVALID_ARG

```
#define M2M_ERR_INVALID_ARG ((sint8)-15)
```

- ◆ M2M_ERR_INVALID

```
#define M2M_ERR_INVALID ((sint8)-16)
```

- ◆ I2C_ERR_LARGE_ADDRESS

```
#define I2C_ERR_LARGE_ADDRESS 0xE1UL /*the address exceed the max addressing mode in i2c flash*/
```

- ◆ I2C_ERR_TX_ABRT

```
#define I2C_ERR_TX_ABRT 0xE2UL /*NO ACK from slave*/
```

- ◆ I2C_ERR_OVER_SIZE

```
#define I2C_ERR_OVER_SIZE 0xE3UL
```

- ◆ **ERR_PREFIX_NMIS**

```
#define ERR_PREFIX_NMIS 0xE4UL /*wrong first four byte in flash NMIS*/
```

- ◆ **ERR_FIRMEWARE_EXCEED_SIZE**

```
#define ERR_FIRMEWARE_EXCEED_SIZE 0xE5UL /*Total size of firmware exceed the max size 256k*/
```

- ◆ **PROGRAM_START**

```
#define PROGRAM_START 0x26961735UL
```

- ◆ **BOOT_SUCCESS**

```
#define BOOT_SUCCESS 0x10add09eUL
```

- ◆ **BOOT_START**

```
#define BOOT_START 0x12345678UL
```

- ◆ **NBIT31**

```
#define NBIT31 (0x80000000)
```

- ◆ **NBIT30**

```
#define NBIT30 (0x40000000)
```

- ◆ **NBIT29**

```
#define NBIT29 (0x20000000)
```

◆ NBIT28

```
#define NBIT28 (0x10000000)
```

◆ NBIT27

```
#define NBIT27 (0x08000000)
```

◆ NBIT26

```
#define NBIT26 (0x04000000)
```

◆ NBIT25

```
#define NBIT25 (0x02000000)
```

◆ NBIT24

```
#define NBIT24 (0x01000000)
```

◆ NBIT23

```
#define NBIT23 (0x00800000)
```

◆ NBIT22

```
#define NBIT22 (0x00400000)
```

◆ NBIT21

```
#define NBIT21 (0x00200000)
```

◆ NBIT20

```
#define NBIT20 (0x00100000)
```

◆ NBIT19

```
#define NBIT19 (0x00080000)
```

◆ NBIT18

```
#define NBIT18 (0x00040000)
```

◆ NBIT17

```
#define NBIT17 (0x00020000)
```

◆ NBIT16

```
#define NBIT16 (0x00010000)
```

◆ NBIT15

```
#define NBIT15 (0x00008000)
```

◆ NBIT14

```
#define NBIT14 (0x00004000)
```

◆ NBIT13

```
#define NBIT13 (0x00002000)
```

◆ NBIT12

```
#define NBIT12 (0x00001000)
```

◆ NBIT11

```
#define NBIT11 (0x00000800)
```

◆ NBIT10

```
#define NBIT10 (0x00000400)
```

◆ NBIT9

```
#define NBIT9 (0x00000200)
```

◆ NBIT8

```
#define NBIT8 (0x00000100)
```

◆ NBIT7

```
#define NBIT7 (0x00000080)
```

◆ NBIT6

```
#define NBIT6 (0x00000040)
```

◆ NBIT5

```
#define NBIT5 (0x00000020)
```

◆ NBIT4

```
#define NBIT4 (0x00000010)
```

◆ NBIT3

```
#define NBIT3 (0x00000008)
```

◆ NBIT2

```
#define NBIT2 (0x00000004)
```

◆ NBIT1

```
#define NBIT1 (0x00000002)
```

◆ NBIT0

```
#define NBIT0 (0x00000001)
```

◆ M2M_MAX

```
#define M2M_MAX ( A,  
                  B  
                )    ((A) > (B) ? (A) : (B))
```

◆ M2M_SEL

```
#define M2M_SEL ( x,  
                  m1,  
                  m2,  
                  m3  
                )    ((x>1)?((x>2)?(m3):(m2)):(m1))
```

◆ WORD_ALIGN

```
#define WORD_ALIGN ( val ) (((val) & 0x03) ? ((val) + 4 - ((val) & 0x03)) : (val))
```

◆ DATA_PKT_OFFSET

```
#define DATA_PKT_OFFSET 4
```

◆ BYTE_0

```
#define BYTE_0 ( word ) ((uint8)((word) >> 0) & 0x000000FFUL)
```

◆ BYTE_1

```
#define BYTE_1 ( word ) ((uint8)((word) >> 8) & 0x000000FFUL)
```

◆ BYTE_2

```
#define BYTE_2 ( word ) ((uint8)((word) >> 16) & 0x000000FFUL)
```

◆ BYTE_3

```
#define BYTE_3 ( word ) ((uint8)((word) >> 24) & 0x000000FFUL)
```



Function

WLAN

Modules

Modules

[m2m_ota_init](#)

[m2m_ota_notif_set_url](#)

[m2m_ota_notif_check_for_update](#)

[m2m_ota_notif_sched](#)

[m2m_ota_start_update](#)

[m2m_ota_rollback](#)

[m2m_ota_abort](#)

[m2m_ota_switch_firmware](#)

[m2m_wifi_download_mode](#)

[m2m_wifi_init](#)

[m2m_wifi_deinit](#)

[m2m_wifi_handle_events](#)

[m2m_wifi_send_crl](#)

[m2m_wifi_default_connect](#)

`m2m_wifi_connect`

`m2m_wifi_disconnect`

`m2m_wifi_start_provision_mode`

`m2m_wifi_stop_provision_mode`

`m2m_wifi_get_connection_info`

`m2m_wifi_set_mac_address`

`m2m_wifi_wps`

`m2m_wifi_wps_disable`

`m2m_wifi_p2p`

`m2m_wifi_p2p_disconnect`

`m2m_wifi_enable_ap`

`m2m_wifi_disable_ap`

`m2m_wifi_set_static_ip`

`m2m_wifi_request_dhcp_client`

`m2m_wifi_request_dhcp_server`

`m2m_wifi_enable_dhcp`

`m2m_wifi_set_scan_options`

`m2m_wifi_set_scan_region`

`m2m_wifi_request_scan`

`m2m_wifi_get_num_ap_found`

`m2m_wifi_req_scan_result`

`m2m_wifi_req_curr_rssi`

`m2m_wifi_get_otp_mac_address`

`m2m_wifi_get_mac_address`

`m2m_wifi_set_sleep_mode`

`m2m_wifi_request_sleep`

`m2m_wifi_get_sleep_mode`

`m2m_wifi_req_client_ctrl`

`m2m_wifi_req_server_init`

`m2m_wifi_set_device_name`

`m2m_wifi_set_lsn_int`

`m2m_wifi_enable_monitoring_mode`

`m2m_wifi_disable_monitoring_mode`

`m2m_wifi_send_wlan_pkt`

`m2m_wifi_send_etherenet_pkt`

`m2m_wifi_enable_sntp`

`m2m_wifi_set_system_time`

`m2m_wifi_get_system_time`

`m2m_wifi_set_cust_InfoElement`

`m2m_wifi_set_power_profile`

`m2m_wifi_set_tx_power`

`m2m_wifi_enable_firmware_logs`

`m2m_wifi_set_battery_voltage`

`m2m_wifi_set_gains`

`m2m_wifi_get_firmware_version`

`m2m_wifi_prng_get_random_bytes`

Detailed Description



m2m_ota_init

WLAN » Function

Functions

Functions

**NMI_API sint8 m2m_ota_init (tpfOtaUpdateCb pfOtaUpdateCb, tpfOtaNotifCb
 pfOtaNotifCb)**

Detailed Description

Synchronous initialization function for the OTA layer by registering the update callback. The notification callback is not supported at the current version. Calling this API is a MUST for all the OTA API's.

Function Documentation

◆ m2m_ota_init()

```
NMI_API sint8 m2m_ota_init ( tpfOtaUpdateCb pfOtaUpdateCb,  
                              tpfOtaNotifCb    pfOtaNotifCb  
                              )
```

Parameters

[in] **pfOtaUpdateCb** OTA Update callback function
[in] **pfOtaNotifCb** OTA notify callback function

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_ota_notif_set_url

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_ota_notif_set_url (uint8 *u8Url)

Detailed Description

Set the OTA notification server URL, the functions need to be called before any check for update

Function Documentation

◆ m2m_ota_notif_set_url()

NMI_API sint8 m2m_ota_notif_set_url (uint8 * u8Url)

Parameters

[in] **u8Url** Set the OTA notification server URL, the functions need to be called before any check for update.

Warning

Calling m2m_ota_init is required Notification Server is not supported in the current version (function is not implemented)

See also

m2m_ota_init

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_ota_notif_check_for_update

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_ota_notif_check_for_update (void)

Detailed Description

Synchronous function to check for the OTA update using the Notification Server URL. Function is not implemented (not supported at the current version)

Function Documentation

◆ m2m_ota_notif_check_for_update()

NMI_API sint8 m2m_ota_notif_check_for_update (void)

Warning

Function is not implemented (not supported at the current version)

See also

[m2m_ota_init](#) [m2m_ota_notif_set_url](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value

otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_ota_notif_sched

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_ota_notif_sched (uint32 u32Period)

Detailed Description

Schedule OTA notification Server check for update request after specific number of days

Function Documentation

◆ m2m_ota_notif_sched()

NMI_API sint8 m2m_ota_notif_sched (uint32 u32Period)

Parameters

[in] **u32Period** Period in days

See also

[m2m_ota_init](#) [m2m_ota_notif_check_for_update](#) [m2m_ota_notif_set_url](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_ota_start_update

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_ota_start_update (uint8 *u8DownloadUrl)

NMI_API sint8 m2m_ota_start_update_crt (uint8 *u8DownloadUrl)

Detailed Description

Request OTA start update using the downloaded URL, the OTA module will download the OTA image and ensure integrity of the image, and update the validity of the image in control structure. Switching to that image requires calling [m2m_ota_switch_firmware](#) API. As a prerequisite [m2m_ota_init](#) should be called before using `m2m_ota_start()`.

Request OTA start for cortus application image using the downloaded URL, the OTA module will download the OTA image and ensure integrity of the image, and update the validity of the image in control structure. Switching to that image requires calling [m2m_ota_switch_crt](#) API. As a prerequisite [m2m_ota_init](#) should be called before using [m2m_ota_start_update_crt\(\)](#).

Function Documentation

- ◆ [m2m_ota_start_update\(\)](#)

NMI_API sint8 m2m_ota_start_update (uint8 * u8DownloadUrl)

Parameters

[in] **u8DownloadUrl** The download firmware URL, you get it from device info according to the application server

Warning

Calling this API does not guarantee OTA WINC image update, It depends on the connection with the download server and the validity of the image. If the API response is failure this may invalidate the roll-back image if it was previously valid, since the WINC does not have any internal memory except the flash roll-back image location to validate the downloaded image from

See also

[m2m_ota_init tpfOtaUpdateCb](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The example shows an example of how the OTA image update is carried out.

```
static void OtaUpdateCb(uint8 u80taUpdateStatusType ,uint8
                        u80taUpdateStatus)
{
    if(u80taUpdateStatusType == DL_STATUS) {
        if(u80taUpdateStatus == OTA_STATUS_SUCSESS) {
            //switch to the upgraded firmware
            m2m_ota_switch_firmware();
        }
    }
    else if(u80taUpdateStatusType == SW_STATUS) {
        if(u80taUpdateStatus == OTA_STATUS_SUCSESS) {
            M2M_INFO("Now OTA successfully done");
            //start the host SW upgrade then system reset is required (Reinitialize
            //the driver)
        }
    }
}
void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    case M2M_WIFI_REQ_DHCP_CONF:
    {
        //after successfully connection, start the over air upgrade
        m2m_ota_start_update(OTA_URL);
    }
    break;
    default:
    break;
}
int main (void)
{
    tstrWifiInitParam param;
    tstr1xAuthCredentials gstrCred1x      = AUTH_CREDENTIALS;
    nm_bsp_init();
```

```

m2m_memset((uint8*)&param, 0, sizeof(param));
param.pfAppWifiCb = wifi_event_cb;

//Initialize the WINC Driver
ret = m2m_wifi_init(&param);
if (M2M_SUCCESS != ret)
{
    M2M_ERR("Driver Init Failed <%d>\n", ret);
    while(1);
}
//Initialize the OTA module
m2m_ota_init(OtaUpdateCb, NULL);
//connect to AP that provide connection to the OTA server
m2m_wifi_default_connect();

while(1)
{

//Handle the app state machine plus the WINC event handler
while(m2m_wifi_handle_events(NULL) != M2M_SUCCESS) {

}

}
}

```

◆ m2m_ota_start_update_crt()

NMI_API **sint8** m2m_ota_start_update_crt (**uint8** * **u8DownloadUrl**)

Parameters

[in] **u8DownloadUrl** The cortus application image url.

Warning

Calling this API does not guarantee cortus application image update, It depends on the connection with the download server and the validity of the image. If the API response is failure this may invalidate the roll-back image if it was previously valid, since the WINC does not have any internal memory except the flash roll-back image location to validate the downloaded image from

See also

[m2m_ota_init](#) [tpfOtaUpdateCb](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_ota_rollback

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_ota_rollback (void)

NMI_API sint8 m2m_ota_rollback_crt (void)

Detailed Description

Request OTA Roll-back to the old (other) WINC image, the WINC firmware will check the validation of the Roll-back image and switch to it if it is valid. If the API response is success, system restart is required (re-initialize the driver with hardware rest) update the host driver version may be required if it is did not match the minimum version supported by the WINC firmware.

Request Cortus application OTA Roll-back to the old (other) cortus application image, the WINC firmware will check the validation of the Roll-back image and switch to it if it is valid. If the API response is success, system restart is required (re-initialize the driver with hardware rest) update the host driver version may be required.

Function Documentation

◆ m2m_ota_rollback()

NMI_API sint8 m2m_ota_rollback (void)

See also

[m2m_ota_init](#) [m2m_ota_start_update](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

◆ [m2m_ota_rollback_crt\(\)](#)

NMI_API **sint8** m2m_ota_rollback_crt (void)

See also

[m2m_ota_init](#) [m2m_ota_start_update_crt](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Functions

m2m_ota_abort

WLAN » Function

Functions

NMI_API sint8 m2m_ota_abort (void)

Detailed Description

Request abort of current OTA download. The WINC firmware will terminate the OTA download if one is in progress. If no download is in progress, the API will respond with failure.

Function Documentation

◆ m2m_ota_abort()

NMI_API sint8 m2m_ota_abort (void)

Returns

The function returns **M2M_SUCCESS** for successful operation and a negative value otherwise.



m2m_ota_switch_firmware

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_ota_switch_firmware (void)

NMI_API sint8 m2m_ota_switch_crt (void)

NMI_API sint8 m2m_ota_get_firmware_version (tstrM2mRev *pstrRev)

Detailed Description

Switch to the upgraded Firmware, that API will update the control structure working image to the upgraded image take effect will be on the next system restart

Switch to the upgraded cortus application, that API will update the control structure working image to the upgraded image take effect will be on the next system restart

Function Documentation

♦ m2m_ota_switch_firmware()

NMI_API sint8 m2m_ota_switch_firmware (void)

Warning

It is important to note that if the API succeeds, system restart is required (re-initializing the driver with hardware reset) updating the host driver version may be required if it does not match the minimum driver version supported by the WINC's firmware.

See also

[m2m_ota_init](#) [m2m_ota_start_update](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

Warning

It is important to note that if the API succeeds, system restart is required (re-initializing the driver with hardware reset) updating the host driver version may be required if it does not match the minimum driver version supported by the WINC's firmware.

See also

[m2m_ota_init](#) [m2m_ota_start_update_crt](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

- ◆ [m2m_ota_switch_crt\(\)](#)

```
NMI_API sint8 m2m_ota_switch_crt ( void )
```

- ◆ [m2m_ota_get_firmware_version\(\)](#)

```
NMI_API sint8 m2m_ota_get_firmware_version ( tstrM2mRev * pstrRev )
```



m2m_wifi_download_mode

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_download_mode (void)

Detailed Description

Synchronous download mode entry function that prepares the WINC board to enter the download mode, ready for the firmware or certificate download. The WINC board is prepared for download, through initializations for the WINC driver including bus initializations and interrupt enabling, it also halts the chip, to allow for the firmware downloads. Firmware can be downloaded through a number of interfaces, UART, I2C and SPI.

Function Documentation

◆ m2m_wifi_download_mode()

NMI_API void m2m_wifi_download_mode (void)

Prepares the WINC board before downloading any data (Firmware, Certificates .. etc)

This function should be called before starting to download any data to the WINC board. The WINC board is prepared for download, through initializations for the WINC driver including bus initializations and interrupt enabling, it also halts the chip, to allow for the firmware downloads

Firmware can be downloaded through a number of interfaces, UART, I2C and SPI.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Functions

m2m_wifi_init

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_init (tstrWifiInitParam *pWifiInitParam)

Detailed Description

Synchronous initialization function for the WINC driver. This function initializes the driver by, registering the call back function for M2M_WIFI layer(also the call back function for bypass mode/monitoring mode if defined), initializing the host interface layer and the bus interfaces. Wi-Fi callback registering is essential to allow the handling of the events received, in response to the asynchronous Wi-Fi operations.

Following are the possible Wi-Fi events that are expected to be received through the call back function(provided by the application) to the M2M_WIFI layer are :

```
@ref M2M_WIFI_RESP_CON_STATE_CHANGED \n
@ref M2M_WIFI_RESP_CONN_INFO \n
@ref M2M_WIFI_REQ_DHCP_CONF \n
@ref M2M_WIFI_REQ_WPS \n
@ref M2M_WIFI_RESP_IP_CONFLICT \n
@ref M2M_WIFI_RESP_SCAN_DONE \n
@ref M2M_WIFI_RESP_SCAN_RESULT \n
@ref M2M_WIFI_RESP_CURRENT_RSSI \n
@ref M2M_WIFI_RESP_CLIENT_INFO \n
@ref M2M_WIFI_RESP_PROVISION_INFO \n
@ref M2M_WIFI_RESP_DEFAULT_CONNECT \n
```

Example: \n

In case Bypass mode is defined : \n

```
@ref M2M_WIFI_RESP_ETHERNET_RX_PACKET  
In case Monitoring mode is used: \n  
@ref M2M_WIFI_RESP_WIFI_RX_PACKET  
  
Any application using the WINC driver must call this function at  
the start of its main function.
```

Function Documentation

◆ m2m_wifi_init()

```
NMI_API sint8 m2m_wifi_init ( tstrWifiInitParam * pWifiInitParam )
```

Initialize the WINC host driver. This function initializes the driver by, registering the call back function for M2M_WIFI layer(also the call back function for bypass mode/monitoring mode if defined), initializing the host interface layer and the bus interfaces.

Parameters

[in] **pWifiInitParam** This is a pointer to the **tstrWifiInitParam** structure which holds the pointer to the application WIFI layer call back function, monitoring mode call back and **tstrEthInitParam** structure containing bypass mode parameters.

Precondition

Prior to this function call, The application should initialize the BSP using "nm_bsp_init". Also,application users must provide a call back function responsible for receiving all the WI-FI events that are received on the M2M_WIFI layer.

Warning

Failure to successfully complete function indicates that the driver couldn't be initialized and a fatal error will prevent the application from proceeding.

See also

[nm_bsp_init](#) [m2m_wifi_deinit](#) [tenuM2mStaCmd](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_deinit

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_deinit (void *arg)

Detailed Description

Synchronous de-initialization function to the WINC1500 driver. De-initializes the host interface and frees any resources used by the M2M_WIFI layer. This function must be called in the application closing phase to ensure that all resources have been correctly released. No arguments are expected to be passed in.

Function Documentation

• m2m_wifi_deinit()

NMI_API sint8 m2m_wifi_deinit (void * arg)

Deinitialize the WINC driver and host interface. This function must be called at the De-initialization stage of the application. Generally This function should be the last function before switching off the chip and it should be followed only by "nm_bsp_deinit" function call. Every function call of "nm_wifi_init" should be matched with a call to nm_wifi_deinit.

Parameters

[in] **arg** Generic argument. Not used in the current implementation.

See also

[**nm_bsp_deinit**](#) [**nm_wifi_init**](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by





m2m_wifi_handle_events

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_handle_events (void *arg)

Detailed Description

Synchronous M2M event handler function, responsible for handling interrupts received from the WINC firmware. Application developers should call this function periodically in-order to receive the events that are to be handled by the callback functions implemented by the application.

Function Documentation

◆ m2m_wifi_handle_events()

NMI_API sint8 m2m_wifi_handle_events (void * arg)

Handle the varios events received from the WINC board. Whenever an event happen in the WINC board (e.g. Connection, Disconnection , DHCP .. etc), WINC will interrupt the host to let it know that a new event has occurred. The host driver will attempt to handle these events whenever the host driver decides to do that by calling the "m2m_wifi_handle_events" function. It's mandatory to call this function periodically and independantly of any other condition. It's ideal to include this function in the main and the most frequent loop of the host application.

Precondition

Prior to receiving events, the WINC driver should have been successfully initialized by calling the `m2m_wifi_init` function.

Warning

Failure to successfully complete this function indicates bus errors and hence a fatal error that will prevent the application from proceeding.

Returns

The function returns `M2M_SUCCESS` for successful interrupt handling and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



m2m_wifi_send_crl

WLAN » Function

Functions

Functions

```
sint8 m2m_wifi_send_crl (tstrTlsCrlInfo *pCRL)
```

Detailed Description

Asynchronous API that notifies the WINC with the Certificate Revocation List to be used for TLS.

Function Documentation

◆ m2m_wifi_send_crl()

```
sint8 m2m_wifi_send_crl ( tstrTlsCrlInfo * pCRL )
```

Asynchronous API that notifies the WINC with the Certificate Revocation List.

Parameters

[in] **pCRL** Pointer to the structure containing certificate revocation list details.

Returns

The function returns **M2M_SUCCESS** if the command has been successfully queued to the WINC, and a negative value otherwise.



m2m_wifi_default_connect

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_default_connect (void)

Detailed Description

Asynchronous Wi-Fi connection function. An application calling this function will cause the firmware to correspondingly connect to the last successfully connected AP from the cached connections. A failure to connect will result in a response of **M2M_WIFI_RESP_DEFAULT_CONNECT** indicating the connection error as defined in the structure **tstrM2MDefaultConnResp**. Possible errors are: The connection list is empty **M2M_DEFAULT_CONN_EMPTY_LIST** or a mismatch for the saved AP name **M2M_DEFAULT_CONN_SCAN_MISMATCH**. only difference between this function and **m2m_wifi_connect**, is the connection parameters. Connection using this function is expected to connect using cached connection parameters.

Function Documentation

- ◆ **m2m_wifi_default_connect()**

NMI_API sint8 m2m_wifi_default_connect (void)

Connect to the last successfully connected AP from the cached connections.

Precondition

Prior to connecting, the WINC driver should have been successfully initialized by calling the [m2m_wifi_init](#) function.

Warning

This function must be called in station mode only. It's important to note that successful completion of a call to [m2m_wifi_default_connect\(\)](#) does not guarantee success of the WIFI connection, and a negative return value indicates only locally-detected errors.

See also

[m2m_wifi_connect](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



m2m_wifi_connect

Functions

WLAN » Function

Functions

```
NMI_API sint8 m2m_wifi_connect (char *pcSsid, uint8 u8SsidLen, uint8 u8SecType, void  
*pvAuthInfo, uint16 u16Ch)
```

```
NMI_API sint8 m2m_wifi_connect_sc (char *pcSsid, uint8 u8SsidLen, uint8 u8SecType,  
void *pvAuthInfo, uint16 u16Ch, uint8 u8SaveCred)
```

Detailed Description

Asynchronous wi-fi connection function to a specific AP. Prior to a successful connection, the application must define the SSID of the AP, the security type, the authentication information parameters and the channel number to which the connection will be established. The connection status is known when a response of **M2M_WIFI_RESP_CON_STATE_CHANGED** is received based on the states defined in **tenuM2mConnState**, successful connection is defined by **M2M_WIFI_CONNECTED**

The only difference between this function and **m2m_wifi_default_connect**, is the connection parameters. Connection using this function is expected to be made to a specific AP and to a specified channel.

Asynchronous wi-fi connection function to a specific AP. Prior to a successful connection, the application developers must know the SSID of the AP, the security type, the authentication information parameters and the channel number to which the connection will be established. This API allows the user to choose whether to receive a response when the connection status is known. A response of **M2M_WIFI_RESP_CON_STATE_CHANGED** is received based on the states defined in

tenuM2mConnState, successful connection is defined by **M2M_WIFI_CONNECTED** The only difference between this function and **m2m_wifi_connect**, is the option to save the access point info (SSID, password...etc) or not. Connection using this function is expected to be made to a specific AP and to a specified channel.

Function Documentation

◆ m2m_wifi_connect()

```
NMI_API sint8 m2m_wifi_connect ( char * pcSsid,
                                  uint8 u8SsidLen,
                                  uint8 u8SecType,
                                  void * pvAuthInfo,
                                  uint16 u16Ch
                                )
```

Parameters

- [in] **pcSsid** A buffer holding the SSID corresponding to the requested AP.
- [in] **u8SsidLen** Length of the given SSID (not including the NULL termination). A length less than ZERO or greater than the maximum defined SSID **M2M_MAX_SSID_LEN** will result in a negative error **M2M_ERR_FAIL**.
- [in] **u8SecType** Wi-Fi security type security for the network. It can be one of the following types: -**M2M_WIFI_SEC_OPEN** -**M2M_WIFI_SEC_WEP** -**M2M_WIFI_SEC_WPA_PSK** -**M2M_WIFI_SEC_802_1X** A value outside these possible values will result in a negative return error **M2M_ERR_FAIL**.
- [in] **pvAuthInfo** Authentication parameters required for completing the connection. Its type is based on the Security type. If the authentication parameters are NULL or are greater than the maximum length of the authentication parameters length as defined by **M2M_MAX_PSK_LEN** a negative error will return **M2M_ERR_FAIL**(-12) indicating connection failure.
- [in] **u16Ch** Wi-Fi channel number as defined in **tenuM2mScanCh** enumeration. Channel number greater than **M2M_WIFI_CH_14** returns a negative error **M2M_ERR_FAIL**(-12). Except if the value is **M2M_WIFI_CH_ALL**(255), since this indicates that the firmware should scan all channels to find the SSID requested to connect to. Failure to find the connection match will return a negative error

M2M_DEFAULT_CONN_SCAN_MISMATCH.

Precondition

Prior to a successful connection request, the Wi-Fi driver must have been successfully initialized through the call of the function

See also

tuniM2MWifiAuth tstr1xAuthCredentials tstrM2mWifiWepParams

Warning

-This function must be called in station mode only. -Successful completion of this function does not guarantee success of the WIFI connection, and a negative return value indicates only locally-detected errors.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

◆ **m2m_wifi_connect_sc()**

```
NMI_API sint8 m2m_wifi_connect_sc ( char * pcSsid,
                                     uint8 u8SsidLen,
                                     uint8 u8SecType,
                                     void * pvAuthInfo,
                                     uint16 u16Ch,
                                     uint8 u8SaveCred
                                   )
```

Parameters

[in] pcSsid	A buffer holding the SSID corresponding to the requested AP.
[in] u8SsidLen	Length of the given SSID (not including the NULL termination). A length less than ZERO or greater than the maximum defined SSID M2M_MAX_SSID_LEN will result in a negative error M2M_ERR_FAIL .
[in] u8SecType	Wi-Fi security type security for the network. It can be one of the following types: - M2M_WIFI_SEC_OPEN - M2M_WIFI_SEC_WEP - M2M_WIFI_SEC_WPA_PSK - M2M_WIFI_SEC_802_1X A value outside these possible values will result in a negative return error M2M_ERR_FAIL .
[in] pvAuthInfo	Authentication parameters required for completing the connection. Its type is based on the Security type. If the

authentication parameters are NULL or are greater than the maximum length of the authentication parameters length as defined by **M2M_MAX_PSK_LEN** a negative error will return **M2M_ERR_FAIL**(-12) indicating connection failure.

[in] u16Ch	Wi-Fi channel number as defined in tenuM2mScanCh enumeration. Channel number greater than M2M_WIFI_CH_14 returns a negative error M2M_ERR_FAIL (-12). Except if the value is M2M_WIFI_CH_ALL (255), since this indicates that the firmware should scan all channels to find the SSID requested to connect to. Failure to find the connection match will return a negative error M2M_DEFAULT_CONN_SCAN_MISMATCH .
[in] u8NoSaveCred	Option to store the access point SSID and password into the WINC flash memory or not.

Precondition

Prior to a successful connection request, the wi-fi driver must have been successfully initialized through the call of the function

See also

[tuniM2MWifiAuth](#) [tstr1xAuthCredentials](#) [tstrM2mWifiWepParams](#)

Warning

-This function must be called in station mode only. -Successful completion of this function does not guarantee success of the WIFI connection, and a negative return value indicates only locally-detected errors.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_disconnect

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_disconnect (void)

Detailed Description

Synchronous wi-fi disconnection function, requesting a Wi-Fi disconnection from the currently connected AP.

Function Documentation

♦ m2m_wifi_disconnect()

NMI_API sint8 m2m_wifi_disconnect (void)

Request a Wi-Fi disconnect from the currently connected AP. After the Disconnect is complete the driver should receive a response of **M2M_WIFI_RESP_CON_STATE_CHANGED** based on the states defined in **tenuM2mConnState**, successful disconnection is defined by **M2M_WIFI_DISCONNECTED**.

Precondition

Disconnection request must be made to a successfully connected AP. If the WINC is not in the connected state, a call to this function will hold insignificant.

Warning

This function must be called in station mode only.

See also

[m2m_wifi_connect](#) [m2m_wifi_default_connect](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



Functions

m2m_wifi_start_provision_mode

WLAN » Function

Functions

```
NMI_API sint8 m2m_wifi_start_provision_mode (tstrM2MAPConfig *pstrAPConfig, char  
*pcHttpServerDomainName, uint8 bEnableHttpRedirect)
```

Detailed Description

Asynchronous Wi-Fi provisioning function, which starts the WINC HTTP PROVISIONING mode. The function triggers the WINC to activate the Wi-Fi AP (HOTSPOT) mode with the passed configuration parameters and then starts the HTTP Provision WEB Server. The provisioning status is returned in an event [M2M_WIFI_RESP_PROVISION_INFO](#)

Function Documentation

◆ m2m_wifi_start_provision_mode()

```
NMI_API sint8  
m2m_wifi_start_provision_mode ( tstrM2MAPConfig * pstrAPConfig,  
                                char * pcHttpServerDomainName,  
                                uint8 bEnableHttpRedirect  
                            )
```

Parameters

[in] pstrAPConfig	AP configuration parameters as defined in tstrM2MAPConfig configuration structure. A NULL value passed in, will result in a negative error M2M_ERR_FAIL .
[in] pcHttpServerDomainName	Domain name of the HTTP Provision WEB server which others will use to load the provisioning Home page. The domain name can have one of the following 3 forms: 1- "wincprov.com" 2- "http://wincprov.com" 3- "https://wincprov.com" The forms 1 and 2 are equivalent, they both will start a plain http server, while form 3 will start a secure HTTP provisioning Session (HTTP over SSL connection).
[in] bEnableHttpRedirect	A flag to enable/disable the HTTP redirect feature. If Secure provisioning is enabled (i.e. the server domain name uses "https" prefix) this flag is ignored (no meaning for redirect in HTTPS). Possible values are: <ul style="list-style-type: none">• ZERO DO NOT Use HTTP Redirect. In this case the associated device could open the provisioning page ONLY when the HTTP Provision URL of the WINC HTTP Server is correctly written on the browser.• Non-Zero value Use HTTP Redirect. In this case, all http traffic (http://URL) from the associated device (Phone, PC, ...etc) will be redirected to the WINC HTTP Provisioning Home page.

Precondition

- A Wi-Fi notification callback of type **tpfAppWifiCb** MUST be implemented and registered at startup. Registering the callback is done through passing it to the initialization **m2m_wifi_init** function.
- The event **M2M_WIFI_RESP_CONN_INFO** must be handled in the callback to receive the requested connection info.

See also

[tpfAppWifiCb](#) [m2m_wifi_init](#) [M2M_WIFI_RESP_PROVISION_INFO](#)
[m2m_wifi_stop_provision_mode](#) [tstrM2MAPConfig](#)

Warning

DO Not use ".local" in the **pcHttpServerDomainName**.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The example demonstrates a code snippet for how provisioning is triggered and the response event received accordingly.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI RESP_PROVISION_INFO:
            {
                tstrM2MProvInfo *pstrProvInfo = (tstrM2MProvInfo*)pvMsg;
                if(pstrProvInfo->u8Status == M2M_SUCCESS)
                {
                    m2m_wifi_connect((char*)pstrProvInfo->au8SSID,
                        (uint8)strlen(pstrProvInfo->au8SSID), pstrProvInfo->u8SecType,
                        pstrProvInfo->au8Password, M2M_WIFI_CH_ALL);

                    printf("PROV SSID : %s\n", pstrProvInfo->au8SSID);
                    printf("PROV PSK : %s\n", pstrProvInfo->au8Password);
                }
            }
        else
        {
            printf("(ERR) Provisioning Failed\n");
        }
    }
    break;

    default:
    break;
}
}

int main()
{
    tstrWifiInitParam param;

    param.pfAppWifiCb = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig apConfig;
        uint8 bEnableRedirect = 1;

        strcpy(apConfig.au8SSID, "WINC_SSID");
        apConfig.u8ListenChannel = 1;
        apConfig.u8SecType = M2M_WIFI_SEC_OPEN;
        apConfig.u8SsidHide = 0;

        // IP Address
        apConfig.au8DHCPServerIP[0] = 192;
        apConfig.au8DHCPServerIP[1] = 168;
        apConfig.au8DHCPServerIP[2] = 1;
    }
}
```

```
    apConfig.au8DHCPServerIP[0] = 1;  
  
    m2m_wifi_start_provision_mode(&apConfig, "atmelwinccconf.com",  
        bEnableRedirect);  
  
    while(1)  
    {  
        m2m_wifi_handle_events(NULL);  
    }  
}
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



Functions

m2m_wifi_stop_provision_mode

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_stop_provision_mode (void)

Detailed Description

Synchronous provision termination function which stops the provision mode if it is active.

Function Documentation

- ◆ [m2m_wifi_stop_provision_mode\(\)](#)

`sint8 m2m_wifi_stop_provision_mode (void)`

Precondition

An active provisioning session must be active before it is terminated through this function.

See also

[m2m_wifi_start_provision_mode](#)

Returns

The function returns ZERO for success and a negative value otherwise.



m2m_wifi_get_connection_info

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_get_connection_info (void)

Detailed Description

Asynchronous connection status retrieval function, retrieves the status information of the currently connected AP. The result is passed to the Wi-Fi notification callback through the event **M2M_WIFI_RESP_CONN_INFO**. Connection information is retrieved from the structure **tstrM2MConnInfo**.

Function Documentation

◆ m2m_wifi_get_connection_info()

sint8 m2m_wifi_get_connection_info (void)

Retrieve the current Connection information. The result is passed to the Wi-Fi notification callback **M2M_WIFI_RESP_CONN_INFO**.

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at startup. Registering the callback is done through passing it to the initialization **m2m_**

function.

- The event **M2M_WIFI_RESP_CONN_INFO** must be handled in the callback to receive requested connection info.

Connection Information retrieved:

-Connection Security
-Connection RSSI
-Remote MAC address
-Remote IP address

and in case of WINC station mode the SSID of the AP is also retrieved

Warning

-In case of WINC AP mode or P2P mode, ignore the SSID field (NULL string).

See also

[M2M_WIFI_RESP_CONN_INFO](#), [tstrM2MConnInfo](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The code snippet shows an example of how wi-fi connection information is retrieved .

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_CONN_INFO:
        {
            tstrM2MConnInfo     *pstrConnInfo = (tstrM2MConnInfo*)pvMsg;

            printf("CONNECTED AP INFO\n");
            printf("SSID           : %s\n", pstrConnInfo->acSSID);
            printf("SEC TYPE       : %d\n", pstrConnInfo->u8SecType);
            printf("Signal Strength : %d\n", pstrConnInfo->s8RSSI);
            printf("Local IP Address : %.d.%d.%d.%d\n",
                   pstrConnInfo->au8IPAddr[0] , pstrConnInfo->au8IPAddr[1],
                   pstrConnInfo->au8IPAddr[2], pstrConnInfo->au8IPAddr[3]);
        }
        break;

        case M2M_WIFI_REQ_DHCP_CONF:
        {
            // Get the current AP information.
            m2m_wifi_get_connection_info();
        }
        break;
        default:
        break;
    }
}
```

```
    }

}

int main()
{
    tstrWifiInitParam    param;

    param.pfAppWifiCb    = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // connect to the default AP
        m2m_wifi_default_connect();

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_set_mac_address

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_set_mac_address (uint8 au8MacAddress[6])

Detailed Description

Synchronous MAC address assigning to the NMC1500. It is used for non-production SW. Assign MAC address to the WINC device.

Function Documentation

♦ m2m_wifi_set_mac_address()

NMI_API sint8 m2m_wifi_set_mac_address (uint8 au8MacAddress[6])

Assign a MAC address to the WINC board. This function override the already assigned MAC address of the WINC board with a user provided one. This is for experimental use only and should never be used in the production SW.

Parameters

[in] **au8MacAddress** MAC Address to be set to the WINC.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_wps

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_wps (uint8 u8TriggerType, const char *pcPinNumber)

Detailed Description

Asynchronous WPS triggering function. This function is called for the WINC to enter the WPS (Wi-Fi Protected Setup) mode. The result is passed to the Wi-Fi notification callback with the event [M2M_WIFI_REQ_WPS](#).

Function Documentation

◆ m2m_wifi_wps()

```
NMI_API sint8 m2m_wifi_wps ( uint8          u8TriggerType,  
                           const char * pcPinNumber  
                         )
```

Parameters

[in] **u8TriggerType** WPS Trigger method. Could be:

- [WPS_PIN_TRIGGER](#) Push button method
- [WPS_PBC_TRIGGER](#) Pin method

[in] **pcPinNumber** PIN number for WPS PIN method. It is not used if the trigger type is WPS_PBC_TRIGGER. It must follow the rules stated by the WPS standard.

Warning

This function is not allowed in AP or P2P modes.

Precondition

- A Wi-Fi notification callback of type (tpfAppWifiCb MUST be implemented and registered at startup. Registering the callback is done through passing it to the **m2m_wifi_init**.
- The event **M2M_WIFI_REQ_WPS** must be handled in the callback to receive the WPS status.
- The WINC device MUST be in IDLE or STA mode. If AP or P2P mode is active, the WPS will not be performed.
- The **m2m_wifi_handle_events** MUST be called periodically to receive the responses in the callback.

See also

tpfAppWifiCb **m2m_wifi_init** **M2M_WIFI_REQ_WPS** **tenuWPSTrigger**
tstrM2MWPSInfo

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The code snippet shows an example of how Wi-Fi WPS is triggered .

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI_REQ_WPS:
            {
                tstrM2MWPSInfo *pstrWPS = (tstrM2MWPSInfo*)pvMsg;
                if(pstrWPS->u8AuthType != 0)
                {
                    printf("WPS SSID           : %s\n", pstrWPS->au8SSID);
                    printf("WPS PSK            : %s\n", pstrWPS->au8PSK);
                    printf("WPS SSID Auth Type : %s\n", pstrWPS->u8AuthType ==
M2M_WIFI_SEC_OPEN ? "OPEN" : "WPA/WPA2");
                    printf("WPS Channel         : %d\n", pstrWPS->u8Ch + 1);

// establish Wi-Fi connection
m2m_wifi_connect((char*)pstrWPS->au8SSID, (uint8)m2m_strlen(pstrWPS-
>au8SSID),
                    pstrWPS->u8AuthType, pstrWPS->au8PSK, pstrWPS->u8Ch);
```

```

        }
    else
    {
        printf("(ERR) WPS Is not enabled OR Timed out\n");
    }
break;

default:
break;
}
}

int main()
{
tstrWifiInitParam    param;

param.pfAppWifiCb    = wifi_event_cb;
if(!m2m_wifi_init(&param))
{
// Trigger WPS in Push button mode.
m2m_wifi_wps(WPS_PBC_TRIGGER, NULL);

while(1)
{
m2m_wifi_handle_events(NULL);
}
}
}

```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_wps_disable

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_wps_disable (void)

Detailed Description

Disable the WINC1500 WPS operation.

Function Documentation

◆ m2m_wifi_wps_disable()

NMI_API sint8 m2m_wifi_wps_disable (void)

Stops the WPS ongoing session.

Precondition

WINC should be already in WPS mode using [m2m_wifi_wps](#)

See also

[m2m_wifi_wps](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value

otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_p2p

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_p2p (uint8 u8Channel)

Detailed Description

Asynchronous Wi-Fi direct (P2P) enabling mode function. The WINC supports P2P in device listening mode ONLY (intent is ZERO). The WINC P2P implementation does not support P2P GO (Group Owner) mode. Active P2P devices (e.g. phones) could find the WINC in the search list. When a device is connected to WINC, a Wi-Fi notification event

M2M_WIFI_RESP_CON_STATE_CHANGED is triggered. After a short while, the DHCP IP Address is obtained and an event **M2M_WIFI_REQ_DHCP_CONF** is triggered. Refer to the code examples for a more illustrative example.

Function Documentation

◆ m2m_wifi_p2p()

NMI_API sint8 m2m_wifi_p2p (uint8 u8Channel)

Parameters

[in] **u8Channel** P2P Listen RF channel. According to the P2P standard It must hold only

following values 1, 6 or 11.

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered. Registering the callback is done through passing it to the [m2m_wifi_init](#).
- The events [M2M_WIFI RESP CON STATE CHANGED](#) and [M2M_WIFI_REQ_DH](#) be handled in the callback.
- The [m2m_wifi_handle_events](#) MUST be called to receive the responses in the callback.

Warning

This function is not allowed in AP or STA modes.

See also

[tpfAppWifiCb](#) [m2m_wifi_init](#) [M2M_WIFI_RESP_CON_STATE_CHANGED](#) [M2M_WIFI_REQ_DH](#)
[tstrM2mWifiStateChanged](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

Example

The code snippet shown an example of how the p2p mode operates.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_CON_STATE_CHANGED:
            tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged*)pvMsg;
            M2M_INFO("Wifi State :: %s :: ErrCode %d\n", pstrWifiState->u8Cur
"CONNECTED": "DISCONNECTED", pstrWifiState->u8ErrCode);

            // Do something
        }
        break;

        case M2M_WIFI_REQ_DHCP_CONF:
            {
                uint8 *pu8IPAddress = (uint8*)pvMsg;

                printf("P2P IP Address
 \"%u.%u.%u.%u\"\n", pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
            }
        break;

        default:
        break;
    }

int main()
{
```

```
tstrWifiInitParam param;

    param.pfAppWifiCb = wifi_event_cb;
if(!m2m_wifi_init(&param))
{
// Trigger P2P
m2m_wifi_p2p(M2M_WIFI_CH_1);

while(1)
{
m2m_wifi_handle_events(NULL);
}
}
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_p2p_disconnect

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_p2p_disconnect (void)

Detailed Description

Disable the WINC1500 device Wi-Fi direct mode (P2P).

Function Documentation

◆ m2m_wifi_p2p_disconnect()

NMI_API sint8 m2m_wifi_p2p_disconnect (void)

Precondition

The p2p mode must have been enabled and active before a disconnect can be called.

See also

[m2m_wifi_p2p](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



m2m_wifi_enable_ap

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_enable_ap (CONST tstrM2MAPConfig *pstrM2MAPConfig)

Detailed Description

Asynchronous Wi-Fi hot-spot enabling function. The WINC supports AP mode operation with the following limitations:

- Only 1 STA could be associated at a time.
- Open and WEP are the only supported security types

Function Documentation

◆ m2m_wifi_enable_ap()

NMI_API sint8 m2m_wifi_enable_ap (CONST tstrM2MAPConfig * pstrM2MAPConfig)

Parameters

[in] **pstrM2MAPConfig** A structure holding the AP configurations.

Warning

This function is not allowed in P2P or STA modes.

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered. Registering the callback is done through passing it to the [m2m_wifi_init](#).
- The event [M2M_WIFI_REQ_DHCP_CONF](#) must be handled in the callback.
- The [m2m_wifi_handle_events](#) MUST be called to receive the responses in the callback.

See also

tpfAppWifiCb [tenuM2mSecType](#) [m2m_wifi_init](#) [M2M_WIFI_REQ_DHCP_CONF](#)
[tstrM2mWifiStateChanged](#) [tstrM2MAPConfig](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

Example

The code snippet demonstrates how the AP mode is enabled after the driver is initialized in the ap function and the handling of the event [M2M_WIFI_REQ_DHCP_CONF](#), to indicate successful connection.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    switch(u8WiFiEvent)
    {
        case M2M_WIFI_REQ_DHCP_CONF:
        {
            uint8 *pu8IPAddress = (uint8*)pvMsg;

            printf("Associated STA has IP Address
                \"%u.%u.%u.%u\"\n", pu8IPAddress[0], pu8IPAddress[1], pu8IPAddress[2], pu8IPAddress[3]);
        }
        break;

        default:
        break;
    }
}

int main()
{
    tstrWifiInitParam    param;

    param.pfAppWifiCb    = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        tstrM2MAPConfig      apConfig;

        strcpy(apConfig.au8SSID, "WINC_SSID");
        apConfig.u8ListenChannel    = 1;
        apConfig.u8SecType          = M2M_WIFI_SEC_OPEN;
        apConfig.u8SsidHide         = 0;

        // IP Address
        apConfig.au8DHCPServerIP[0] = 192;
        apConfig.au8DHCPServerIP[1] = 168;
        apConfig.au8DHCPServerIP[2] = 1;
    }
}
```

```
    apConfig.au8DHCPServerIP[0] = 1;

// Trigger AP
m2m_wifi_enable_ap(&apConfig);

while(1)
{
    m2m_wifi_handle_events(NULL);
}

}
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

Functions

m2m_wifi_disable_ap

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_disable_ap (void)

Detailed Description

Synchronous Wi-Fi hot-spot disabling function. Must be called only when the AP is enabled through the [m2m_wifi_enable_ap](#) function. Otherwise the call to this function will not be useful.

Function Documentation

◆ m2m_wifi_disable_ap()

NMI_API sint8 m2m_wifi_disable_ap (void)

See also

[m2m_wifi_enable_ap](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



m2m_wifi_set_static_ip

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_set_static_ip (tstrM2MIPConfig *pstrStaticIPConf)

Detailed Description

Synchronous static IP Address configuration function.

Function Documentation

♦ m2m_wifi_set_static_ip()

NMI_API sint8 m2m_wifi_set_static_ip (tstrM2MIPConfig * pstrStaticIPConf)

Assign a static IP address to the WINC board. This function assigns a static IP address in case the AP doesn't have a DHCP server or in case the application wants to assign a predefined known IP address. The user must take in mind that assigning a static IP address might result in an IP address conflict. In case of an IP address conflict observed by the WINC board the user will get a response of **M2M_WIFI_RESP_IP_CONFLICT** in the wifi callback. The application is then responsible to either solve the conflict or assign another IP address.

Parameters

[in] **pstrStaticIPConf** Pointer to a structure holding the static IP Configurations (IP,

Gateway, subnet mask and DNS address).

Precondition

The application must disable auto DHCP using `m2m_wifi_enable_dhcp` before assigning a static IP address.

Warning

Normally this function normally should not be used. DHCP configuration is requested automatically after successful Wi-Fi connection is established.

See also

[tstrM2MIPConfig](#)

Returns

The function returns `M2M_SUCCESS` for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by





m2m_wifi_request_dhcp_client

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_request_dhcp_client (void)

Detailed Description

Starts the DHCP client operation(DHCP requested by the firmware automatically in STA/AP/P2P mode).

Function Documentation

◆ m2m_wifi_request_dhcp_client()

NMI_API sint8 m2m_wifi_request_dhcp_client (void)

Warning

This function is legacy and exists only for compatibility with older applications. DHCP configuration is requested automatically after successful Wi-Fi connection is established.

Returns

The function returns **M2M_SUCCESS** always.



m2m_wifi_request_dhcp_server

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_request_dhcp_server (uint8 *addr)

Detailed Description

Dhcp requested by the firmware automatically in STA/AP/P2P mode).

Function Documentation

◆ m2m_wifi_request_dhcp_server()

NMI_API sint8 m2m_wifi_request_dhcp_server (uint8 * addr)

Warning

This function is legacy and exists only for compatibility with older applications. DHCP server is started automatically when enabling the AP mode.

Returns

The function returns **M2M_SUCCESS** always.



m2m_wifi_enable_dhcp

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_enable_dhcp (uint8 u8DhcpEn)

Detailed Description

Synchronous Wi-Fi DHCP enable function. This function Enable/Disable DHCP protocol.

Function Documentation

♦ m2m_wifi_enable_dhcp()

NMI_API sint8 m2m_wifi_enable_dhcp (uint8 u8DhcpEn)

Enable/Disable the DHCP client after connection.

Parameters

[in] **u8DhcpEn** Possible values: 1: Enable DHCP client after connection. 0: Disable DHCP client after connection. -DHCP client is enabled by default - This Function should be called before using [m2m_wifi_set_static_ip\(\)](#)

See also

`m2m_wifi_set_static_ip()`

Returns

The function SHALL return **M2M_SUCCESS** for successful operation and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_set_scan_options

[WLAN](#) » [Function](#)

Synchronous Wi-Fi scan settings function. This function sets the time configuration parameters for the scan operation.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by doxygen 1.8.13



m2m_wifi_set_scan_region

WLAN » Function

Functions

Functions

NMI_API **sint8** m2m_wifi_set_scan_region (**uint16** ScanRegion)

Detailed Description

Synchronous wi-fi scan region setting function. This function sets the scan region, which will affect the range of possible scan channels. For 2.5GHz supported in the current release, the requested scan region can't exceed the maximum number of channels (14).

Function Documentation

◆ m2m_wifi_set_scan_region()

sint8 m2m_wifi_set_scan_region (**uint16** ScanRegion)

Parameters

[in] **ScanRegion**; ASIA NORTH_AMERICA

See also

[tenuM2mScanCh m2m_wifi_request_scan](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_request_scan

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_request_scan (uint8 ch)

NMI_API sint8 m2m_wifi_request_scan_passive (uint8 ch, uint16 scan_time)

NMI_API sint8 m2m_wifi_request_scan_ssid_list (uint8 ch, uint8 *u8Ssidlist)

Detailed Description

Asynchronous Wi-Fi scan request on the given channel. The scan status is delivered in the wifi event callback and then the application is supposed to read the scan results sequentially. The number of APs found (N) is returned in event **M2M_WIFI_RESP_SCAN_DONE** with the number of found APs. The application reads the list of APs by calling the function **m2m_wifi_req_scan_result** N times.

Same as m2m_wifi_request_scan but perform passive scanning while the other one perform active scanning.

Asynchronous wi-fi scan request on the given channel and the hidden scan list. The scan status is delivered in the wi-fi event callback and then the application is to read the scan results sequentially. The number of APs found (N) is returned in event **M2M_WIFI_RESP_SCAN_DONE** with the number of found APs. The application could read the list of APs by calling the function **m2m_wifi_req_scan_result** N times.

Function Documentation

◆ m2m_wifi_request_scan()

```
NMI_API sint8 m2m_wifi_request_scan ( uint8 ch )
```

Parameters

- [in] **ch** RF Channel ID for SCAN operation. It should be set according to tenuM2mScanCh. With a value of M2M_WIFI_CH_ALL(255), means to scan all channels.

Warning

This function is not allowed in P2P or AP modes. It works only for STA mode (both connected or disconnected states).

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at initialization. Registering the callback is done through passing it to the [m2m_wifi_init](#).
- The events [M2M_WIFI_RESP_SCAN_DONE](#) and [M2M_WIFI_RESP_SCAN_RESULT](#) must be handled in the callback.
- The [m2m_wifi_handle_events](#) function MUST be called to receive the responses in the callback.

See also

[M2M_WIFI_RESP_SCAN_DONE](#) [M2M_WIFI_RESP_SCAN_RESULT](#) [tpfAppWifiCb](#)
[tstrM2mWifiscanResult](#) [tenuM2mScanCh](#) [m2m_wifi_init](#) [m2m_wifi_handle_events](#)
[m2m_wifi_req_scan_result](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

Example

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8     u8ScanResultIdx = 0;
```

```

switch(u8WiFiEvent)
{
case M2M_WIFI_RESP_SCAN_DONE:
{
tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;

    printf("Num of AP found %d\n", pstrInfo->u8NumofCh);
if(pstrInfo->s8ScanState == M2M_SUCCESS)
{
    u8ScanResultIdx = 0;
if(pstrInfo->u8NumofCh >= 1)
{
m2m_wifi_req_scan_result(u8ScanResultIdx);
    u8ScanResultIdx++;
}
else
{
    printf("No AP Found Rescan\n");
m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
}
}
else
{
    printf("(ERR) Scan fail with error <%d>\n", pstrInfo-
>s8ScanState);
}
break;
}

case M2M_WIFI_RESP_SCAN_RESULT:
{
tstrM2mWifiscanResult *pstrScanResult =
(tstrM2mWifiscanResult*)pvMsg;
uint8 u8NumFoundAPs = m2m_wifi_get_num_ap_found();

    printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X SSID %s\n",
pstrScanResult->u8index, pstrScanResult->s8rssi,
pstrScanResult->u8AuthType,
pstrScanResult->u8ch,
pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
pstrScanResult->au8SSID);

if(u8ScanResultIdx < u8NumFoundAPs)
{
// Read the next scan result
m2m_wifi_req_scan_result(index);
    u8ScanResultIdx++;
}
}
break;
default:
break;
}
}

int main()
{
tstrWifiInitParam param;

    param.pfAppWifiCb    = wifi_event_cb;

```

```

if(!m2m_wifi_init(&param))
{
// Scan all channels
m2m_wifi_request_scan(M2M_WIFI_CH_ALL);

while(1)
{
m2m_wifi_handle_events(NULL);
}
}

```

◆ m2m_wifi_request_scan_passive()

```

NMI_API sint8 m2m_wifi_request_scan_passive ( uint8 ch,
                                              uint16 scan_time
                                              )

```

Parameters

- [in] **ch** RF Channel ID for SCAN operation. It should be set according to tenuM2mScanCh. With a value of M2M_WIFI_CH_ALL(255), means to scan all channels.
- [in] **scan_time** The time in ms that passive scan is listening to beacons on each channel per one slot, enter 0 for deafult setting.

Warning

This function is not allowed in P2P or AP modes. It works only for STA mode (both connected or disconnected states).

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at initialization. Registering the callback is done through passing it to the [m2m_wifi_init](#).
- The events [M2M_WIFI_RESP_SCAN_DONE](#) and [M2M_WIFI_RESP_SCAN_RESULT](#). must be handled in the callback.
- The [m2m_wifi_handle_events](#) function MUST be called to receive the responses in the callback.

See also

[m2m_wifi_request_scan](#) [M2M_WIFI_RESP_SCAN_DONE](#)
[M2M_WIFI_RESP_SCAN_RESULT](#) [tpfAppWifiCb](#) [tstrM2mWifiscanResult](#)
[tenuM2mScanCh](#) [m2m_wifi_init](#) [m2m_wifi_handle_events](#)
[m2m_wifi_req_scan_result](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value

otherwise.

◆ m2m_wifi_request_scan_ssid_list()

```
NMI_API sint8 m2m_wifi_request_scan_ssid_list ( uint8 ch,  
                                              uint8 * u8SsidList  
                                              )
```

Parameters

- [in] **ch** RF Channel ID for SCAN operation. It should be set according to tenuM2mScanCh. With a value of M2M_WIFI_CH_ALL(255), means to scan all channels.
- [in] **u8SsidList** u8SsidList is a buffer containing a list of hidden SSIDs to include during the scan. The first byte in the buffer, u8SsidList[0], is the number of SSIDs encoded in the string. The number of hidden SSIDs cannot exceed MAX_HIDDEN_SITES. All SSIDs are concatenated in the following bytes and each SSID is prefixed with a one-byte header containing its length. The total number of bytes in u8SsidList buffer, including length byte, cannot exceed 133 bytes (MAX_HIDDEN_SITES SSIDs x 32 bytes each, which is max SSID length). For instance, encoding the two hidden SSIDs "DEMO_AP" and "TEST" results in the following buffer content:

```
uint8 u8SsidList[14];  
u8SsidList[0] = 2; // Number of SSIDs is 2  
u8SsidList[1] = 7; // Length of the string "DEMO_AP"  
                  // without NULL termination  
memcpy(&u8SsidList[2], "DEMO_AP", 7); // Bytes index  
                                         2-9 containing the string DEMO_AP  
u8SsidList[9] = 4; // Length of the string "TEST"  
                  // without NULL termination  
memcpy(&u8SsidList[10], "TEST", 4); // Bytes index  
                                         10-13 containing the string TEST
```

Warning

This function is not allowed in P2P. It works only for STA/AP mode (connected or disconnected).

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered at initialization. Registering the callback is done through passing it to the **m2m_wifi_init**.
- The events **M2M_WIFI_RESP_SCAN_DONE** and **M2M_WIFI_RESP_SCAN_RESULT**. must be handled in the callback.
- The **m2m_wifi_handle_events** function MUST be called to receive the responses in the callback.

See also

[M2M_WIFI_RESP_SCAN_DONE](#) [M2M_WIFI_RESP_SCAN_RESULT](#) [tpfAppWifiCb](#)
[tstrM2mWifiscanResult](#) [tenuM2mScanCh](#) [m2m_wifi_init](#) [m2m_wifi_handle_events](#)
[m2m_wifi_req_scan_result](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.

Example

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

static void request_scan_hidden_demo_ap(void);

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8     u8ScanResultIdx = 0;

    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_SCAN_DONE:
            {
                tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;

                printf("Num of AP found %d\n", pstrInfo->u8NumofCh);
                if(pstrInfo->s8ScanState == M2M_SUCCESS)
                {
                    u8ScanResultIdx = 0;
                    if(pstrInfo->u8NumofCh >= 1)
                    {
                        m2m_wifi_req_scan_result(u8ScanResultIdx);
                        u8ScanResultIdx++;
                    }
                }
                else
                {
                    printf("No AP Found Rescan\n");
                    request_scan_hidden_demo_ap();
                }
            }
        else
        {
            printf("(ERR) Scan fail with error <%d>\n", pstrInfo-
>s8ScanState);
        }
    }
    break;

    case M2M_WIFI_RESP_SCAN_RESULT:
        {
            tstrM2mWifiscanResult      *pstrScanResult =
                (tstrM2mWifiscanResult*)pvMsg;
            uint8                      u8NumFoundAPs = m2m_wifi_get_num_ap_found();
        }
    break;
}
```

```

        printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X SSID %s\n",
        pstrScanResult->u8index,pstrScanResult->s8rssi,
        pstrScanResult->u8AuthType,
        pstrScanResult->u8ch,
        pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
        pstrScanResult->au8BSSID[2],
        pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
        pstrScanResult->au8BSSID[5],
        pstrScanResult->au8SSID);

    if(u8ScanResultIdx < u8NumFoundAPs)
    {
        // Read the next scan result
        m2m_wifi_req_scan_result(index);
        u8ScanResultIdx++;
    }
    break;
    default:
    break;
}
}

static void request_scan_hidden_demo_ap(void)
{
    uint8 list[9];
    char ssid[] = "DEMO_AP";
    uint8 len = (uint8)(sizeof(ssid)-1);

    list[0] = 1;
    list[1] = len;
    memcpy(&list[2], ssid, len); // copy 7 bytes
    // Scan all channels
    m2m_wifi_request_scan_ssid_list(M2M_WIFI_CH_ALL, list);
}

int main()
{
    tstrWifiInitParam param;

    param.pfAppWifiCb = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        request_scan_hidden_demo_ap();

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}

```



m2m_wifi_get_num_ap_found

WLAN » Function

Functions

Functions

NMI_API uint8 m2m_wifi_get_num_ap_found (void)

Detailed Description

Synchronous function to retrieve the number of AP's found in the last scan request. The function reads the number of APs from global variable which was updated in the Wi-Fi callback function through the M2M_WIFI_RESP_SCAN_DONE event. Function used only in STA mode only.

Function Documentation

◆ m2m_wifi_get_num_ap_found()

NMI_API uint8 m2m_wifi_get_num_ap_found (void)

See also

[m2m_wifi_request_scan](#) [M2M_WIFI_RESP_SCAN_DONE](#)
[M2M_WIFI_RESP_SCAN_RESULT](#)

Precondition

m2m_wifi_request_scan need to be called first

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and

registered at initialization. Registering the callback is done through passing it to the [m2m_wifi_init](#).

- The event [M2M_WIFI_RESP_SCAN_DONE](#) must be handled in the callback to receive the requested scan information.

Warning

This function must be called only in the wi-fi callback function when the events [M2M_WIFI_RESP_SCAN_DONE](#) or [M2M_WIFI_RESP_SCAN_RESULT](#) are received. Calling this function in any other place will result in undefined/outdated numbers.

Returns

Return the number of AP's found in the last Scan Request.

Example

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8     u8ScanResultIdx = 0;

    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_SCAN_DONE:
            {
                tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;

                printf("Num of AP found %d\n", pstrInfo->u8NumofCh);
                if(pstrInfo->s8ScanState == M2M_SUCCESS)
                {
                    u8ScanResultIdx = 0;
                    if(pstrInfo->u8NumofCh >= 1)
                    {
                        m2m_wifi_req_scan_result(u8ScanResultIdx);
                        u8ScanResultIdx++;
                    }
                }
                else
                {
                    printf("No AP Found Rescan\n");
                    m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
                }
            }
        else
        {
            printf("(ERR) Scan fail with error <%d>\n", pstrInfo-
>s8ScanState);
        }
    }
    break;

    case M2M_WIFI_RESP_SCAN_RESULT:
        {
            tstrM2mWifiscanResult      *pstrScanResult =

```

```

(tstrM2mWifiscanResult*)pvMsg;
uint8 u8NumFoundAPs = m2m_wifi_get_num_ap_found();

printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X SSID %s\n",
pstrScanResult->u8index,pstrScanResult->s8rssi,
pstrScanResult->u8AuthType,
pstrScanResult->u8ch,
pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
pstrScanResult->au8SSID);

if(u8ScanResultIdx < u8NumFoundAPs)
{
// Read the next scan result
m2m_wifi_req_scan_result(index);
u8ScanResultIdx++;
}
}
break;
default:
break;
}
}

int main()
{
tstrWifiInitParam param;

param.pfAppWifiCb = wifi_event_cb;
if(!m2m_wifi_init(&param))
{
// Scan all channels
m2m_wifi_request_scan(M2M_WIFI_CH_ALL);

while(1)
{
m2m_wifi_handle_events(NULL);
}
}
}

```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_req_scan_result

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_req_scan_result (uint8 index)

Detailed Description

Synchronous call to read the AP information from the SCAN Result list with the given index. This function is expected to be called when the response events M2M_WIFI_RESP_SCAN_RESULT or M2M_WIFI_RESP_SCAN_DONE are received in the wi-fi callback function. The response information received can be obtained through the casting to the **tstrM2mWifiscanResult** structure

Function Documentation

◆ m2m_wifi_req_scan_result()

NMI_API sint8 m2m_wifi_req_scan_result (uint8 index)

Parameters

[in] **index** Index for the requested result, the index range start from 0 till number of AP's found

See also

`tstrM2mWifiscanResult m2m_wifi_get_num_ap_found m2m_wifi_request_scan`

Precondition

`m2m_wifi_request_scan` needs to be called first, then `m2m_wifi_get_num_ap_found` to get the number of AP's found

- A Wi-Fi notification callback of type `tpfAppWifiCb` MUST be implemented and registered at startup. Registering the callback is done through passing it to the `m2m_wifi_init` function.
- The event `M2M_WIFI_RESP_SCAN_RESULT` must be handled in the callback to receive the requested scan information.

Warning

Function used in STA mode only. the scan results are updated only if the scan request is called. Calling this function only without a scan request will lead to firmware errors. Refrain from introducing a large delay between the scan request and the scan result request, to prevent errors occurring.

Returns

The function returns `M2M_SUCCESS` for successful operations and a negative value otherwise.

Example

The code snippet demonstrates an example of how the scan request is called from the application's main function and the handling of the events received in response.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8     u8ScanResultIdx = 0;

    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_SCAN_DONE:
            {
                tstrM2mScanDone *pstrInfo = (tstrM2mScanDone*)pvMsg;

                printf("Num of AP found %d\n", pstrInfo->u8NumofCh);
                if(pstrInfo->s8ScanState == M2M_SUCCESS)
                {
                    u8ScanResultIdx = 0;
                    if(pstrInfo->u8NumofCh >= 1)
                    {
                        m2m_wifi_req_scan_result(u8ScanResultIdx);
                        u8ScanResultIdx++;
                    }
                }
                else
                {
                    printf("No AP Found Rescan\n");
                    m2m_wifi_request_scan(M2M_WIFI_CH_ALL);
                }
            }
    }
}
```

```

else
{
    printf("(ERR) Scan fail with error <%d>\n", pstrInfo->s8ScanState);
}
break;

case M2M_WIFI_RESP_SCAN_RESULT:
{
tstrM2mWifiscanResult *pstrScanResult =
(tstrM2mWifiscanResult*)pvMsg;
uint8 u8NumFoundAPs = m2m_wifi_get_num_ap_found();

printf(">>%02d RI %d SEC %s CH %02d BSSID
%02X:%02X:%02X:%02X:%02X SSID %s\n",
pstrScanResult->u8index, pstrScanResult->s8rssi,
pstrScanResult->u8AuthType,
pstrScanResult->u8ch,
pstrScanResult->au8BSSID[0], pstrScanResult->au8BSSID[1],
pstrScanResult->au8BSSID[2],
pstrScanResult->au8BSSID[3], pstrScanResult->au8BSSID[4],
pstrScanResult->au8BSSID[5],
pstrScanResult->au8SSID);

if(u8ScanResultIdx < u8NumFoundAPs)
{
// Read the next scan result
m2m_wifi_req_scan_result(index);
u8ScanResultIdx++;
}
break;
default:
break;
}
}

int main()
{
tstrWifiInitParam param;

param.pfAppWifiCb = wifi_event_cb;
if(!m2m_wifi_init(&param))
{
// Scan all channels
m2m_wifi_request_scan(M2M_WIFI_CH_ALL);

while(1)
{
m2m_wifi_handle_events(NULL);
}
}
}

```



m2m_wifi_req_curr_rssi

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_req_curr_rssi (void)

Detailed Description

Asynchronous request for the current RSSI of the connected AP. The response received in through the **M2M_WIFI_RESP_CURRENT_RSSI** event.

Function Documentation

◆ m2m_wifi_req_curr_rssi()

NMI_API sint8 m2m_wifi_req_curr_rssi (void)

Precondition

- A Wi-Fi notification callback of type tpfAppWifiCb MUST be implemented and registered before initialization. Registering the callback is done through passing it to the **m2m_wifi_init** through the **tstrWifiInitParam** initialization structure.
 - The event **M2M_WIFI_RESP_CURRENT_RSSI** must be handled in the callback to receive the requested Rssi information.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The code snippet demonstrates how the RSSI request is called in the application's main function and the handling of the event received in the callback.

```
#include "m2m_wifi.h"
#include "m2m_types.h"

void wifi_event_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    static uint8     u8ScanResultIdx = 0;

    switch(u8WiFiEvent)
    {
        case M2M_WIFI_RESP_CURRENT_RSSI:
            {
                sint8    *rss = (sint8*)pvMsg;
                M2M_INFO("ch rss %d\n", *rss);
            }
        break;
        default:
        break;
    }
}

int main()
{
    tstrWifiInitParam    param;

    param.pfAppWifiCb    = wifi_event_cb;
    if(!m2m_wifi_init(&param))
    {
        // Scan all channels
        m2m_wifi_req_curr_rssi();

        while(1)
        {
            m2m_wifi_handle_events(NULL);
        }
    }
}
```



m2m_wifi_get_otp_mac_address

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_get_otp_mac_address (uint8 *pu8MacAddr, uint8 *pu8IsValid)

Detailed Description

Request the MAC address stored on the One Time Programmable(OTP) memory of the device. The function is blocking until the response is received.

Function Documentation

◆ m2m_wifi_get_otp_mac_address()

```
NMI_API sint8 m2m_wifi_get_otp_mac_address ( uint8 * pu8MacAddr,  
                                            uint8 * pu8IsValid  
)
```

Parameters

[out] **pu8MacAddr** Output MAC address buffer of 6 bytes size. Valid only if *pu8IsValid=1.

[out] **pu8IsValid** Output boolean value to indicate the validity of pu8MacAddr in OTP. Output zero if the OTP memory is not programmed, non-

zero otherwise.

Precondition

`m2m_wifi_init` required to be called before any WIFI/socket function

See also

`m2m_wifi_get_mac_address`

Returns

The function returns `M2M_SUCCESS` for success and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_get_mac_address

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_get_mac_address (uint8 *pu8MacAddr)

Detailed Description

Function to retrieve the current MAC address. The function is blocking until the response is received.

Function Documentation

◆ m2m_wifi_get_mac_address()

NMI_API sint8 m2m_wifi_get_mac_address (uint8 * pu8MacAddr)

Parameters

[out] **pu8MacAddr** Output MAC address buffer of 6 bytes size.

Precondition

m2m_wifi_init required to be called before any WIFI/socket function

See also

[m2m_wifi_get_otp_mac_address](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_set_sleep_mode

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_set_sleep_mode (uint8 PsTyp, uint8 BcastEn)

Detailed Description

This is one of the two synchronous power-save setting functions that allow the host MCU application to tweak the system power consumption. Such tweaking can be done through one of two ways: 1) Changing the power save mode, to one of the allowed power save modes **tenuPowerSaveModes**. This is done by setting the first parameter 2) Configuring DTIM monitoring: Configuring beacon monitoring parameters by enabling or disabling the reception of broadcast/multicast data. this is done by setting the second parameter.

Function Documentation

- ◆ **m2m_wifi_set_sleep_mode()**

```
NMI_API sint8 m2m_wifi_set_sleep_mode ( uint8 PsTyp,  
                                         uint8 BcastEn  
                                       )
```

Parameters

[in] **PsTyp** Desired power saving mode. Supported types are enumerated in [tenuPowerSaveModes](#).

[in] **BcastEn** Broadcast reception enable flag. If it is 1, the WINC1500 will be awake each DTIM beacon for receiving broadcast traffic. If it is 0, the WINC1500: disable broadcast traffic. Through this flag the WINC1500 will not wakeup at the DTIM beacon, but it will wakeup depends only on the the configured Listen Interval.

Warning

The function called once after initialization.

See also

[tenuPowerSaveModes](#) [m2m_wifi_get_sleep_mode](#) [m2m_wifi_set_lsn_int](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



m2m_wifi_request_sleep

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_request_sleep (uint32 u32SlpReqTime)

Detailed Description

Synchronous power-save sleep request function, which requests from the WINC1500 device to sleep in the currently configured power save mode as defined by the [m2m_wifi_set_sleep_mode](#), for a specific time as defined by the passed in parameter. This function should be used in the [M2M_PS_MANUAL](#) power save mode only. A wake up request is automatically performed by the WINC1500 device when any host driver API function, e.g. Wi-Fi or socket operation is called.

Function Documentation

- ◆ [m2m_wifi_request_sleep\(\)](#)

NMI_API sint8 m2m_wifi_request_sleep (uint32 u32SlpReqTime)

Parameters

[in] **u32SlpReqTime** Request sleep time in ms The best recommended sleep duration is left to be determined by the application. Taking into

account that if the application sends notifications very rarely, sleeping for a long time can be a power-efficient decision. In contrast applications that are sensitive for long periods of absence can experience performance degradation in the connection if long sleeping times are used.

Warning

The function should be called in **M2M_PS_MANUAL** power save mode only. As enumerated in **tenuPowerSaveModes** It's also important to note that during the sleeping time while in the M2M_PS_MANUAL mode, AP beacon monitoring is bypassed and the wifi-connection may drop if the sleep period is elongated.

See also

[tenuPowerSaveModes](#) [m2m_wifi_set_sleep_mode](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_get_sleep_mode

WLAN » Function

Functions

Functions

NMI_API uint8 m2m_wifi_get_sleep_mode (void)

Detailed Description

Synchronous power save mode retrieval function.

Function Documentation

◆ m2m_wifi_get_sleep_mode()

NMI_API uint8 m2m_wifi_get_sleep_mode (void)

See also

[tenuPowerSaveModes](#) [m2m_wifi_set_sleep_mode](#)

Returns

The current operating power saving mode based on the enumerated sleep modes
[tenuPowerSaveModes](#).



m2m_wifi_req_client_ctrl

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_req_client_ctrl (uint8 cmd)

Detailed Description

Asynchronous command sending function to the PS Client (An WINC1500 board running the ps_firmware) if the PS client send any command it will be received through the **M2M_WIFI_RESP_CLIENT_INFO** event

Function Documentation

◆ m2m_wifi_req_client_ctrl()

NMI_API sint8 m2m_wifi_req_client_ctrl (uint8 cmd)

Parameters

[in] **cmd** Control command sent from PS Server to PS Client (command values defined by the application)

Precondition

m2m_wifi_req_server_init should be called first

Warning

This mode is not supported in the current release.

See also

[m2m_wifi_req_server_init M2M_WIFI_RESP_CLIENT_INFO](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



m2m_wifi_req_server_init

Functions

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_req_server_init (uint8 ch)

Detailed Description

Synchronous function to initialize the PS Server. The WINC1500 supports non secure communication with another WINC1500, (SERVER/CLIENT) through one byte command (probe request and probe response) without any connection setup. The server mode can't be used with any other modes (STA/P2P/AP)

Function Documentation

- m2m_wifi_req_server_init()

NMI_API sint8 m2m_wifi_req_server_init (uint8 ch)

Parameters

[in] **ch** Server listening channel

See also

[m2m_wifi_req_client_ctrl](#)

Warning

This mode is not supported in the current release.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_set_device_name

WLAN » Function

Functions

Functions

```
NMI_API sint8 m2m_wifi_set_device_name (uint8 *pu8DeviceName, uint8  
u8DeviceNameLength)
```

Detailed Description

Sets the WINC device name. The name string is used as a device name in both (P2P) WiFi-Direct mode as well as DHCP hostname (option 12). For P2P devices to communicate a device name must be present. If it is not set through this function a default name is assigned. The default name is WINC-XX-YY, where XX and YY are the last 2 octets of the OTP MAC address. If OTP (eFuse) is programmed, then the default name is WINC-00-00.

Function Documentation

- ◆ [m2m_wifi_set_device_name\(\)](#)

```
NMI_API sint8 m2m_wifi_set_device_name ( uint8 * pu8DeviceName,  
                                         uint8 u8DeviceNameLength  
                                         )
```

Parameters

[in] **pu8DeviceName** A Buffer holding the device name. Device name is a null terminated C string.

[in] **u8DeviceNameLength** The length of the device name. Should not exceed the maximum device name's length
M2M_DEVICE_NAME_MAX (including null character).

Warning

The function called once after initialization. Used for the Wi-Fi Direct (P2P) as well as DHCP client hostname option (12).

Device name shall contain only characters allowed in valid internet host name as defined in RFC 952 and 1123.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_set_lsn_int

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_set_lsn_int (tstrM2mLsnInt *pstrM2mLsnInt)

Detailed Description

This is one of the two synchronous power-save setting functions that allow the host MCU application to tweak the system power consumption. Such tweaking can be done by modifying the the Wi-Fi listen interval. The listen interval is how many beacon periods the station can sleep before it wakes up to receive data buffer in AP. It is represented in units of AP beacon periods(100ms).

Function Documentation

◆ m2m_wifi_set_lsn_int()

NMI_API sint8 m2m_wifi_set_lsn_int (tstrM2mLsnInt * pstrM2mLsnInt)

Parameters

[in] **pstrM2mLsnInt** Structure holding the listen interval configurations.

Precondition

Function shall be called first, to set the power saving mode required.

Warning

The function should be called once after initialization.

See also

[tstrM2mLsnInt m2m_wifi_set_sleep_mode](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_enable_monitoring_mode

WLAN » Function

Functions

Functions

```
NMI_API sint8 m2m_wifi_enable_monitoring_mode (tstrM2MWifiMonitorModeCtrl  
*pstrMtrCtrl, uint8 *pu8PayloadBuffer, uint16 u16BufferSize, uint16  
u16DataOffset)
```

Detailed Description

Asynchronous Wi-Fi monitoring mode (Promiscuous mode) enabling function. This function enables the monitoring mode, thus allowing two operations to be performed: 1) Transmission of manually configured frames, through using the [m2m_wifi_send_wlan_pkt](#) function. 2) Reception of frames based on a defined filtering criteria. When the monitoring mode is enabled, reception of all frames that satisfy the filter criteria passed in as a parameter is allowed, on the current wireless channel.

- . All packets that meet the filtering criteria are passed to the application layer, to be handled by the assigned monitoring callback function
- . The monitoring callback function must be implemented before starting the monitoring mode, in-order to handle the packets received
- . Registering of the implemented callback function is through the callback pointer tpfAppMonCb in the [tstrWifiInitParam](#) structure
- . passed to [m2m_wifi_init](#) function at initialization.

Function Documentation

- ◆ [m2m_wifi_enable_monitoring_mode\(\)](#)

NMI_API sint8

```
m2m_wifi_enable_monitoring_mode ( tstrM2MWifiMonitorModeCtrl * pstrMtrCtrl,  
                                  uint8 * pu8PayloadBuffer,  
                                  uint16 u16BufferSize,  
                                  uint16 u16DataOffset  
                                )
```

Parameters

[in] pstrMtrCtrl	Pointer to tstrM2MWifiMonitorModeCtrl structure holding the filtering parameters.
[in] pu8PayloadBuffer	Pointer to a buffer allocated by the application. The buffer SHALL hold the Data field of the WIFI RX Packet (Or a part from it). If it is set to NULL, the WIFI data payload will be discarded by the monitoring driver.
[in] u16BufferSize	The total size of the pu8PayloadBuffer in bytes.
[in] u16DataOffset	Starting offset in the DATA FIELD of the received WIFI packet. The application may be interested in reading specific information from the received packet. It must assign the offset to the starting position of it relative to the DATA payload start. <i>Example, if the SSID is needed to be read from a PROBE REQ packet, the u16Offset MUST be set to 0.</i>

Warning

When This mode is enabled, you can not be connected in any mode (Station, Access Point, or P2P).

See also

[tstrM2MWifiMonitorModeCtrl](#) [tstrM2MWifiRxPacketInfo](#) [tstrWifiInitParam](#)
[tenuM2mScanCh](#) [m2m_wifi_disable_monitoring_mode](#) [m2m_wifi_send_wlan_pkt](#)
[m2m_wifi_send_ethernet_pkt](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The example demonstrates the main function where-by the monitoring enable function is called after the initialization of the driver and the packets are handled in the callback function.

```

#include "m2m_wifi.h"
#include "m2m_types.h"

//Declare receive buffer
uint8 gmgmt[1600];

//Callback functions
void wifi_cb(uint8 u8WiFiEvent, void * pvMsg)
{
    ;
}
void wifi_monitoring_cb(tstrM2MWifiRxPacketInfo *pstrWifiRxPacket, uint8
                        *pu8Payload, uint16 u16PayloadSize)
{
    if((NULL != pstrWifiRxPacket) && (0 != u16PayloadSize)) {
        if(MANAGEMENT == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("****# MGMT PACKET ****\n");
        } else if(DATA_BASICTYPE == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("****# DATA PACKET ****\n");
        } else if(CONTROL == pstrWifiRxPacket->u8FrameType) {
            M2M_INFO("****# CONTROL PACKET ****\n");
        }
    }
}

int main()
{
    //Register wifi_monitoring_cb
    tstrWifiInitParam param;
    param.pfAppWifiCb = wifi_cb;
    param.pfAppMonCb = wifi_monitoring_cb;

    nm_bsp_init();

    if(!m2m_wifi_init(&param)) {
        //Enable Monitor Mode with filter to receive all data frames on channel
        1
        tstrM2MWifiMonitorModeCtrl strMonitorCtrl = {0};
        strMonitorCtrl.u8ChannelID      = M2M_WIFI_CH_1;
        strMonitorCtrl.u8FrameType     = DATA_BASICTYPE;
        strMonitorCtrl.u8FrameSubtype   = M2M_WIFI_FRAME_SUB_TYPE_ANY;
        //Receive any subtype of data frame
        m2m_wifi_enable_monitoring_mode(&strMonitorCtrl, gmgmt, sizeof(gmgmt),
                                         0);

        while(1) {
            m2m_wifi_handle_events(NULL);
        }
    }
    return 0;
}

```



m2m_wifi_disable_monitoring_mode

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_disable_monitoring_mode (void)

Detailed Description

Synchronous function to disable Wi-Fi monitoring mode (Promiscuous mode). Expected to be called, if the enable monitoring mode is set, but if it was called without enabling no negative impact will reside.

Function Documentation

- ◆ [m2m_wifi_disable_monitoring_mode\(\)](#)

NMI_API sint8 m2m_wifi_disable_monitoring_mode (void)

See also

[m2m_wifi_enable_monitoring_mode](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_send_wlan_pkt

WLAN » Function

Functions

Functions

```
NMI_API sint8 m2m_wifi_send_wlan_pkt (uint8 *pu8WlanPacket, uint16  
u16WlanHeaderLength, uint16 u16WlanPktSize)
```

Detailed Description

Synchronous function to transmit a WIFI RAW packet while the implementation of this packet is left to the application developer.

Function Documentation

♦ m2m_wifi_send_wlan_pkt()

```
NMI_API sint8 m2m_wifi_send_wlan_pkt ( uint8 * pu8WlanPacket,  
                                         uint16 u16WlanHeaderLength,  
                                         uint16 u16WlanPktSize  
                                         )
```

Parameters

[in] **pu8WlanPacket** Pointer to a buffer holding the whole WIFI frame.

[in] **u16WlanHeaderLength** The size of the WIFI packet header ONLY.

[in] **u16WlanPktSize** The size of the whole bytes in packet.

See also

[m2m_wifi_enable_monitoring_mode](#) [m2m_wifi_disable_monitoring_mode](#)

Precondition

Enable Monitoring mode first using [m2m_wifi_enable_monitoring_mode](#)

Warning

This function available in monitoring mode ONLY.

Note

Packets are user's responsibility.

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_send_ethernet_pkt

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_send_ethernet_pkt (uint8 *pu8Packet, uint16 u16PacketSize)

Detailed Description

Synchronous function to transmit an Ethernet packet. Transmit a packet directly in ETHERNET/bypass mode where the TCP/IP stack is disabled and the implementation of this packet is left to the application developer. The Ethernet packet composition is left to the application developer.

Function Documentation

• m2m_wifi_send_ethernet_pkt()

```
NMI_API sint8 m2m_wifi_send_ethernet_pkt ( uint8 * pu8Packet,  
                                            uint16 u16PacketSize  
                                         )
```

Parameters

[in] **pu8Packet** Pointer to a buffer holding the whole Ethernet frame.

[in] **u16PacketSize** The size of the whole bytes in packet.

Warning

This function available in ETHERNET/Bypass mode ONLY. Make sure that application defines ETH_MODE.

Note

Packets are the user's responsibility.

See also

`m2m_wifi_enable_mac_mcast,m2m_wifi_set_receive_buffer`

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by





m2m_wifi_enable_sntp

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_enable_sntp (uint8 bEnable)

Detailed Description

Synchronous function to enable/disable the native Simple Network Time Protocol(SNTP) client in the WINC1500 firmware.

The SNTP is enabled by default at start-up. The SNTP client at firmware is used to synchronize the system clock to the UTC time from the well known time servers (e.g. "time-c.nist.gov"). The SNTP client uses a default update cycle of 1 day. The UTC is important for checking the expiration date of X509 certificates used while establishing TLS (Transport Layer Security) connections. It is highly recommended to use it if there is no other means to get the UTC time. If there is a RTC on the host MCU, the SNTP could be disabled and the host should set the system time to the firmware using the m2m_wifi_set_system_time function.

Function Documentation

- ◆ m2m_wifi_enable_sntp()

NMI_API sint8 m2m_wifi_enable_sntp (uint8 bEnable)

Parameters

[in] **bEnable** Enabling/Disabling flag '0' :disable SNTP '1' :enable SNTP

See also

[m2m_wifi_set_system_time](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_set_system_time

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_set_system_time (uint32 u32UTCSeconds)

Detailed Description

Synchronous function for setting the system time in time/date format (**uint32**).

The **tstrSystemTime** structure can be used as a reference to the time values that should be set and pass its value as **uint32**

Function Documentation

◆ m2m_wifi_set_system_time()

NMI_API sint8 m2m_wifi_set_system_time (uint32 u32UTCSeconds)

Parameters

[in] **u32UTCSeconds** Seconds elapsed since January 1, 1900 (NTP Timestamp).

See also

[m2m_wifi_enable_sntp tstrSystemTime](#)

Note

If there is an RTC on the host MCU, the SNTP could be disabled and the host should set the system time to the firmware using the API [m2m_wifi_set_system_time](#).

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



m2m_wifi_get_system_time

WLAN » Function

Functions

Functions

NMI_API sint8 m2m_wifi_get_system_time (void)

Detailed Description

Asynchronous function used to retrieve the system time through the use of the response **M2M_WIFI_RESP_GET_SYS_TIME**. Response time retrieved is parsed into the members defined in the structure **tstrSystemTime**.

Function Documentation

◆ m2m_wifi_get_system_time()

NMI_API sint8 m2m_wifi_get_system_time (void)

See also

[m2m_wifi_enable_sntp](#) [tstrSystemTime](#)

Note

Get the system time from the SNTP client using the API [m2m_wifi_get_system_time](#).

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



Functions

m2m_wifi_set_cust_InfoElement

WLAN » Function

Functions

NMI_API sint8 m2m_wifi_set_cust_InfoElement (uint8 *pau8M2mCustInfoElement)

Detailed Description

Synchronous function to Add/Remove user-defined Information Element to the WIFI Beacon and Probe Response frames while chip mode is Access Point Mode.

According to the information element layout shown below, if it is required to set new data for the information elements, pass in the buffer with the information according to the sizes and ordering defined below. However, if it's required to delete these IEs, fill the buffer with zeros.

Function Documentation

◆ m2m_wifi_set_cust_InfoElement()

NMI_API sint8 m2m_wifi_set_cust_InfoElement (uint8 * pau8M2mCustInfoElement)

Parameters

[in] **pau8M2mCustInfoElement** Pointer to Buffer containing the IE to be sent. It is the ap ordering passed in.

Warning

- Size of All elements combined must not exceed 255 byte.
 - Used in Access Point Mode

Note

IEs Format will be follow the following layout:

	Byte[0]	Byte[1]	Byte[2]	Byte[3:1e]
*	#of all Bytes	IE1 ID	Length1	Data1(Hex)

See also

[m2m_wifi_enable_sntp](#) [tstrSystemTime](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Example

The example demonstrates how the information elements are set using this function.

```
char elementData[21];
static char state = 0; // To Add, Append, and Delete
if(0 == state) { //Add 3 IEs
    state = 1;
//Total Number of Bytes
    elementData[0]=12;
//First IE
    elementData[1]=200; elementData[2]=1; elementData[3]='A';
//Second IE
    elementData[4]=201; elementData[5]=2; elementData[6]='B'; elementData[7]=
//Third IE
    elementData[8]=202; elementData[9]=3; elementData[10]='D'; elementData[11]
} else if(1 == state) {
//Append 2 IEs to others, Notice that we keep old data in array starting with
//element 13 and total number of bytes increased to 20
    state = 2;
//Total Number of Bytes
    elementData[0]=20;
//Fourth IE
    elementData[13]=203; elementData[14]=1; elementData[15]='G';
//Fifth IE
    elementData[16]=204; elementData[17]=3; elementData[18]='X'; elementData[19]
} else if(2 == state) { //Delete All IEs
    state = 0;
//Total Number of Bytes
    elementData[0]=0;
}
m2m_wifi_set_cust_InfoElement(elementData);
```



m2m_wifi_set_power_profile

Functions

WLAN » Function

Functions

`sint8 m2m_wifi_set_power_profile (uint8 u8PwrMode)`

Detailed Description

Change the power profile mode

Function Documentation

◆ `m2m_wifi_set_power_profile()`

NMI_API `sint8 m2m_wifi_set_power_profile (uint8 u8PwrMode)`

Parameters

[in] `u8PwrMode` Change the WINC1500 power profile to different mode based on the enumeration `tenuM2mPwrMode`

Precondition

Must be called after the initializations and before any connection request and can't be changed in run time.

See also

`tenuM2mPwrMode m2m_wifi_init`

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_set_tx_power

WLAN » Function

Functions

Functions

`sint8 m2m_wifi_set_tx_power (uint8 u8TxPwrLevel)`

Detailed Description

Set the TX power tenuM2mTxPwrLevel

Function Documentation

◆ m2m_wifi_set_tx_power()

NMI_API `sint8 m2m_wifi_set_tx_power (uint8 u8TxPwrLevel)`

Parameters

[in] **u8TxPwrLevel** change the TX power based on the enumeration
tenuM2mTxPwrLevel

Precondition

Must be called after the initialization and before any connection request and can't be changed in runtime.

See also

`tenuM2mTxPwrLevel m2m_wifi_init`

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_wifi_enable_firmware_logs

WLAN » Function

Functions

Functions

`sint8 m2m_wifi_enable_firmware_logs (uint8 u8Enable)`

Detailed Description

Enable or Disable logs in run time (Disabling Firmware logs will enhance the firmware start-up time and performance)

Function Documentation

◆ `m2m_wifi_enable_firmware_logs()`

NMI_API `sint8 m2m_wifi_enable_firmware_logs (uint8 u8Enable)`

Parameters

[in] `u8Enable` Set 1 to enable the logs, 0 for disable

Precondition

Must be called after initialization through the following function `m2m_wifi_init`

See also

`DISABLE_FIRMWARE_LOGS` (build option to disable logs from initializations)

m2m_wifi_init

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Functions

m2m_wifi_set_battery_voltage

WLAN » Function

Functions

```
sint8 m2m_wifi_set_battery_voltage (uint16 u16BattVoltx100)
```

Detailed Description

Set the battery voltage to update the firmware calculations

Function Documentation

- ◆ [m2m_wifi_set_battery_voltage\(\)](#)

```
sint8 m2m_wifi_set_battery_voltage ( uint16 u16BattVoltx100 )
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by 1.8.13



m2m_wifi_set_gains

WLAN » Function

Functions

Functions

```
sint8 m2m_wifi_set_gains (tstrM2mWifiGainsParams *pstrM2mGain)
```

Detailed Description

Set the chip gains mainly (PPA for 11b/11gn)

Function Documentation

◆ m2m_wifi_set_gains()

```
sint8 m2m_wifi_set_gains ( tstrM2mWifiGainsParams * pstrM2mGain )
```

Set the chip PPA gain for 11b/11gn.

Parameters

[in] **pstrM2mGain** **tstrM2mWifiGainsParams** contain gain parameters as implemented in rf document

Precondition

Must be called after initialization through the following function **m2m_wifi_init**

See also

[m2m_wifi_init](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_get_firmware_version

WLAN » Function

Functions

Functions

`sint8 m2m_wifi_get_firmware_version (tstrM2mRev *pstrRev)`

Detailed Description

Get Firmware version info as defined in the structure tstrM2mRev.

Function Documentation

◆ `m2m_wifi_get_firmware_version()`

`m2m_wifi_get_firmware_version (tstrM2mRev * pstrRev)`

Parameters

[out] **M2mRev** Pointer to the structure tstrM2mRev that contains the firmware version parameters

Precondition

Must be called after initialization through the following function `m2m_wifi_init`

See also

`m2m_wifi_init`

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.



m2m_wifi_prng_get_random_bytes

WLAN » Function

Functions

Functions

```
sint8 m2m_wifi_prng_get_random_bytes (uint8 *pu8PrngBuff, uint16 u16PrngSize)
```

Detailed Description

Asynchronous function for retrieving from the firmware a pseudo-random set of bytes as specified in the size passed in as a parameter. The registered wifi-cb function retrieves the random bytes through the response [M2M_WIFI_RESP_GET_PRNG](#)

Function Documentation

◆ m2m_wifi_prng_get_random_bytes()

```
sint8 m2m_wifi_prng_get_random_bytes ( uint8 * pu8PRNGBuff,  
                                         uint16 u16PRNGSize  
                                       )
```

Parameters

[out] **pu8PrngBuff** Pointer to a buffer to receive data.

[in] **u16PrngSize** Request size in bytes

Warning

Size greater than the maximum specified (`M2M_BUFFER_MAX_SIZE - sizeof(tstrPrng)`) causes a negative error `M2M_ERR_FAIL`.

See also

[tstrPrng](#)

Returns

The function returns `M2M_SUCCESS` for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

[Data Structures](#) | [Typedefs](#) | [Enumerations](#)

Data Types

WLAN

Data Structures

struct **tstrM2mPwrMode**

struct **tstrM2mTxPwrLevel**

struct **tstrM2mEnableLogs**

struct **tstrM2mBatteryVoltage**

struct **tstrM2mWifiGainsParams**

struct **tstrM2mWifiWepParams**

struct **tstr1xAuthCredentials**

union **tuniM2MWifiAuth**

struct **tstrM2MWifiSecInfo**

struct **tstrM2mWifiConnect**

struct **tstrM2MWPSConnect**

struct **tstrM2MWPSInfo**

struct **tstrM2MDefaultConnResp**

struct **tstrM2MScanOption**

struct **tstrM2MScanRegion**

struct **tstrM2MScan**

struct **tstrCryptoResp**

struct **tstrM2mScanDone**

struct **tstrM2mReqScanResult**

struct **tstrM2mWifiscanResult**

struct **tstrM2mWifiStateChanged**

struct **tstrM2mPsType**

struct **tstrM2mSlpReqTime**

struct **tstrM2mLsnInt**

struct **tstrM2MWifiMonitorModeCtrl**

struct **tstrM2MWifiRxPacketInfo**

struct **tstrM2MWifiTxPacketInfo**

struct **tstrM2MP2PConnect**

struct **tstrM2MAPConfig**

struct **tstrM2mServerInit**

struct **tstrM2mClientState**

struct **tstrM2Mservercmd**

struct **tstrM2mSetMacAddress**

struct **tstrM2MDeviceNameConfig**

struct **tstrM2MIPConfig**

struct **tstrM2mIpRsvdPkt**

struct **tstrM2MProvisionModeConfig**

struct **tstrM2MProvisionInfo**

```
struct tstrM2MConnInfo
```

```
struct tstrOtaInitHdr
```

```
struct tstrOtaControlSec
```

```
struct tstrOtaUpdateStatusResp
```

```
struct tstrOtaUpdateInfo
```

```
struct tstrSystemTime
```

```
struct tstrM2MMulticastMac
```

```
struct tstrPrng
```

```
struct tstrTlsCrlEntry
```

```
struct tstrTlsCrlInfo
```

```
struct tstrTlsSrvSecFileEntry
```

```
struct tstrTlsSrvSecHdr
```

```
struct tstrSslSetActiveCsList
```

```
struct tstrM2mPwrState
```

Typedefs

```
typedef void(* tpfOtaNotifCb) (tstrOtaUpdateInfo *pstrOtaUpdateInfo)
```

```
typedef void(* tpfOtaUpdateCb) (uint8 u8OtaUpdateStatusType, uint8 u8OtaUpdateStatus)
```

Enumerations

```
enum tenuOtaError {
    OTA_SUCCESS = (0), OTA_ERR_WORKING_IMAGE_LOAD_FAIL = ((sint8) -1),
    OTA_ERR_INVALID_CONTROL_SEC = ((sint8) -2), M2M_ERR_OTA_SWITCH_FAIL
    = ((sint8) -3),
    M2M_ERR_OTA_START_UPDATE_FAIL = ((sint8) -4),
    M2M_ERR_OTA_ROLLBACK_FAIL = ((sint8) -5),
    M2M_ERR_OTA_INVALID_FLASH_SIZE = ((sint8) -6),
    M2M_ERR_OTA_INVALID_ARG = ((sint8) -7),
    M2M_ERR_OTA_INPROGRESS = ((sint8) -8)
}
```

```

    }

enum tenuM2mConnChangedErrcode {
    M2M_ERR_SCAN_FAIL = ((uint8)1), M2M_ERR_JOIN_FAIL,
    M2M_ERR_AUTH_FAIL, M2M_ERR_ASSOC_FAIL,
    M2M_ERR_CONN_INPROGRESS
}

enum tenuM2mWepKeyIndex { M2M_WIFI_WEP_KEY_INDEX_1 = ((uint8) 1),
    M2M_WIFI_WEP_KEY_INDEX_2, M2M_WIFI_WEP_KEY_INDEX_3,
    M2M_WIFI_WEP_KEY_INDEX_4 }

enum tenuM2mPwrMode { PWR_AUTO = ((uint8) 1), PWR_LOW1, PWR_LOW2,
    PWR_HIGH }

enum tenuM2mTxPwrLevel { TX_PWR_HIGH = ((uint8) 1), TX_PWR_MED, TX_PWR_LOW
}

enum tenuM2mReqGroup {
    M2M_REQ_GROUP_MAIN = 0, M2M_REQ_GROUP_WIFI, M2M_REQ_GROUP_IP,
    M2M_REQ_GROUP_HIF,
    M2M_REQ_GROUP_OTA, M2M_REQ_GROUP_SSL,
    M2M_REQ_GROUP_CRYPTO, M2M_REQ_GROUP_SIGMA
}

enum tenuM2mReqpkt { M2M_REQ_CONFIG_PKT, M2M_REQ_DATA_PKT = 0x80 }

enum tenuM2mConfigCmd {
    M2M_WIFI_REQ_RESTART = M2M_CONFIG_CMD_BASE,
    M2M_WIFI_REQ_SET_MAC_ADDRESS, M2M_WIFI_REQ_CURRENT_RSSI,
    M2M_WIFI_RESP_CURRENT_RSSI,
    M2M_WIFI_REQ_GET_CONN_INFO, M2M_WIFI_RESP_CONN_INFO,
    M2M_WIFI_REQ_SET_DEVICE_NAME,
    M2M_WIFI_REQ_START_PROVISION_MODE,
    M2M_WIFI_RESP_PROVISION_INFO, M2M_WIFI_REQ_STOP_PROVISION_MODE,
    M2M_WIFI_REQ_SET_SYS_TIME, M2M_WIFI_REQ_ENABLE_SNTP_CLIENT,
    M2M_WIFI_REQ_DISABLE_SNTP_CLIENT,
    M2M_WIFI_RESP_MEMORY_RECOVER, M2M_WIFI_REQ_CUST_INFO_ELEMENT,
    M2M_WIFI_REQ_SCAN,
    M2M_WIFI_RESP_SCAN_DONE, M2M_WIFI_REQ_SCAN_RESULT,
    M2M_WIFI_RESP_SCAN_RESULT, M2M_WIFI_REQ_SET_SCAN_OPTION,
    M2M_WIFI_REQ_SET_SCAN_REGION, M2M_WIFI_REQ_SET_POWER_PROFILE,
    M2M_WIFI_REQ_SET_TX_POWER, M2M_WIFI_REQ_SET_BATTERY_VOLTAGE,
    M2M_WIFI_REQ_SET_ENABLE_LOGS, M2M_WIFI_REQ_GET_SYS_TIME,
    M2M_WIFI_RESP_GET_SYS_TIME, M2M_WIFI_REQ_SEND_ETHERNET_PACKET,
}

```

```
    M2M_WIFI_RESP_ETHERNET_RX_PACKET, M2M_WIFI_REQ_SET_MAC_MCAST,  
    M2M_WIFI_REQ_GET_PRNG, M2M_WIFI_RESP_GET_PRNG,  
    M2M_WIFI_REQ_SCAN_SSID_LIST, M2M_WIFI_REQ_SET_GAINS,  
    M2M_WIFI_REQ_PASSIVE_SCAN, M2M_WIFI_MAX_CONFIG_ALL  
}
```

```
enum tenuM2mStaCmd {  
    M2M_WIFI_REQ_CONNECT = M2M_STA_CMD_BASE,  
    M2M_WIFI_REQ_DEFAULT_CONNECT, M2M_WIFI_RESP_DEFAULT_CONNECT,  
    M2M_WIFI_REQ_DISCONNECT,  
    M2M_WIFI_RESP_CON_STATE_CHANGED, M2M_WIFI_REQ_SLEEP,  
    M2M_WIFI_REQ_WPS_SCAN, M2M_WIFI_REQ_WPS,  
    M2M_WIFI_REQ_START_WPS, M2M_WIFI_REQ_DISABLE_WPS,  
    M2M_WIFI_REQ_DHCP_CONF, M2M_WIFI_RESP_IP_CONFIGURED,  
    M2M_WIFI_RESP_IP_CONFLICT, M2M_WIFI_REQ_ENABLE_MONITORING,  
    M2M_WIFI_REQ_DISABLE_MONITORING, M2M_WIFI_RESP_WIFI_RX_PACKET,  
    M2M_WIFI_REQ_SEND_WIFI_PACKET, M2M_WIFI_REQ_LSN_INT,  
    M2M_WIFI_REQ_DOZE, M2M_WIFI_MAX_STA_ALL  
}
```

```
enum tenuM2mApCmd { M2M_WIFI_REQ_ENABLE_AP = M2M_AP_CMD_BASE,  
    M2M_WIFI_REQ_DISABLE_AP, M2M_WIFI_REQ_RESTART_AP,  
    M2M_WIFI_MAX_AP_ALL }
```

```
enum tenuM2mP2pCmd {  
    M2M_WIFI_REQ_P2P_INT_CONNECT = M2M_P2P_CMD_BASE,  
    M2M_WIFI_REQ_ENABLE_P2P, M2M_WIFI_REQ_DISABLE_P2P,  
    M2M_WIFI_REQ_P2P_REPOST,  
    M2M_WIFI_MAX_P2P_ALL  
}
```

```
enum tenuM2mServerCmd { M2M_WIFI_REQ_CLIENT_CTRL =  
    M2M_SERVER_CMD_BASE, M2M_WIFI_RESP_CLIENT_INFO,  
    M2M_WIFI_REQ_SERVER_INIT, M2M_WIFI_MAX_SERVER_ALL }
```

```
enum tenuM2mOtaCmd {  
    M2M_OTA_REQ_NOTIF_SET_URL = M2M_OTA_CMD_BASE,  
    M2M_OTA_REQ_NOTIF_CHECK_FOR_UPDATE, M2M_OTA_REQ_NOTIF_SCHED,  
    M2M_OTA_REQ_START_FW_UPDATE,  
    M2M_OTA_REQ_SWITCH_FIRMWARE, M2M_OTA_REQ_ROLLBACK_FW,  
    M2M_OTA_RESP_NOTIF_UPDATE_INFO, M2M_OTA_RESP_UPDATE_STATUS,  
    M2M_OTA_REQ_TEST, M2M_OTA_REQ_START_CRT_UPDATE,  
    M2M_OTA_REQ_SWITCH_CRT_IMG, M2M_OTA_REQ_ROLLBACK_CRT,  
    M2M_OTA_REQ_ABORT, M2M_OTA_MAX_ALL  
}
```

```

enum tenuM2mCryptoCmd {
    M2M_CRYPTO_REQ_SHA256_INIT = M2M_CRYPTO_CMD_BASE,
    M2M_CRYPTO_RESP_SHA256_INIT, M2M_CRYPTO_REQ_SHA256_UPDATE,
    M2M_CRYPTO_RESP_SHA256_UPDATE,
    M2M_CRYPTO_REQ_SHA256_FINSIH, M2M_CRYPTO_RESP_SHA256_FINSIH,
    M2M_CRYPTO_REQ_RSA_SIGN_GEN, M2M_CRYPTO_RESP_RSA_SIGN_GEN,
    M2M_CRYPTO_REQ_RSA_SIGN_VERIFY,
    M2M_CRYPTO_RESP_RSA_SIGN_VERIFY, M2M_CRYPTO_MAX_ALL
}

enum tenuM2mIpCmd { M2M_IP_REQ_STATIC_IP_CONF = ((uint8) 10),
    M2M_IP_REQ_ENABLE_DHCP, M2M_IP_REQ_DISABLE_DHCP }

enum tenuM2mSigmaCmd {
    M2M_SIGMA_ENABLE = ((uint8) 3), M2M_SIGMA_TA_START,
    M2M_SIGMA_TA_STATS, M2M_SIGMA_TA_RECEIVE_STOP,
    M2M_SIGMA_ICMP_ARP, M2M_SIGMA_ICMP_RX, M2M_SIGMA_ICMP_TX,
    M2M_SIGMA_UDP_TX,
    M2M_SIGMA_UDP_TX_DEFER, M2M_SIGMA_SECURITY_POLICY,
    M2M_SIGMA_SET_SYSTIME
}

enum tenuM2mSslCmd {
    M2M_SSL_REQ_CERT_VERIF, M2M_SSL_REQ_ECC, M2M_SSL_RESP_ECC,
    M2M_SSL_IND_CRL,
    M2M_SSL_IND_CERTS_ECC, M2M_SSL_REQ_SET_CS_LIST,
    M2M_SSL_RESP_SET_CS_LIST
}

enum tenuM2mConnState { M2M_WIFI_DISCONNECTED = 0, M2M_WIFI_CONNECTED,
    M2M_WIFI_UNDEF = 0xff }

enum tenuM2mSecType {
    M2M_WIFI_SEC_INVALID = 0, M2M_WIFI_SEC_OPEN,
    M2M_WIFI_SEC_WPA_PSK, M2M_WIFI_SEC_WEP,
    M2M_WIFI_SEC_802_1X
}

enum tenuM2mSsidMode { SSID_MODE_VISIBLE = 0, SSID_MODE_HIDDEN }

enum tenuM2mScanCh {
    M2M_WIFI_CH_1 = ((uint8) 1), M2M_WIFI_CH_2, M2M_WIFI_CH_3,
    M2M_WIFI_CH_4,
    M2M_WIFI_CH_5, M2M_WIFI_CH_6, M2M_WIFI_CH_7, M2M_WIFI_CH_8,
}

```

```
    M2M_WIFI_CH_9, M2M_WIFI_CH_10, M2M_WIFI_CH_11, M2M_WIFI_CH_12,  
    M2M_WIFI_CH_13, M2M_WIFI_CH_14, M2M_WIFI_CH_ALL = ((uint8) 255)  
}
```

```
enum tenuM2mScanRegion {  
    REG_CH_1 = ((uint16) 1 << 0), REG_CH_2 = ((uint16) 1 << 1), REG_CH_3 = ((uint16)  
    1 << 2), REG_CH_4 = ((uint16) 1 << 3),  
    REG_CH_5 = ((uint16) 1 << 4), REG_CH_6 = ((uint16) 1 << 5), REG_CH_7 = ((uint16)  
    1 << 6), REG_CH_8 = ((uint16) 1 << 7),  
    REG_CH_9 = ((uint16) 1 << 8), REG_CH_10 = ((uint16) 1 << 9), REG_CH_11 =  
    ((uint16) 1 << 10), REG_CH_12 = ((uint16) 1 << 11),  
    REG_CH_13 = ((uint16) 1 << 12), REG_CH_14 = ((uint16) 1 << 13), REG_CH_ALL =  
    ((uint16) 0x3FFF), NORTH_AMERICA = ((uint16) 0x7FF),  
    EUROPE = ((uint16) 0x1FFF), ASIA = ((uint16) 0x3FFF)  
}
```

```
enum tenuPowerSaveModes {  
    M2M_NO_PS, M2M_PS_AUTOMATIC, M2M_PS_H_AUTOMATIC,  
    M2M_PS_DEEP_AUTOMATIC,  
    M2M_PS_MANUAL  
}
```

```
enum tenuM2mWifiMode {  
    M2M_WIFI_MODE_NORMAL = ((uint8) 1), M2M_WIFI_MODE_ATE_HIGH,  
    M2M_WIFI_MODE_ATE_LOW, M2M_WIFI_MODE_ETHERNET,  
    M2M_WIFI_MODE_MAX  
}
```

```
enum tenuWPSTrigger { WPS_PIN_TRIGGER = 0, WPS_PBC_TRIGGER = 4 }
```

```
enum tenuOtaUpdateStatus {  
    OTA_STATUS_SUCSESS = 0, OTA_STATUS_FAIL = 1,  
    OTA_STATUS_INVALID_ARG = 2, OTA_STATUS_INVALID_RB_IMAGE = 3,  
    OTA_STATUS_INVALID_FLASH_SIZE = 4, OTA_STATUS_AREADY_ENABLED =  
    5, OTA_STATUS_UPDATE_INPROGRESS = 6,  
    OTA_STATUS_IMAGE_VERIF_FAILED = 7,  
    OTA_STATUS_CONNECTION_ERROR = 8, OTA_STATUS_SERVER_ERROR = 9,  
    OTA_STATUS_ABORTED = 10  
}
```

```
enum tenuOtaUpdateStatusType { DL_STATUS = 1, SW_STATUS = 2, RB_STATUS = 3,  
    AB_STATUS = 4 }
```

```
enum tenuSslCertExpSettings { SSL_CERT_EXP_CHECK_DISABLE,  
    SSL_CERT_EXP_CHECK_ENABLE, SSL_CERT_EXP_CHECK_EN_IF_SYS_TIME }
```

Detailed Description

Typedef Documentation

- ◆ **tpfOtaNotifCb**

```
void(* tpfOtaNotifCb)(tstrOtaUpdateInfo *)
```

A callback to get notification about a potential OTA update.

Parameters

[in] **pstrOtaUpdateInfo** A structure to provide notification payload.

See also

[tstrOtaUpdateInfo](#)

Warning

The notification is not supported (Not implemented yet)

- ◆ **tpfOtaUpdateCb**

```
void(* tpfOtaUpdateCb)(uint8 u8OtaUpdateStatusType,uint8 u8OtaUpdateStatus)
```

A callback to get OTA status update, the callback provide the status type and its status. The OTA callback provides the download status, the switch to the downloaded firmware status and roll-back status.

Parameters

[in] **u8OtaUpdateStatusType** Possible values are listed in [tenuOtaUpdateStatusType](#).

[in] **u8OtaUpdateStatus** Possible values are listed as enumerated by [tenuOtaUpdateStatus](#).

See also

[tenuOtaUpdateStatusType](#) [tenuOtaUpdateStatus](#)

Enumeration Type Documentation

◆ tenuOtaError

enum **tenuOtaError**

Enumerator	
OTA_SUCCESS	OTA Success status
OTA_ERR_WORKING_IMAGE_LOAD_FAIL	Failure to load the firmware image
OTA_ERR_INVALID_CONTROL_SEC	Control structure is being corrupted
M2M_ERR_OTA_SWITCH_FAIL	switching to the updated image failed as may be the image is invalid
M2M_ERR_OTA_START_UPDATE_FAIL	OTA update fail due to multiple reasons <ul style="list-style-type: none">• Connection failure• Image integrity fail
M2M_ERR_OTA_ROLLBACK_FAIL	Roll-back failed due to Roll-back image is not valid
M2M_ERR_OTA_INVALID_FLASH_SIZE	The OTA Support at least 4MB flash size, if the above error will appear if the current flash is less than 4M
M2M_ERR_OTA_INVALID_ARG	Ota still in progress
M2M_ERR_OTA_INPROGRESS	Invalid argument in any OTA Function

◆ tenuM2mConnChangedErrcode

enum **tenuM2mConnChangedErrcode**

Enumerator	
M2M_ERR_SCAN_FAIL	Indicate that the WINC board has failed to perform the scan operation.
M2M_ERR_JOIN_FAIL	Indicate that the WINC board has failed to join the BSS .
M2M_ERR_AUTH_FAIL	Indicate that the WINC board has failed to authenticate with the AP.
M2M_ERR_ASSOC_FAIL	Indicate that the WINC board has failed to associate

	with the AP.
M2M_ERR_CONN_INPROGRESS	Indicate that the WINC board has another connection request in progress.

◆ tenuM2mWepKeyIndex

enum **tenuM2mWepKeyIndex**

Enumerator	
M2M_WIFI_WEP_KEY_INDEX_1	
M2M_WIFI_WEP_KEY_INDEX_2	
M2M_WIFI_WEP_KEY_INDEX_3	
M2M_WIFI_WEP_KEY_INDEX_4	Index for WEP key Authentication

◆ tenuM2mPwrMode

enum **tenuM2mPwrMode**

Enumerator	
PWR_AUTO	FW will decide the best power mode to use internally.
PWR_LOW1	
PWR_LOW2	
PWR_HIGH	

◆ tenuM2mTxPwrLevel

enum **tenuM2mTxPwrLevel**

Enumerator	
TX_PWR_HIGH	PPA Gain 6dbm PA Gain 18dbm
TX_PWR_MED	PPA Gain 6dbm PA Gain 12dbm
TX_PWR_LOW	PPA Gain 6dbm PA Gain 6dbm

◆ tenuM2mReqGroup

enum tenuM2mReqGroup

Enumerator	
M2M_REQ_GROUP_MAIN	
M2M_REQ_GROUP_WIFI	
M2M_REQ_GROUP_IP	
M2M_REQ_GROUP_HIF	
M2M_REQ_GROUP_OTA	
M2M_REQ_GROUP_SSL	
M2M_REQ_GROUP_CRYPTO	
M2M_REQ_GROUP_SIGMA	

◆ tenuM2mReqpkt

enum tenuM2mReqpkt

Enumerator	
M2M_REQ_CONFIG_PKT	
M2M_REQ_DATA_PKT	

◆ tenuM2mConfigCmd

enum tenuM2mConfigCmd

This enum contains all the host commands used to configure the WINC board.

Enumerator	
M2M_WIFI_REQ_RESTART	Restart the WINC MAC layer, it's doesn't restart the IP layer.
M2M_WIFI_REQ_SET_MAC_ADDRESS	Set the WINC mac address (not possible for production effused boards).
M2M_WIFI_REQ_CURRENT_RSSI	Request the current connected AP RSSI.
M2M_WIFI_RESP_CURRENT_RSSI	Response to M2M_WIFI_REQ_CURRENT_RSSI with the RSSI value.

M2M_WIFI_REQ_GET_CONN_INFO	Request connection information command.
M2M_WIFI_RESP_CONN_INFO	Connect with default AP response.
M2M_WIFI_REQ_SET_DEVICE_NAME	Set the WINC device name property.
M2M_WIFI_REQ_START_PROVISION_MODE	Start the provisioning mode for the M2M Device.
M2M_WIFI_RESP_PROVISION_INFO	Send the provisioning information to the host.
M2M_WIFI_REQ_STOP_PROVISION_MODE	Stop the current running provision mode.
M2M_WIFI_REQ_SET_SYS_TIME	Set time of day from host.
M2M_WIFI_REQ_ENABLE_SNTP_CLIENT	Enable the simple network time protocol to get the time from the Internet. this is required for security purposes.
M2M_WIFI_REQ_DISABLE_SNTP_CLIENT	Disable the simple network time protocol for applications that do not need it.
M2M_WIFI_RESP_MEMORY_RECOVER	Reserved for debuging
M2M_WIFI_REQ_CUST_INFO_ELEMENT	Add Custom EElement to Beacon Managment Frame.
M2M_WIFI_REQ_SCAN	Request scan command.
M2M_WIFI_RESP_SCAN_DONE	Scan complete notification response.
M2M_WIFI_REQ_SCAN_RESULT	Request Scan results command.
M2M_WIFI_RESP_SCAN_RESULT	Request Scan results resopnse.
M2M_WIFI_REQ_SET_SCAN_OPTION	Set Scan options "slot time, slot number .. etc" .
M2M_WIFI_REQ_SET_SCAN_REGION	Set scan region.
M2M_WIFI_REQ_SET_POWER_PROFILE	The API shall set power mode to one of 3 modes
M2M_WIFI_REQ_SET_TX_POWER	API to set TX power.
M2M_WIFI_REQ_SET_BATTERY_VOLTAGE	API to set Battery Voltage.
M2M_WIFI_REQ_SET_ENABLE_LOGS	API to set Battery Voltage.
M2M_WIFI_REQ_GET_SYS_TIME	REQ GET time of day from WINC.
M2M_WIFI_RESP_GET_SYS_TIME	RESP time of day from host.
M2M_WIFI_REQ_SEND_ETHERNET_PACKET	Send Ethernet packet in bypass mode.

M2M_WIFI_RESP_ETHERNET_RX_PACKET	Receive Ethernet packet in bypass mode.
M2M_WIFI_REQ_SET_MAC_MCAST	Set the WINC multicast filters.
M2M_WIFI_REQ_GET_PRNG	Request PRNG.
M2M_WIFI_RESP_GET_PRNG	Response for PRNG.
M2M_WIFI_REQ_SCAN_SSID_LIST	Request scan with list of hidden SSID plus the broadcast scan.
M2M_WIFI_REQ_SET_GAINS	Request set the PPA gain
M2M_WIFI_REQ_PASSIVE_SCAN	Request a passivr scan command.
M2M_WIFI_MAX_CONFIG_ALL	

◆ tenuM2mStaCmd

enum tenuM2mStaCmd

This enum contains all the WINC commands while in Station mode.

Enumerator	
M2M_WIFI_REQ_CONNECT	Connect with AP command.
M2M_WIFI_REQ_DEFAULT_CONNECT	Connect with default AP command.
M2M_WIFI_RESP_DEFAULT_CONNECT	Request connection information response.
M2M_WIFI_REQ_DISCONNECT	Request to disconnect from AP command.
M2M_WIFI_RESP_CON_STATE_CHANGED	Connection state changed response.
M2M_WIFI_REQ_SLEEP	Set PS mode command.
M2M_WIFI_REQ_WPS_SCAN	Request WPS scan command.
M2M_WIFI_REQ_WPS	Request WPS start command.
M2M_WIFI_REQ_START_WPS	This command is for internal use by the WINC and should not be used by the host driver.
M2M_WIFI_REQ_DISABLE_WPS	Request to disable WPS command.
M2M_WIFI_REQ_DHCP_CONF	Response indicating that IP address was obtained.
M2M_WIFI_RESP_IP_CONFIGURED	This command is for internal use by the WINC and should not be used by the host driver.

M2M_WIFI_RESP_IP_CONFLICT	Response indicating a conflict in obtained IP address. The user should re attempt the DHCP request.
M2M_WIFI_REQ_ENABLE_MONITORING	Request to enable monitor mode command.
M2M_WIFI_REQ_DISABLE_MONITORING	Request to disable monitor mode command.
M2M_WIFI_RESP_WIFI_RX_PACKET	Indicate that a packet was received in monitor mode.
M2M_WIFI_REQ_SEND_WIFI_PACKET	Send packet in monitor mode.
M2M_WIFI_REQ_LSN_INT	Set WiFi listen interval.
M2M_WIFI_REQ_DOZE	Used to force the WINC to sleep in manual PS mode.
M2M_WIFI_MAX_STA_ALL	

◆ tenuM2mApCmd

enum tenuM2mApCmd

This enum contains all the WINC commands while in AP mode.

Enumerator	
M2M_WIFI_REQ_ENABLE_AP	Enable AP mode command.
M2M_WIFI_REQ_DISABLE_AP	Disable AP mode command.
M2M_WIFI_REQ_RESTART_AP	
M2M_WIFI_MAX_AP_ALL	

◆ tenuM2mP2pCmd

enum tenuM2mP2pCmd

This enum contains all the WINC commands while in P2P mode.

Enumerator	
M2M_WIFI_REQ_P2P_INT_CONNECT	This command is for internal use by the WINC and should not be used by the host driver.

M2M_WIFI_REQ_ENABLE_P2P	Enable P2P mode command.
M2M_WIFI_REQ_DISABLE_P2P	Disable P2P mode command.
M2M_WIFI_REQ_P2P_REPOST	This command is for internal use by the WINC and should not be used by the host driver.
M2M_WIFI_MAX_P2P_ALL	

◆ tenuM2mServerCmd

enum tenuM2mServerCmd

This enum contains all the WINC commands while in PS mode. These command are currently not supported.

Enumerator	
M2M_WIFI_REQ_CLIENT_CTRL	
M2M_WIFI_RESP_CLIENT_INFO	
M2M_WIFI_REQ_SERVER_INIT	
M2M_WIFI_MAX_SERVER_ALL	

◆ tenuM2mOtaCmd

enum tenuM2mOtaCmd

Enumerator	
M2M_OTA_REQ_NOTIF_SET_URL	
M2M_OTA_REQ_NOTIF_CHECK_FOR_UPDATE	
M2M_OTA_REQ_NOTIF_SCHED	
M2M_OTA_REQ_START_FW_UPDATE	
M2M_OTA_REQ_SWITCH_FIRMWARE	
M2M_OTA_REQ_ROLLBACK_FW	
M2M_OTA_RESP_NOTIF_UPDATE_INFO	
M2M_OTA_RESP_UPDATE_STATUS	
M2M_OTA_REQ_TEST	
M2M_OTA_REQ_START_CRT_UPDATE	
M2M_OTA_REQ_SWITCH_CRT_IMG	

M2M_OTA_REQ_ROLLBACK_CRT
M2M_OTA_REQ_ABORT
M2M_OTA_MAX_ALL

◆ tenuM2mCryptoCmd

enum **tenuM2mCryptoCmd**

Enumerator
M2M_CRYPTO_REQ_SHA256_INIT
M2M_CRYPTO_RESP_SHA256_INIT
M2M_CRYPTO_REQ_SHA256_UPDATE
M2M_CRYPTO_RESP_SHA256_UPDATE
M2M_CRYPTO_REQ_SHA256_FINSIH
M2M_CRYPTO_RESP_SHA256_FINSIH
M2M_CRYPTO_REQ_RSA_SIGN_GEN
M2M_CRYPTO_RESP_RSA_SIGN_GEN
M2M_CRYPTO_REQ_RSA_SIGN_VERIFY
M2M_CRYPTO_RESP_RSA_SIGN_VERIFY
M2M_CRYPTO_MAX_ALL

◆ tenuM2mlpCmd

enum **tenuM2mlpCmd**

Enumerator
M2M_IP_REQ_STATIC_IP_CONF
M2M_IP_REQ_ENABLE_DHCP
M2M_IP_REQ_DISABLE_DHCP

◆ tenuM2mSigmaCmd

enum **tenuM2mSigmaCmd**

Enumerator

M2M_SIGMA_ENABLE
M2M_SIGMA_TA_START
M2M_SIGMA_TA_STATS
M2M_SIGMA_TA_RECEIVE_STOP
M2M_SIGMA_ICMP_ARP
M2M_SIGMA_ICMP_RX
M2M_SIGMA_ICMP_TX
M2M_SIGMA_UDP_TX
M2M_SIGMA_UDP_TX_DEFER
M2M_SIGMA_SECURITY_POLICY
M2M_SIGMA_SET_SYSTIME

◆ tenuM2mSslCmd

enum **tenuM2mSslCmd**

Enumerator
M2M_SSL_REQ_CERT_VERIF
M2M_SSL_REQ_ECC
M2M_SSL_RESP_ECC
M2M_SSL_IND_CRL
M2M_SSL_IND_CERTS_ECC
M2M_SSL_REQ_SET_CS_LIST
M2M_SSL_RESP_SET_CS_LIST

◆ tenuM2mConnState

enum **tenuM2mConnState**

Wi-Fi Connection State.

Enumerator	
M2M_WIFI_DISCONNECTED	Wi-Fi state is disconnected.
M2M_WIFI_CONNECTED	Wi-Fi state is connected.
M2M_WIFI_UNDEF	Undefined Wi-Fi State.

◆ tenuM2mSecType

enum tenuM2mSecType

Wi-Fi Supported Security types.

Wi-Fi Supported SSID types.

Enumerator	
M2M_WIFI_SEC_INVALID	Invalid security type.
M2M_WIFI_SEC_OPEN	Wi-Fi network is not secured.
M2M_WIFI_SEC_WPA_PSK	Wi-Fi network is secured with WPA/WPA2 personal(PSK).
M2M_WIFI_SEC_WEP	Security type WEP (40 or 104) OPEN OR SHARED.
M2M_WIFI_SEC_802_1X	Wi-Fi network is secured with WPA/WPA2 Enterprise.IEEE802.1x user-name/password authentication.

◆ tenuM2mSsidMode

enum tenuM2mSsidMode

Enumerator	
SSID_MODE_VISIBLE	SSID is visible to others.
SSID_MODE_HIDDEN	SSID is hidden.

◆ tenuM2mScanCh

enum tenuM2mScanCh

Wi-Fi RF Channels.

See also

[tstrM2MScan](#) [tstrM2MScanOption](#)

Enumerator	
M2M_WIFI_CH_1	

M2M_WIFI_CH_2
M2M_WIFI_CH_3
M2M_WIFI_CH_4
M2M_WIFI_CH_5
M2M_WIFI_CH_6
M2M_WIFI_CH_7
M2M_WIFI_CH_8
M2M_WIFI_CH_9
M2M_WIFI_CH_10
M2M_WIFI_CH_11
M2M_WIFI_CH_12
M2M_WIFI_CH_13
M2M_WIFI_CH_14
M2M_WIFI_CH_ALL

◆ tenuM2mScanRegion

enum **tenuM2mScanRegion**

Wi-Fi RF Channels.

Enumerator
REG_CH_1
REG_CH_2
REG_CH_3
REG_CH_4
REG_CH_5
REG_CH_6
REG_CH_7
REG_CH_8
REG_CH_9
REG_CH_10
REG_CH_11
REG_CH_12
REG_CH_13

REG_CH_14	
REG_CH_ALL	
NORTH_AMERICA	
EUROPE	11 channel
ASIA	13 channel

◆ tenuPowerSaveModes

enum tenuPowerSaveModes

Power Save Modes.

Enumerator	
M2M_NO_PS	Power save is disabled.
M2M_PS_AUTOMATIC	Power save is done automatically by the WINC. This mode doesn't disable all of the WINC modules and use higher amount of power than the H_AUTOMATIC and the DEEP_AUTOMATIC modes..
M2M_PS_H_AUTOMATIC	Power save is done automatically by the WINC. Achieve higher power save than the AUTOMATIC mode by shutting down more parts of the WINC board.
M2M_PS_DEEP_AUTOMATIC	Power save is done automatically by the WINC. Achieve the highest possible power save.
M2M_PS_MANUAL	Power save is done manually by the user.

◆ tenuM2mWifiMode

enum tenuM2mWifiMode

Wi-Fi Operation Mode.

Enumerator	
M2M_WIFI_MODE_NORMAL	Normal Mode means to run customer firmware version.
M2M_WIFI_MODE_ATE_HIGH	Config Mode in HIGH POWER means to run production test firmware version which is known as ATE (Burst) firmware.

M2M_WIFI_MODE_ATE_LOW	Config Mode in LOW POWER means to run production test firmware version which is known as ATE (Burst) firmware.
M2M_WIFI_MODE_ETHERNET	etherent Mode
M2M_WIFI_MODE_MAX	

◆ tenuWPSTrigger

enum tenuWPSTrigger

WPS Triggering Methods.

Enumerator	
WPS_PIN_TRIGGER	WPS is triggered in PIN method.
WPS_PBC_TRIGGER	WPS is triggered via push button.

◆ tenuOtaUpdateStatus

enum tenuOtaUpdateStatus

OTA return status.

Enumerator	
OTA_STATUS_SUCSESS	OTA Success with not errors.
OTA_STATUS_FAIL	OTA generic fail.
OTA_STATUS_INVALID_ARG	Invalid or malformed download URL.
OTA_STATUS_INVALID_RB_IMAGE	Invalid rollback image.
OTA_STATUS_INVALID_FLASH_SIZE	Flash size on device is not enough for OTA.
OTA_STATUS_AREADY_ENABLED	An OTA operation is already enabled.
OTA_STATUS_UPDATE_INPROGRESS	An OTA operation update is in progress
OTA_STATUS_IMAGE_VERIF_FAILED	OTA Verification failed
OTA_STATUS_CONNECTION_ERROR	OTA connection error
OTA_STATUS_SERVER_ERROR	OTA server Error (file not found or else ...)
OTA_STATUS_ABORTED	OTA download has been aborted by the application.

◆ tenuOtaUpdateStatusType

enum tenuOtaUpdateStatusType

OTA update Status type.

Enumerator	
DL_STATUS	Download OTA file status
SW_STATUS	Switching to the upgrade firmware status
RB_STATUS	Roll-back status
AB_STATUS	Abort status

◆ tenuSslCertExpSettings

enum tenuSslCertExpSettings

SSL Certificate Expiry Validation Options.

Enumerator	
SSL_CERT_EXP_CHECK_DISABLE	ALWAYS OFF. Ignore certificate expiration date validation. If a certificate is expired or there is no configured system time, the SSL connection SUCCEEDs.
SSL_CERT_EXP_CHECK_ENABLE	ALWAYS ON. Validate certificate expiration date. If a certificate is expired or there is no configured system time, the SSL connection FAILs.
SSL_CERT_EXP_CHECK_EN_IF_SYS_TIME	CONDITIONAL VALIDATION (Default setting at startup). Validate the certificate expiration date only if there is a configured system time. If there is no configured system time, the certificate expiration is bypassed and the SSL connection SUCCEEDs.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mPwrMode Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8PwrMode`

`uint8 __PAD24__ [3]`

Field Documentation

◆ `u8PwrMode`

`uint8 u8PwrMode`

power Save Mode

◆ `__PAD24__`

`uint8 __PAD24__[3]`

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mTxPwrLevel Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8TxPwrLevel`

`uint8 __PAD24__ [3]`

Detailed Description

Tx power level.

Field Documentation

◆ `u8TxPwrLevel`

`uint8 u8TxPwrLevel`

Tx power level

◆ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mEnableLogs Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8Enable`

`uint8 __PAD24__ [3]`

Detailed Description

Enable Firmware logs.

Field Documentation

◆ `u8Enable`

`uint8 u8Enable`

Enable/Disable firmware logs

◆ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mBatteryVoltage Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16BattVolt`

`uint8 __PAD16__ [2]`

Detailed Description

Battery Voltage.

Field Documentation

♦ `u16BattVolt`

`uint16 u16BattVolt`

Battery Voltage

♦ `__PAD16__`

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mWifiGainsParams Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u8PPAGFor11B`

`uint16 u8PPAGFor11GN`

Detailed Description

Gain Values.

Field Documentation

◆ `u8PPAGFor11B`

`uint16 u8PPAGFor11B`

PPA gain for 11B (as the RF document representation) PPA_AGC<0:2> Every bit have 3dB gain control each. for example: 1 ->3db 3 ->6db 7 ->9db

◆ `u8PPAGFor11GN`

uint16 u8PPAGFor11GN

PPA gain for 11GN (as the RF document represented) PPA_AGC<0:2> Every bit have 3dB gain control each. for example: 1 ->3db 3 ->6db 7 ->9db

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mWifiWepParams Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8KeyIndx`

`uint8 u8KeySz`

`uint8 au8WepKey [WEP_104_KEY_STRING_SIZE+1]`

`uint8 __PAD24__ [3]`

Detailed Description

WEP security key parameters.

Field Documentation

◆ `u8KeyIndx`

`uint8 u8KeyIndx`

Wep key Index.

- ◆ u8KeySz

```
uint8 u8KeySz
```

Wep key Size.

- ◆ au8WepKey

```
uint8 au8WepKey[WEP_104_KEY_STRING_SIZE+1]
```

WEP Key represented as a NULL terminated ASCII string.

- ◆ __PAD24__

```
uint8 __PAD24__[3]
```

Padding bytes to keep the structure word alligned.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstr1xAuthCredentials Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8UserName [M2M_1X_USR_NAME_MAX]`

`uint8 au8Passwd [M2M_1X_PWD_MAX]`

Detailed Description

Credentials for the user to authenticate with the AAA server (WPA-Enterprise Mode IEEE802.1x).

Field Documentation

◆ au8UserName

`uint8 au8UserName[M2M_1X_USR_NAME_MAX]`

User Name. It must be Null terminated string.

◆ au8Passwd

```
uint8 au8Passwd[M2M_1X_PWD_MAX]
```

Password corresponding to the user name. It must be Null terminated string.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tuniM2MWifiAuth Union Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8PSK [M2M_MAX_PSK_LEN]`

`tstr1xAuthCredentials strCred1x`

`tstrM2mWifiWepParams strWepInfo`

Detailed Description

Wi-Fi Security Parameters for all supported security modes.

Field Documentation

◆ au8PSK

`uint8 au8PSK[M2M_MAX_PSK_LEN]`

Pre-Shared Key in case of WPA-Personal security.

- ◆ strCred1x

```
tstr1xAuthCredentials strCred1x
```

Credentials for RADIUS server authentication in case of WPA-Enterprise security.

- ◆ strWepInfo

```
tstrM2mWifiWepParams strWepInfo
```

WEP key parameters in case of WEP security.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MWifiSecInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`tuniM2MWifiAuth uniAuth`

`uint8 u8SecType`

`uint8 __PAD__ [__PADDING__]`

Detailed Description

Authentication credentials to connect to a Wi-Fi network.

Field Documentation

◆ uniAuth

`tuniM2MWifiAuth uniAuth`

Union holding all possible authentication parameters corresponding the current security types.

- ◆ u8SecType

```
uint8 u8SecType
```

Wi-Fi network security type. See tenuM2mSecType for supported security types.

- ◆ __PAD__

```
uint8 __PAD__[__PADDING__]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mWifiConnect Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`tstrM2MWifiSecInfo strSec`

`uint16 u16Ch`

`uint8 au8SSID [M2M_MAX_SSID_LEN]`

`uint8 u8NoSaveCred`

`uint8 __PAD__ [__CONN_PAD_SIZE__]`

Detailed Description

Wi-Fi Connect Request.

Field Documentation

◆ strSec

`tstrM2MWifiSecInfo strSec`

Security parameters for authenticating with the AP.

- ◆ **u16Ch**

```
uint16 u16Ch
```

RF Channel for the target SSID.

- ◆ **au8SSID**

```
uint8 au8SSID[M2M_MAX_SSID_LEN]
```

SSID of the desired AP. It must be NULL terminated string.

- ◆ **u8NoSaveCred**

```
uint8 u8NoSaveCred
```

- ◆ **_PAD_**

```
uint8 __PAD__[__CONN_PAD_SIZE__]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2WPSCConnect Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8TriggerType`

`char acPinNumber [8]`

`uint8 __PAD24__ [3]`

Detailed Description

WPS Configuration parameters.

See also

[tenuWPSTrigger](#)

Field Documentation

- ◆ `u8TriggerType`

`uint8 u8TriggerType`

WPS triggering method (Push button or PIN)

- ◆ acPinNumber

```
char acPinNumber[8]
```

WPS PIN No (for PIN method)

- ◆ __PAD24__

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MWPSInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8AuthType`

`uint8 u8Ch`

`uint8 au8SSID [M2M_MAX_SSID_LEN]`

`uint8 au8PSK [M2M_MAX_PSK_LEN]`

Detailed Description

WPS Result.

This structure is passed to the application in response to a WPS request. If the WPS session is completed successfully, the structure will have Non-ZERO authentication type. If the WPS Session fails (due to error or timeout) the authentication type is set to ZERO.

See also

[tenuM2mSecType](#)

Field Documentation

- ◆ u8AuthType

```
uint8 u8AuthType
```

Network authentication type.

- ◆ u8Ch

```
uint8 u8Ch
```

RF Channel for the AP.

- ◆ au8SSID

```
uint8 au8SSID[M2M_MAX_SSID_LEN]
```

SSID obtained from WPS.

- ◆ au8PSK

```
uint8 au8PSK[M2M_MAX_PSK_LEN]
```

PSK for the network obtained from WPS.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MDefaultConnResp Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`sint8 s8ErrorCode`

`uint8 __PAD24__ [3]`

Detailed Description

Response error of the m2m_default_connect.

See also

[M2M_DEFAULT_CONN_SCAN_MISMATCH](#) [M2M_DEFAULT_CONN_EMPTY_LIST](#)

Field Documentation

◆ `s8ErrorCode`

`sint8 s8ErrorCode`

Default connect error code. possible values are:

- [M2M_DEFAULT_CONN_EMPTY_LIST](#)

- M2M_DEFAULT_CONN_SCAN_MISMATCH

- ◆ PAD24

```
uint8 __PAD24__[3]
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MScanOption Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8NumOfSlot`

`uint8 u8SlotTime`

`uint8 u8ProbesPerSlot`

`sint8 s8RssiThresh`

Detailed Description

Scan options and configurations.

See also

[tenuM2mScanCh](#) [tstrM2MScan](#)

Field Documentation

◆ `u8NumOfSlot`

`uint8 u8NumOfSlot`

- ◆ u8SlotTime

```
uint8 u8SlotTime
```

- ◆ u8ProbesPerSlot

```
uint8 u8ProbesPerSlot
```

Number of probe requests to be sent per channel scan slot.

- ◆ s8RssiThresh

```
sint8 s8RssiThresh
```



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MScanRegion Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16ScanRegion`

`uint8 __PAD16__ [2]`

Detailed Description

Wi-Fi channel regulation region information.

See also

[tenuM2mScanRegion](#)

Field Documentation

- ◆ `u16ScanRegion`

`uint16 u16ScanRegion`

- ◆ `__PAD16__`

```
uint8 __PAD16__[2]
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  doxygen 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MScan Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8ChNum`

`uint8 __RSVD8 [1]`

`uint16 u16PassiveScanTime`

Detailed Description

Wi-Fi Scan Request.

See also

[tenuM2mScanCh](#) [tstrM2MScanOption](#)

Field Documentation

◆ `u8ChNum`

`uint8 u8ChNum`

The Wi-Fi RF Channel number

- ◆ [__RSVD8__](#)

```
uint8 __RSVD8__[1]
```

Reserved for future use.

- ◆ [u16PassiveScanTime](#)

```
uint16 u16PassiveScanTime
```

Passive Scan Timeout in ms. The field is ignored for active scan.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrCryptoResp Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`sint8 s8Resp`

`uint8 __PAD24__ [3]`

Detailed Description

crypto response

Field Documentation

- ◆ `s8Resp`

`sint8 s8Resp`

- ◆ `__PAD24__`

`uint8 __PAD24__[3]`



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mScanDone Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8NumofCh`

`sint8 s8ScanState`

`uint8 __PAD16__ [2]`

Detailed Description

Wi-Fi Scan Result.

Field Documentation

◆ `u8NumofCh`

`uint8 u8NumofCh`

Number of found APs

- ◆ s8ScanState

```
sint8 s8ScanState
```

Scan status

- ◆ __PAD16__

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mReqScanResult Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8Index`

`uint8 __PAD24__ [3]`

Detailed Description

Scan Result Request.

The Wi-Fi Scan results list is stored in Firmware. The application can request a certain scan result by its index.

Field Documentation

◆ `u8Index`

`uint8 u8Index`

Index of the desired scan result

◆ PAD24

uint8 __PAD24__[3]

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mWifiscanResult Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8index`

`sint8 s8rssI`

`uint8 u8AuthType`

`uint8 u8ch`

`uint8 au8BSSID [6]`

`uint8 au8SSID [M2M_MAX_SSID_LEN]`

`uint8 _PAD8_`

Detailed Description

Wi-Fi Scan Result.

Information corresponding to an AP in the Scan Result list identified by its order (index) in the list.

Field Documentation

- ◆ u8index

```
uint8 u8index
```

AP index in the scan result list.

- ◆ s8rssи

```
sint8 s8rssи
```

AP signal strength.

- ◆ u8AuthType

```
uint8 u8AuthType
```

AP authentication type.

- ◆ u8ch

```
uint8 u8ch
```

AP RF channel.

- ◆ au8BSSID

```
uint8 au8BSSID[6]
```

BSSID of the AP.

- ◆ au8SSID

```
uint8 au8SSID[M2M_MAX_SSID_LEN]
```

AP ssid.

- ◆ PAD8

```
uint8 _PAD8_
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mWifiStateChanged Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8CurrState`

`uint8 u8ErrCode`

`uint8 __PAD16__ [2]`

Detailed Description

Wi-Fi Connection State.

See also

[M2M_WIFI_DISCONNECTED](#), [M2M_WIFI_CONNECTED](#),
[M2M_WIFI_REQ_CON_STATE_CHANGED](#),[tenuM2mConnChangedErrcode](#)

Field Documentation

◆ `u8CurrState`

`uint8 u8CurrState`

Current Wi-Fi connection state

- ◆ **u8ErrCode**

```
uint8 u8ErrCode
```

Error type review tenuM2mConnChangedErrcode

- ◆ **__PAD16__**

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mPsType Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8PsType`

`uint8 u8BcastEn`

`uint8 __PAD16__ [2]`

Detailed Description

Power Save Configuration.

See also

[tenuPowerSaveModes](#)

Field Documentation

◆ `u8PsType`

`uint8 u8PsType`

Power save operating mode

- ◆ u8BcastEn

```
uint8 u8BcastEn
```

- ◆ __PAD16__

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mSlpReqTime Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint32 u32SleepTime`

Detailed Description

Manual power save request sleep time.

Field Documentation

◆ `u32SleepTime`

`uint32 u32SleepTime`

< Sleep time in ms



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mLsnInt Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16LsnInt`

`uint8 __PAD16 [2]`

Detailed Description

Listen interval.

It is the value of the Wi-Fi STA listen interval for power saving. It is given in units of Beacon period. Periodically after the listen interval fires, the WINC is wakeup and listen to the beacon and check for any buffered frames for it from the AP.

Field Documentation

◆ `u16LsnInt`

`uint16 u16LsnInt`

Listen interval in Beacon period count.

- ◆ PAD16

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment



tstrM2MWifiMonitorModeCtrl Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

Data Fields

uint8 u8ChannelID

uint8 u8FrameType

uint8 u8FrameSubtype

uint8 au8SrcMacAddress [6]

uint8 au8DstMacAddress [6]

uint8 au8BSSID [6]

uint8 u8EnRecvHdr

uint8 __PAD16__ [2]

Detailed Description

Wi-Fi Monitor Mode Filter.

This structure sets the filtering criteria for WLAN packets when monitoring mode is enable. The received packets matching the filtering parameters, are passed directly to the application.

Field Documentation

- ◆ u8ChannelID

```
uint8 u8ChannelID
```

- ◆ u8FrameType

```
uint8 u8FrameType
```

It must use values from tenuWifiFrameType.

- ◆ u8FrameSubtype

```
uint8 u8FrameSubtype
```

It must use values from tenuSubTypes.

- ◆ au8SrcMacAddress

```
uint8 au8SrcMacAddress[6]
```

- ◆ au8DstMacAddress

```
uint8 au8DstMacAddress[6]
```

- ◆ au8BSSID

```
uint8 au8BSSID[6]
```

- ◆ u8EnRecvHdr

```
uint8 u8EnRecvHdr
```

- ◆ __PAD16__

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MWifiRxPacketInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

uint8 u8FrameType

uint8 u8FrameSubtype

uint8 u8ServiceClass

uint8 u8Priority

uint8 u8HeaderLength

uint8 u8CipherType

uint8 au8SrcMacAddress [6]

uint8 au8DstMacAddress [6]

uint8 au8BSSID [6]

uint16 u16DataLength

uint16 u16FrameLength

uint32 u32DataRateKbps

sint8 s8RSSI

```
uint8 __PAD24__ [3]
```

Detailed Description

Wi-Fi RX Frame Header.

The M2M application has the ability to allow Wi-Fi monitoring mode for receiving all Wi-Fi Raw frames matching a well defined filtering criteria. When a target Wi-Fi packet is received, the header information are extracted and assigned in this structure.

Field Documentation

◆ u8FrameType

```
uint8 u8FrameType
```

It must use values from tenuWifiFrameType.

◆ u8FrameSubtype

```
uint8 u8FrameSubtype
```

It must use values from tenuSubTypes.

◆ u8ServiceClass

```
uint8 u8ServiceClass
```

Service class from Wi-Fi header.

◆ u8Priority

```
uint8 u8Priority
```

Priority from Wi-Fi header.

- ◆ **u8HeaderLength**

```
uint8 u8HeaderLength
```

Frame Header length.

- ◆ **u8CipherType**

```
uint8 u8CipherType
```

Encryption type for the rx packet.

- ◆ **au8SrcMacAddress**

```
uint8 au8SrcMacAddress[6]
```

- ◆ **au8DstMacAddress**

```
uint8 au8DstMacAddress[6]
```

- ◆ **au8BSSID**

```
uint8 au8BSSID[6]
```

- ◆ **u16DataLength**

```
uint16 u16DataLength
```

Data payload length (Header excluded).

- ◆ **u16FrameLength**

```
uint16 u16FrameLength
```

Total frame length (Header + Data).

- ◆ **u32DataRateKbps**

```
uint32 u32DataRateKbps
```

Data Rate in Kbps.

- ◆ **s8RSSI**

```
sint8 s8RSSI
```

RSSI.

- ◆ **__PAD24__**

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MWifiTxPacketInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16PacketSize`

`uint16 u16HeaderLength`

Detailed Description

Wi-Fi TX Packet Info.

The M2M Application has the ability to compose a RAW Wi-Fi frames (under the application responsibility). When transmitting a Wi-Fi packet, the application must supply the firmware with this structure for sending the target frame.

Field Documentation

◆ `u16PacketSize`

`uint16 u16PacketSize`

Wlan frame length.

◆ u16HeaderLength

```
uint16 u16HeaderLength
```

Wlan frame header length.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MP2PConnect Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8ListenChannel`

`uint8 __PAD24__ [3]`

Detailed Description

Set the device to operate in the Wi-Fi Direct (P2P) mode.

Field Documentation

♦ `u8ListenChannel`

`uint8 u8ListenChannel`

P2P Listen Channel (1, 6 or 11)

♦ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MAPConfig Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8SSID [M2M_MAX_SSID_LEN]`

`uint8 u8ListenChannel`

`uint8 u8KeyIndx`

`uint8 u8KeySz`

`uint8 au8WepKey [WEP_104_KEY_STRING_SIZE+1]`

`uint8 u8SecType`

`uint8 u8SsidHide`

`uint8 au8DHCPServerIP [4]`

`uint8 au8Key [M2M_MAX_PSK_LEN]`

`uint8 __PAD24__ [2]`

Detailed Description

AP Configuration.

This structure holds the configuration parameters for the M2M AP mode. It should be set by the application when it requests to enable the M2M AP operation mode. The M2M AP mode currently supports only WEP security (with the NO Security option available of course).

Field Documentation

- ◆ au8SSID

```
uint8 au8SSID[M2M_MAX_SSID_LEN]
```

< Configuration parameters for the WiFi AP.AP SSID

- ◆ u8ListenChannel

```
uint8 u8ListenChannel
```

Wi-Fi RF Channel which the AP will operate on

- ◆ u8KeyIdx

```
uint8 u8KeyIdx
```

Wep key Index

- ◆ u8KeySz

```
uint8 u8KeySz
```

Wep/WPA key Size

- ◆ au8WepKey

```
uint8 au8WepKey[WEP_104_KEY_STRING_SIZE+1]
```

Wep key

- ◆ u8SecType

```
uint8 u8SecType
```

Security type: Open or WEP or WPA in the current implementation

- ◆ u8SsidHide

```
uint8 u8SsidHide
```

SSID Status "Hidden(1)/Visible(0)"

- ◆ au8DHCPServerIP

```
uint8 au8DHCPServerIP[4]
```

Ap IP server address

- ◆ au8Key

```
uint8 au8Key[M2M_MAX_PSK_LEN]
```

WPA key

- ◆ __PAD24__

```
uint8 __PAD24__[2]
```

Padding bytes for forcing alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mServerInit Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8Channel`

`uint8 __PAD24__ [3]`

Detailed Description

PS Server initialization.

Field Documentation

◆ `u8Channel`

`uint8 u8Channel`

Server Listen channel

◆ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mClientState Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8State`

`uint8 __PAD24__ [3]`

Detailed Description

PS Client State.

Field Documentation

◆ `u8State`

`uint8 u8State`

PS Client State

◆ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



tstrM2Mservercmd Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

Data Fields

`uint8 u8cmd`

`uint8 __PAD24__ [3]`

Detailed Description

PS Server CMD.

Field Documentation

◆ `u8cmd`

`uint8 u8cmd`

PS Server Cmd

◆ `__PAD24__`

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mSetMacAddress Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8Mac [6]`

`uint8 __PAD16__ [2]`

Detailed Description

Sets the MAC address from application. The WINC load the mac address from the effuse by default to the WINC configuration memory, but that function is used to let the application overwrite the configuration memory with the mac address from the host.

Note

It's recommended to call this only once before calling connect request and after the `m2m_wifi_init`

Field Documentation

◆ au8Mac

`uint8 au8Mac[6]`

MAC address array

- ◆ __PAD16__

```
uint8 __PAD16__[2]
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MDeviceNameConfig Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8DeviceName [M2M_DEVICE_NAME_MAX]`

Detailed Description

Device name.

It is assigned by the application. It is used mainly for Wi-Fi Direct device discovery and WPS device information.

Field Documentation

◆ au8DeviceName

`uint8 au8DeviceName[M2M_DEVICE_NAME_MAX]`

NULL terminated device name



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MIPConfig Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint32 u32StaticIP`

`uint32 u32Gateway`

`uint32 u32DNS`

`uint32 u32SubnetMask`

`uint32 u32DhcpLeaseTime`

Detailed Description

Static IP configuration.

Note

All member IP addresses are expressed in Network Byte Order (eg. "192.168.10.1" will be expressed as 0x010AA8C0).

Field Documentation

- ◆ `u32StaticIP`

uint32 u32StaticIP

The static IP assigned to the device.

◆ **u32Gateway**

uint32 u32Gateway

IP of the Default internet gateway.

◆ **u32DNS**

uint32 u32DNS

IP for the DNS server.

◆ **u32SubnetMask**

uint32 u32SubnetMask

Subnet mask for the local area network.

◆ **u32DhcpLeaseTime**

uint32 u32DhcpLeaseTime

Dhcp Lease Time in sec



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2mIpRsvdPkt Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16PktSz`

`uint16 u16PktOffset`

Detailed Description

Received Packet Size and Data Offset.

Field Documentation

◆ `u16PktSz`

`uint16 u16PktSz`

◆ `u16PktOffset`

`uint16 u16PktOffset`



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MProvisionModeConfig Struct Reference

WLAN » DataTypes

Data Fields

tstrM2MAPConfig strApConfig

char acHttpServerDomainName [64]

uint8 u8EnableRedirect

uint8 __PAD24__ [3]

Detailed Description

M2M Provisioning Mode Configuration.

Field Documentation

- ◆ **strApConfig**

tstrM2MAPConfig strApConfig

Configuration parameters for the WiFi AP.

◆ acHttpServerDomainName

```
char acHttpServerDomainName[64]
```

The device domain name for HTTP provisioning.

◆ u8EnableRedirect

```
uint8 u8EnableRedirect
```

A flag to enable/disable HTTP redirect feature for the HTTP Provisioning server. If the Redirect is enabled, all HTTP traffic (<http://URL>) from the device associated with WINC AP will be redirected to the HTTP Provisioning Web page.

- 0 : Disable HTTP Redirect.
- 1 : Enable HTTP Redirect.

◆ __PAD24__

```
uint8 __PAD24__[3]
```



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MProvisionInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8SSID [M2M_MAX_SSID_LEN]`

`uint8 au8Password [M2M_MAX_PSK_LEN]`

`uint8 u8SecType`

`uint8 u8Status`

Detailed Description

M2M Provisioning Information obtained from the HTTP Provisioning server.

Field Documentation

◆ au8SSID

`uint8 au8SSID[M2M_MAX_SSID_LEN]`

Provisioned SSID.

◆ au8Password

```
uint8 au8Password[M2M_MAX_PSK_LEN]
```

Provisioned Password.

◆ u8SecType

```
uint8 u8SecType
```

Wifi Security type.

◆ u8Status

```
uint8 u8Status
```

Provisioning status. It must be checked before reading the provisioning information. It may be

- M2M_SUCCESS : Provision successful.
- M2M_FAIL : Provision Failed.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MConnInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

char acSSID [M2M_MAX_SSID_LEN]

uint8 u8SecType

uint8 au8IPAddr [4]

uint8 au8MACAddress [6]

sint8 s8RSSI

uint8 __PAD24__ [3]

Detailed Description

M2M Provisioning Information obtained from the HTTP Provisioning server.

Field Documentation

◆ acSSID

char acSSID[M2M_MAX_SSID_LEN]

AP connection SSID name

- ◆ u8SecType

```
uint8 u8SecType
```

Security type

- ◆ au8IPAddr

```
uint8 au8IPAddr[4]
```

Connection IP address

- ◆ au8MACAddress

```
uint8 au8MACAddress[6]
```

MAC address of the peer Wi-Fi station

- ◆ s8RSSI

```
sint8 s8RSSI
```

Connection RSSI signal

- ◆ __PAD24__

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrOtaInitHdr Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint32 u32OtaMagicValue`

`uint32 u32OtaPayloadSzie`

Detailed Description

OTA Image Header.

Field Documentation

◆ `u32OtaMagicValue`

`uint32 u32OtaMagicValue`

Magic value kept in the OTA image after the sha256 Digest buffer to define the Start of OTA Header

◆ `u32OtaPayloadSzie`

uint32 u32OtaPayloadSzie

The Total OTA image payload size, include the sha256 key size

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrOtaControlSec Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

uint32 u32OtaMagicValue

uint32 u32OtaFormatVersion

uint32 u32OtaSequenceNumber

uint32 u32OtaLastCheckTime

uint32 u32OtaCurrentworkingImagOffset

uint32 u32OtaCurrentworkingImagFirmwareVer

uint32 u32OtaRollbackImageOffset

uint32 u32OtaRollbackImageValidStatus

uint32 u32OtaRollbackImagFirmwareVer

uint32 u32OtaCortusAppWorkingOffset

uint32 u32OtaCortusAppWorkingValidSts

uint32 u32OtaCortusAppWorkingVer

uint32 u32OtaCortusAppRollbackOffset

```
uint32 u32OtaCortusAppRollbackValidSts
```

```
uint32 u32OtaCortusAppRollbackVer
```

```
uint32 u32OtaControlSecCrc
```

Detailed Description

Control section structure is used to define the working image and the validity of the roll-back image and its offset, also both firmware versions is kept in that structure.

Field Documentation

- ◆ **u32OtaMagicValue**

```
uint32 u32OtaMagicValue
```

Magic value used to ensure the structure is valid or not

- ◆ **u32OtaFormatVersion**

```
uint32 u32OtaFormatVersion
```

NA NA NA Flash version cs struct version 00 00 00 00 00 Control structure format version, the value will be incremented in case of structure changed or updated

- ◆ **u32OtaSequenceNumber**

```
uint32 u32OtaSequenceNumber
```

Sequence number is used while update the control structure to keep track of how many times that section updated

- ◆ **u32OtaLastCheckTime**

uint32 u32OtaLastCheckTime

Last time OTA check for update

◆ **u32OtaCurrentworkingImagOffset**

uint32 u32OtaCurrentworkingImagOffset

Current working offset in flash

◆ **u32OtaCurrentworkingImagFirmwareVer**

uint32 u32OtaCurrentworkingImagFirmwareVer

current working image version ex 18.0.1

◆ **u32OtaRollbackImageOffset**

uint32 u32OtaRollbackImageOffset

Roll-back image offset in flash

◆ **u32OtaRollbackImageValidStatus**

uint32 u32OtaRollbackImageValidStatus

roll-back image valid status

◆ **u32OtaRollbackImagFirmwareVer**

uint32 u32OtaRollbackImagFirmwareVer

Roll-back image version (ex 18.0.3)

- ◆ u32OtaCortusAppWorkingOffset

```
uint32 u32OtaCortusAppWorkingOffset
```

cortus app working offset in flash

- ◆ u32OtaCortusAppWorkingValidSts

```
uint32 u32OtaCortusAppWorkingValidSts
```

Working Cortus app valid status

- ◆ u32OtaCortusAppWorkingVer

```
uint32 u32OtaCortusAppWorkingVer
```

Working cortus app version (ex 18.0.3)

- ◆ u32OtaCortusAppRollbackOffset

```
uint32 u32OtaCortusAppRollbackOffset
```

cortus app rollback offset in flash

- ◆ u32OtaCortusAppRollbackValidSts

```
uint32 u32OtaCortusAppRollbackValidSts
```

roll-back cortus app valid status

- ◆ u32OtaCortusAppRollbackVer

```
uint32 u32OtaCortusAppRollbackVer
```

Roll-back cortus app version (ex 18.0.3)

- ◆ u32OtaControlSecCrc

```
uint32 u32OtaControlSecCrc
```

CRC for the control structure to ensure validity

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrOtaUpdateStatusResp Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8OtaUpdateStatusType`

`uint8 u8OtaUpdateStatus`

`uint8 _PAD16_ [2]`

Detailed Description

OTA Update Information.

See also

[tenuWPSTrigger](#)

Field Documentation

◆ `u8OtaUpdateStatusType`

`uint8 u8OtaUpdateStatusType`

Status type `tenuOtaUpdateStatusType`

- ◆ u8OtaUpdateStatus

```
uint8 u8OtaUpdateStatus
```

OTA_SUCCESS OTA_ERR_WORKING_IMAGE_LOAD_FAIL
OTA_ERR_INVALID_CONTROL_SEC M2M_ERR_OTA_SWITCH_FAIL
M2M_ERR_OTA_START_UPDATE_FAIL M2M_ERR_OTA_ROLLBACK_FAIL
M2M_ERR_OTA_INVALID_FLASH_SIZE M2M_ERR_OTA_INVALID_ARG

- ◆ _PAD16_

```
uint8 _PAD16_[2]
```



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrOtaUpdateInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint32 u8NcfUpgradeVersion`

`uint32 u8NcfCurrentVersion`

`uint32 u8NcdUpgradeVersion`

`uint8 u8NcdRequiredUpgrade`

`uint8 u8DownloadUrlOffset`

`uint8 u8DownloadUrlSize`

`uint8 __PAD8__`

Detailed Description

OTA Update Information.

See also

[tenuWPSTrigger](#)

Field Documentation

- ◆ u8NcfUpgradeVersion

```
uint32 u8NcfUpgradeVersion
```

NCF OTA Upgrade Version

- ◆ u8NcfCurrentVersion

```
uint32 u8NcfCurrentVersion
```

NCF OTA Current firmware version

- ◆ u8NcdUpgradeVersion

```
uint32 u8NcdUpgradeVersion
```

NCD (host) upgraded version (if the u8NcdRequiredUpgrade == true)

- ◆ u8NcdRequiredUpgrade

```
uint8 u8NcdRequiredUpgrade
```

NCD Required upgrade to the above version

- ◆ u8DownloadUrlOffset

```
uint8 u8DownloadUrlOffset
```

Download URL offset in the received packet

- ◆ u8DownloadUrlSize

```
uint8 u8DownloadUrlSize
```

Download URL size in the received packet

- ◆ PAD8

```
uint8 PAD8
```

Padding bytes for forcing 4-byte alignment

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrSystemTime Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint16 u16Year`

`uint8 u8Month`

`uint8 u8Day`

`uint8 u8Hour`

`uint8 u8Minute`

`uint8 u8Second`

`uint8 __PAD8__`

Detailed Description

Used for time storage.

Field Documentation

- ◆ `u16Year`

uint16 u16Year

◆ u8Month

uint8 u8Month

◆ u8Day

uint8 u8Day

◆ u8Hour

uint8 u8Hour

◆ u8Minute

uint8 u8Minute

◆ u8Second

uint8 u8Second

◆ __PAD8__

uint8 __PAD8__



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrM2MMulticastMac Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8macaddress [M2M_MAC_ADDRES_LEN]`

`uint8 u8AddRemove`

`uint8 __PAD8__`

Detailed Description

M2M add/remove multi-cast mac address.

Field Documentation

◆ `au8macaddress`

`uint8 au8macaddress[M2M_MAC_ADDRES_LEN]`

Mac address needed to be added or removed from filter.

- ◆ u8AddRemove

```
uint8 u8AddRemove
```

set by 1 to add or 0 to remove from filter.

- ◆ __PAD8__

```
uint8 __PAD8__
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrPrng Struct Reference

WLAN » DataTypes

Data Fields

`uint8 * pu8RngBuff`

`uint16 u16PrngSize`

`uint8 __PAD16__ [2]`

Detailed Description

M2M Request PRNG.

Field Documentation

◆ pu8RngBuff

`uint8* pu8RngBuff`

< return buffer address PRNG size requested

◆ u16PrngSize

```
uint16 u16PrngSize
```

PRNG pads

- ◆ PAD16

```
uint8 __PAD16__[2]
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrTlsCrlEntry Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8DataLen`

`uint8 au8Data [TLS_CRL_DATA_MAX_LEN]`

`uint8 __PAD24__ [3]`

Detailed Description

Certificate data for inclusion in a revocation list (CRL)

Field Documentation

◆ `u8DataLen`

`uint8 u8DataLen`

Length of certificate data (maximum possible is `TLS_CRL_DATA_MAX_LEN`)

- ◆ au8Data

```
uint8 au8Data[TLS_CRL_DATA_MAX_LEN]
```

Certificate data

- ◆ __PAD24__

```
uint8 __PAD24__[3]
```

Padding bytes for forcing 4-byte alignment



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrTlsCrlInfo Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 u8CrlType`

`uint8 u8Rsv1`

`uint8 u8Rsv2`

`uint8 u8Rsv3`

`tstrTlsCrlEntry astrTlsCrl [TLS_CRL_MAX_ENTRIES]`

Detailed Description

Certificate revocation list details.

Field Documentation

◆ `u8CrlType`

`uint8 u8CrlType`

Type of certificate data contained in list

- ◆ u8Rsv1

```
uint8 u8Rsv1
```

Reserved for future use

- ◆ u8Rsv2

```
uint8 u8Rsv2
```

Reserved for future use

- ◆ u8Rsv3

```
uint8 u8Rsv3
```

Reserved for future use

- ◆ astrTlsCrl

```
tstrTlsCrlEntry astrTlsCrl[TLS_CRL_MAX_ENTRIES]
```

List entries



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

Data Fields

tstrTlsSrvSecFileEntry Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

char acFileName [TLS_FILE_NAME_MAX]

uint32 u32FileSize

uint32 u32FileAddr

Detailed Description

This struct contains a TLS certificate.

Field Documentation

◆ acFileName

char acFileName[TLS_FILE_NAME_MAX]

Name of the certificate.

◆ u32FileSize

```
uint32 u32FileSize
```

Size of the certificate.

◆ u32FileAddr

```
uint32 u32FileAddr
```

Error Code.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrTlsSrvSecHdr Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint8 au8SecStartPattern [TLS_SRV_SEC_START_PATTERN_LEN]`

`uint32 u32nEntries`

`uint32 u32NextWriteAddr`

`tstrTlsSrvSecFileEntry astrEntries [TLS_SRV_SEC_MAX_FILES]`

Detailed Description

This struct contains a set of TLS certificates.

Field Documentation

◆ au8SecStartPattern

`uint8 au8SecStartPattern[TLS_SRV_SEC_START_PATTERN_LEN]`

Start pattern.

- ◆ u32nEntries

```
uint32 u32nEntries
```

Number of certificates stored in the struct.

- ◆ u32NextWriteAddr

```
uint32 u32NextWriteAddr
```

TLS Certificates.

- ◆ astrEntries

```
tstrTlsSrvSecFileEntry astrEntries[TLS_SRV_SEC_MAX_FILES]
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrSslSetActiveCsList Struct Reference

[WLAN](#) » [DataTypes](#)

Data Fields

`uint32 u32CsBMP`

Field Documentation

◆ `u32CsBMP`

`uint32 u32CsBMP`

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

tstrM2mPwrState Struct Reference

[WLAN](#) » [DataTypes](#)

Detailed Description

Power Mode.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by doxygen 1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

BSP

Modules

Defines

DataTypes

Function

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

doxygen 1.8.13



Defines

BSP

Macros

Macros

```
#define NMI_API  
  
#define CONST const  
  
#define NULL ((void*)0)  
  
#define BSP_MIN(x, y) ((x)>(y)?(y):(x))
```

Detailed Description

Macro Definition Documentation

◆ NMI_API

```
#define NMI_API
```

Attribute used to define memory section to map Functions in host memory.

◆ CONST

```
#define CONST const
```

Used for code portability.

◆ NULL

```
#define NULL ((void*)0)
```

Void Pointer to '0' in case of NULL is not defined.

◆ BSP_MIN

```
#define BSP_MIN ( x,  
                 y  
             )   ((x)>(y)?(y):(x))
```

Computes the minimum of **x** and **y**.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



DataTypes

BSP

Typedefs

```
typedef void(* tpfNmBspls) (void)
```

```
typedef unsigned char uint8
```

```
typedef unsigned short uint16
```

```
typedef unsigned long uint32
```

```
typedef signed char sint8
```

```
typedef signed short sint16
```

```
typedef signed long sint32
```

Detailed Description

Typedef Documentation

◆ tpfNmBspls

```
void(* tpfNmBspls) (void)
```

Pointer to function.

Used as a data type of ISR function registered by [nm_bsp_register_isr](#).

Returns

None

◆ uint8

```
unsigned char uint8
```

Range of values between 0 to 255.

◆ uint16

```
unsigned short uint16
```

Range of values between 0 to 65535.

◆ uint32

```
unsigned long uint32
```

Range of values between 0 to 4294967295.

◆ sint8

```
signed char sint8
```

Range of values between -128 to 127.

◆ sint16

```
signed short sint16
```

Range of values between -32768 to 32767.

◆ **sint32**

signed long **sint32**

Range of values between -2147483648 to 2147483647.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

Function

BSP

Modules

[nm_bsp_init](#)

[nm_bsp_deinit](#)

[nm_bsp_reset](#)

[nm_bsp_sleep](#)

[nm_bsp_register_isr](#)

[nm_bsp_interrupt_ctrl](#)

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



nm_bsp_init

BSP » Function

Functions

Functions

`sint8 nm_bsp_init (void)`

Detailed Description

Initialization for BSP (Board Support Package) such as Reset and Chip Enable Pins for WINC, delays, register ISR, enable/disable IRQ for WINC, ...etc. You must use this function in the head of your application to enable WINC and Host Driver to communicate with each other.

Function Documentation

◆ nm_bsp_init()

`sint8 nm_bsp_init (void)`

This function is used to initialize the **Board Support Package (BSP)** in order to prepare the WINC before it starts working.

The `nm_bsp_init` function is the first function that should be called at the beginning of every application to initialize the BSP and the WINC board. Otherwise, the rest of the BSP function calls will return with failure. This function should also be called after the WINC has been switched off with a successful call to "`nm_bsp_deinit`" in order to reinitialize the BSP before the

user can use any of the WINC API functions again. After the function initialize the WINC. Hard reset must be applied to start the WINC board.

Note

Implementation of this function is host dependent.

Warning

Inappropriate use of this function will lead to unavailability of host-chip communication.

See also

[nm_bsp_deinit](#), [nm_bsp_reset](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



nm_bsp_deinit

BSP » Function

Functions

Functions

`sint8 nm_bsp_deinit (void)`

Detailed Description

De-initialization for BSP ((Board Support Package)). This function should be called only after a successful call to `nm_bsp_init`.

Function Documentation

◆ `nm_bsp_deinit()`

`sint8 nm_bsp_deinit (void)`

This function is used to de-initialize the BSP and turn off the WINC board.

Precondition

The BSP should be initialized through `nm_bsp_init` first. The `nm_bsp_deinit` is the last function that should be called after the application has finished and before the WINC is switched off. The function call turns off the WINC board by setting CHIP_EN and RESET_N signals low. Every function call of "`nm_bsp_init`" should be matched with a call to `nm_bsp_deinit`. Failure to do so may result in the WINC consuming higher power

than expected.

Note

Implementation of this function is host dependent.

Warning

misuse may lead to unknown behavior in case of soft reset.

See also

[nm_bsp_init](#)

Returns

The function returns [M2M_SUCCESS](#) for successful operations and a negative value otherwise.



nm_bsp_reset

BSP » Function

Functions

Functions

void **nm_bsp_reset** (void)

Detailed Description

Resetting WINC1500 SoC by setting CHIP_EN and RESET_N signals low, then after specific delay the function will put CHIP_EN high then RESET_N high, for the timing between signals please review the WINC data-sheet

Function Documentation

◆ nm_bsp_reset()

```
void nm_bsp_reset ( void )
```

Applies a hardware reset to the WINC board. The "nm_bsp_reset" is used to apply a hard reset to the WINC board by setting CHIP_EN and RESET_N signals low, then after specific delay the function will put CHIP_EN high then RESET_N high, for the detailed timing between signals please review the WINC data-sheet. After a successful call, the WINC board firmware will kick off to load and kick off the WINC firmware. This function should be called to reset the WINC firmware after the BSP is initialized and before the start of any communication with WINC board. Calling this function at any other time will result in losing the state and

connections saved in the WINC board and starting again from the initial state. The host driver will need to be de-initialized before calling `nm_bsp_reset` and initialized again after it using the "`m2m_wifi_(de)init`".

Parameters

[in] **None**

Precondition

Initialize `nm_bsp_init` first

Note

Implementation of this function is host dependent and called by HIF layer.

Warning

Calling this function will drop any connection and internal state saved on the WINC firmware.

See also

`nm_bsp_init`, `m2m_wifi_init`, `m2m_wifi_deinit`

Returns

None

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



nm_bsp_sleep

BSP » Function

Functions

Functions

```
void nm_bsp_sleep (uint32 u32TimeMsec)
```

Detailed Description

Sleep in units of milliseconds.

This function used by HIF Layer according to different situations.

Function Documentation

• nm_bsp_sleep()

```
void nm_bsp_sleep ( uint32 u32TimeMsec )
```

Used to put the host to sleep for the specified duration. Forcing the host to sleep for extended period may lead to host not being able to respond to WINC board events. It's important to be considerate while choosing the sleep period.

Parameters

[in] **u32TimeMsec** Time unit in milliseconds

Precondition

Initialize [nm_bsp_init](#) first

Warning

Maximum value must not exceed 4294967295 milliseconds which is equal to 4294967.295 seconds.

Note

Implementation of this function is host dependent.

See also

[nm_bsp_init](#)

Returns

None



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Functions

nm_bsp_register_isr

BSP » Function

Functions

```
void nm_bsp_register_isr (tpfNmBsplsr pfslr)
```

Detailed Description

Register ISR (Interrupt Service Routine) in the initialization of HIF (Host Interface) Layer. When the interrupt trigger the BSP layer should call the pfslr function once inside the interrupt.

Function Documentation

◆ nm_bsp_register_isr()

```
void nm_bsp_register_isr ( tpfNmBsplsr pfslr )
```

Register the host interface interrupt service routine. WINC board utilize SPI interface to communicate with the host. This function register the SPI interrupt to notify the host whenever there is an outstanding message from the WINC board. The function should be called during the initialization of the host interface. It is an internal driver function and shouldn't be called by the application.

Parameters

[in] **tpfNmBsplsr** pfslr Pointer to ISR handler in HIF

Warning

Make sure that ISR for IRQ pin for WINC is disabled by default in your implementation.

Note

Implementation of this function is host dependent and called by HIF layer.

See also

[tpfNmBspls](#)

Returns

None



nm_bsp_interrupt_ctrl

BSP » Function

Functions

Functions

```
void nm_bsp_interrupt_ctrl (uint8 u8Enable)
```

Detailed Description

Synchronous enable/disable interrupts function

Function Documentation

◆ nm_bsp_interrupt_ctrl()

```
void nm_bsp_interrupt_ctrl ( uint8 u8Enable )
```

Enable/Disable interrupts This function can be used to enable/disable the WINC to host interrupt as the depending on how the driver is implemented. It an internal driver function and shouldn't be called by the application.

Precondition

The interrupt must be registered using nm_bsp_register_isr first.

Parameters

[in] **u8Enable** '0' disable interrupts. '1' enable interrupts

See also

[tpfNmBsplsr](#), [nm_bsp_register_isr](#)

Note

Implementation of this function is host dependent and called by HIF layer.

Returns

None

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



Socket

Modules

Modules

Defines

Error Codes

DataTypes

Function

Detailed Description

BSD compatible socket interface between the host layer and the network protocol stacks in the firmware. These functions are used by the host application to send or receive packets and to do other socket operations.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by 1.8.13



WINC1500 IoT Software APIs 19.5.2

WINC Software API Reference
Manual

Modules

Defines

Socket

Modules

TCP/IP Defines

TLS Defines

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



TCP/IP Defines

Socket » Defines

Macros

```
#define HOSTNAME_MAX_SIZE 64  
  
#define SOCKET_BUFFER_MAX_LENGTH 1400  
  
#define AF_INET 2  
  
#define SOCK_STREAM 1  
  
#define SOCK_DGRAM 2  
  
#define SOCKET_FLAGS_SSL 0x01  
  
#define TCP_SOCK_MAX (7)  
  
#define UDP_SOCK_MAX 4  
  
#define MAX_SOCKET (TCP_SOCK_MAX + UDP_SOCK_MAX)  
  
#define SOL_SOCKET 1  
  
#define SOL_SSL_SOCKET 2  
  
#define SO_SET_UDP_SEND_CALLBACK 0x00  
  
#define IP_ADD_MEMBERSHIP 0x01  
  
#define IP_DROP_MEMBERSHIP 0x02
```

Detailed Description

The following list of macros are used to define constants used throughout the socket layer.

Macro Definition Documentation

◆ HOSTNAME_MAX_SIZE

```
#define HOSTNAME_MAX_SIZE 64
```

Maximum allowed size for a host domain name passed to the function `gethostbyname` `gethostbyname`. command value. Used with the `setsockopt` function.

◆ SOCKET_BUFFER_MAX_LENGTH

```
#define SOCKET_BUFFER_MAX_LENGTH 1400
```

Maximum allowed size for a socket data buffer. Used with `send` socket function to ensure that the buffer sent is within the allowed range.

◆ AF_INET

```
#define AF_INET 2
```

The `AF_INET` is the address family used for IPv4. An IPv4 transport address is specified with the `sockaddr_in` structure. (It is the only supported type for the current implementation.)

◆ SOCK_STREAM

```
#define SOCK_STREAM 1
```

One of the IPv4 supported socket types for reliable connection-oriented stream connection. Passed to the `socket` function for the socket creation operation.

◆ SOCK_DGRAM

```
#define SOCK_DGRAM 2
```

One of the IPv4 supported socket types for unreliable connectionless datagram connection.
Passed to the [socket](#) function for the socket creation operation.

◆ SOCKET_FLAGS_SSL

```
#define SOCKET_FLAGS_SSL 0x01
```

This flag shall be passed to the socket API for SSL session.

◆ TCP_SOCK_MAX

```
#define TCP_SOCK_MAX (7)
```

Maximum number of simultaneous TCP sockets.

◆ UDP_SOCK_MAX

```
#define UDP_SOCK_MAX 4
```

Maximum number of simultaneous UDP sockets.

◆ MAX_SOCKET

```
#define MAX_SOCKET (TCP_SOCK_MAX + UDP_SOCK_MAX)
```

Maximum number of Sockets.

◆ SOL_SOCKET

```
#define SOL_SOCKET 1
```

Socket option. Used with the [setsockopt](#) function

- ◆ **SOL_SSL_SOCKET**

```
#define SOL_SSL_SOCKET 2
```

SSL Socket option level. Used with the [setsockopt](#) function

- ◆ **SO_SET_UDP_SEND_CALLBACK**

```
#define SO_SET_UDP_SEND_CALLBACK 0x00
```

Socket option used by the application to enable/disable the use of UDP send callbacks. Used with the [setsockopt](#) function.

- ◆ **IP_ADD_MEMBERSHIP**

```
#define IP_ADD_MEMBERSHIP 0x01
```

Set Socket Option Add Membership command value (to join a multicast group). Used with the [setsockopt](#) function.

- ◆ **IP_DROP_MEMBERSHIP**

```
#define IP_DROP_MEMBERSHIP 0x02
```

Set Socket Option Drop Membership command value (to leave a multicast group). Used with the [setsockopt](#) function.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

TLS Defines

[Socket](#) » [Defines](#)

Modules

[TLS Socket Options](#)

[Legacy names for TLS Cipher Suite IDs](#)

[TLS Cipher Suite IDs](#)

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



TLS Socket Options

[Socket](#) » [Defines](#) » [TLS Defines](#)

Macros

```
#define SO_SSL_BYPASS_X509_VERIF 0x01  
  
#define SO_SSL_SNI 0x02  
  
#define SO_SSL_ENABLE_SESSION_CACHING 0x03  
  
#define SO_SSL_ENABLE_SNI_VALIDATION 0x04
```

Detailed Description

The following list of macros are used to define SSL Socket options.

See also

[setsockopt](#)

Macro Definition Documentation

◆ SO_SSL_BYPASS_X509_VERIF

```
#define SO_SSL_BYPASS_X509_VERIF 0x01
```

Allow an opened SSL socket to bypass the X509 certificate verification process. It is highly

required NOT to use this socket option in production software applications. It is supported for debugging and testing purposes. The option value should be casted to int type and it is handled as a boolean flag.

◆ SO_SSL_SNI

```
#define SO_SSL_SNI 0x02
```

Set the Server Name Indicator (SNI) for an SSL socket. The SNI is a NULL terminated string containing the server name associated with the connection. It must not exceed the size of HOSTNAME_MAX_SIZE.

◆ SO_SSL_ENABLE_SESSION_CACHING

```
#define SO_SSL_ENABLE_SESSION_CACHING 0x03
```

This option allow the TLS to cache the session information for fast TLS session establishment in future connections using the TLS Protocol session resume features.

◆ SO_SSL_ENABLE_SNI_VALIDATION

```
#define SO_SSL_ENABLE_SNI_VALIDATION 0x04
```

Enable SNI validation against the server's certificate subject common name. If there is no SNI provided (via the SO_SSL_SNI option), setting this option does nothing.



Legacy names for TLS Cipher Suite IDs

[Socket](#) » [Defines](#) » [TLS Defines](#)

Macros

```
#define SSL_ENABLE_RSA_SHA_SUITES 0x01  
  
#define SSL_ENABLE_RSA_SHA256_SUITES 0x02  
  
#define SSL_ENABLE_DHE_SHA_SUITES 0x04  
  
#define SSL_ENABLE_DHE_SHA256_SUITES 0x08  
  
#define SSL_ENABLE_RSA_GCM_SUITES 0x10  
  
#define SSL_ENABLE_DHE_GCM_SUITES 0x20  
  
#define SSL_ENABLE_ALL_SUITES 0x0000003F
```

Detailed Description

The following list of macros MUST NOT be used. Instead use the new names under `SSLCipherSuiteID`

See also

[sslSetActiveCipherSuites](#)

Macro Definition Documentation

◆ SSL_ENABLE_RSA_SHA_SUITES

```
#define SSL_ENABLE_RSA_SHA_SUITES 0x01
```

Enable RSA Hmac_SHA based Cipher suites. For example,
TLS_RSA_WITH_AES_128_CBC_SHA

◆ SSL_ENABLE_RSA_SHA256_SUITES

```
#define SSL_ENABLE_RSA_SHA256_SUITES 0x02
```

Enable RSA Hmac_SHA256 based Cipher suites. For example,
TLS_RSA_WITH_AES_128_CBC_SHA256

◆ SSL_ENABLE_DHE_SHA_SUITES

```
#define SSL_ENABLE_DHE_SHA_SUITES 0x04
```

Enable DHE Hmac_SHA based Cipher suites. For example,
TLS_DHE_RSA_WITH_AES_128_CBC_SHA

◆ SSL_ENABLE_DHE_SHA256_SUITES

```
#define SSL_ENABLE_DHE_SHA256_SUITES 0x08
```

Enable DHE Hmac_SHA256 based Cipher suites. For example,
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256

◆ SSL_ENABLE_RSA_GCM_SUITES

```
#define SSL_ENABLE_RSA_GCM_SUITES 0x10
```

Enable RSA AEAD based Cipher suites. For example,

TLS_RSA_WITH_AES_128_GCM_SHA256

- ◆ **SSL_ENABLE_DHE_GCM_SUITES**

```
#define SSL_ENABLE_DHE_GCM_SUITES 0x20
```

Enable DHE AEAD based Cipher suites. For example,
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256

- ◆ **SSL_ENABLE_ALL_SUITES**

```
#define SSL_ENABLE_ALL_SUITES 0x0000003F
```

Enable all possible supported cipher suites.



TLS Cipher Suite IDs

Socket » Defines » TLS Defines

Macros

```
#define SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA NBIT0  
  
#define SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA256 NBIT1  
  
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA NBIT2  
  
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA256 NBIT3  
  
#define SSL_CIPHER_RSA_WITH_AES_128_GCM_SHA256 NBIT4  
  
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_GCM_SHA256 NBIT5  
  
#define SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA NBIT6  
  
#define SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA256 NBIT7  
  
#define SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA NBIT8  
  
#define SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA256 NBIT9  
  
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA NBIT10  
  
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_256_CBC_SHA NBIT11  
  
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA256 NBIT12  
  
#define SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 NBIT13
```

```
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_GCM_SHA256 NBIT14

#define SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 NBIT15

#define SSL_ECC_ONLY_CIPHERS

#define SSL_ECC_ALL_CIPHERS

#define SSL_NON_ECC_CIPHERS_AES_128

#define SSL_ECC_CIPHERS_AES_256

#define SSL_NON_ECC_CIPHERS_AES_256

#define SSL_CIPHER_ALL
```

Detailed Description

The following list of macros defined the list of supported TLS Cipher suites. Each MACRO defines a single Cipher suite.

See also

[m2m_ssl_set_active_ciphersuites](#)

Macro Definition Documentation

- ◆ **SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA**

```
#define SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA NBIT0
```

- ◆ **SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA256**

```
#define SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA256 NBIT1
```

- ◆ **SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA**

```
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA NBIT2
```

- **SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA256**

```
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA256 NBIT3
```

- **SSL_CIPHER_RSA_WITH_AES_128_GCM_SHA256**

```
#define SSL_CIPHER_RSA_WITH_AES_128_GCM_SHA256 NBIT4
```

- **SSL_CIPHER_DHE_RSA_WITH_AES_128_GCM_SHA256**

```
#define SSL_CIPHER_DHE_RSA_WITH_AES_128_GCM_SHA256 NBIT5
```

- **SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA**

```
#define SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA NBIT6
```

- **SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA256**

```
#define SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA256 NBIT7
```

- **SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA**

```
#define SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA NBIT8
```

- **SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA256**

```
#define SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA256 NBIT9
```

- **SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA**

```
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA NBIT10
```

- ◆ **SSL_CIPHER_ECDHE_RSA_WITH_AES_256_CBC_SHA**

```
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_256_CBC_SHA NBIT11
```

- ◆ **SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA256**

```
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA256 NBIT12
```

- ◆ **SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256**

```
#define SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 NBIT13
```

- ◆ **SSL_CIPHER_ECDHE_RSA_WITH_AES_128_GCM_SHA256**

```
#define SSL_CIPHER_ECDHE_RSA_WITH_AES_128_GCM_SHA256 NBIT14
```

- ◆ **SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256**

```
#define SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 NBIT15
```

- ◆ **SSL_ECC_ONLY_CIPHERS**

```
#define SSL_ECC_ONLY_CIPHERS
```

Value:

```
(\           SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 | \  
     SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256  
)
```

All ciphers that use ECC crypto only. This excludes ciphers that use RSA. They use ECDSA instead. These ciphers are turned off by default at startup. The application may enable them if it has an ECC math engine (like ATECC508).

◆ SSL_ECC_ALL_CIPHERS

```
#define SSL_ECC_ALL_CIPHERS
```

Value:

```
( \
    SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA      | \
    SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA256   | \
    SSL_CIPHER_ECDHE_RSA_WITH_AES_128_GCM_SHA256   | \
    SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 | \
    SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 ) \
```

All supported ECC Ciphers including those ciphers that depend on RSA and ECC. These ciphers are turned off by default at startup. The application may enable them if it has an ECC math engine (like ATECC508).

◆ SSL_NON_ECC_CIPHERS_AES_128

```
#define SSL_NON_ECC_CIPHERS_AES_128
```

Value:

```
( \
    SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA      | \
    SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA256   | \
    SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA   | \
    SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA256 | \
    SSL_CIPHER_RSA_WITH_AES_128_GCM_SHA256   | \
    SSL_CIPHER_DHE_RSA_WITH_AES_128_GCM_SHA256 ) \
```

All supported AES-128 Ciphers (ECC ciphers are not counted). This is the default active group after startup.

◆ SSL_ECC_CIPHERS_AES_256

```
#define SSL_ECC_CIPHERS_AES_256
```

Value:

```
( \
    SSL_CIPHER_ECDHE_RSA_WITH_AES_256_CBC_SHA     \ )
```

ECC AES-256 supported ciphers.

◆ SSL_NON_ECC_CIPHERS_AES_256

```
#define SSL_NON_ECC_CIPHERS_AES_256
```

Value:

```
(\SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA | \
SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA256 | \
SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA | \
SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA256 \
)
```

AES-256 Ciphers. This group is disabled by default at startup because the WINC1500 HW Accelerator supports only AES-128. If the application needs to force AES-256 cipher support, it could enable them (or any of them) explicitly by calling `sslSetActiveCipherSuites`.

◆ **SSL_CIPHER_ALL**

```
#define SSL_CIPHER_ALL
```

Value:

```
(\SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA \
SSL_CIPHER_RSA_WITH_AES_128_CBC_SHA256 \
SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA \
SSL_CIPHER_DHE_RSA_WITH_AES_128_CBC_SHA256 \
SSL_CIPHER_RSA_WITH_AES_128_GCM_SHA256 \
SSL_CIPHER_DHE_RSA_WITH_AES_128_GCM_SHA256 \
SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA \
SSL_CIPHER_RSA_WITH_AES_256_CBC_SHA256 \
SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA \
SSL_CIPHER_DHE_RSA_WITH_AES_256_CBC_SHA256 \
SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA \
SSL_CIPHER_ECDHE_RSA_WITH_AES_128_CBC_SHA256 \
SSL_CIPHER_ECDHE_RSA_WITH_AES_128_GCM_SHA256 \
SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 \
SSL_CIPHER_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 \
SSL_CIPHER_ECDHE_RSA_WITH_AES_256_CBC_SHA \
)
```

Turn On All TLS Ciphers.



Error Codes

Socket

Macros

Macros

```
#define SOCK_ERR_NO_ERROR 0

#define SOCK_ERR_INVALID_ADDRESS -1

#define SOCK_ERR_ADDR_ALREADY_IN_USE -2

#define SOCK_ERR_MAX_TCP_SOCK -3

#define SOCK_ERR_MAX_UDP_SOCK -4

#define SOCK_ERR_INVALID_ARG -6

#define SOCK_ERR_MAX_LISTEN_SOCK -7

#define SOCK_ERR_INVALID -9

#define SOCK_ERR_ADDR_IS_REQUIRED -11

#define SOCK_ERR_CONN_ABORTED -12

#define SOCK_ERR_TIMEOUT -13

#define SOCK_ERR_BUFFER_FULL -14

#define _htonl(m) ((uint32)((uint32)(m << 24)) | ((uint32)((m & 0x0000FF00) << 8)) |
((uint32)((m & 0x00FF0000) >> 8)) | ((uint32)(m >> 24)))

#define _htons(A) ((uint16)((uint16)(A) << 8) | (((uint16)(A)) >> 8))
```

```
#define _ntohl _htonl
```

```
#define _ ntohs _ htons
```

Detailed Description

The following list of macros are used to define the possible error codes returned as a result of a call to a socket function. Errors are listed in numerical order with the error macro name.

Macro Definition Documentation

◆ SOCK_ERR_NO_ERROR

```
#define SOCK_ERR_NO_ERROR 0
```

Successful socket operation

◆ SOCK_ERR_INVALID_ADDRESS

```
#define SOCK_ERR_INVALID_ADDRESS -1
```

Socket address is invalid. The socket operation cannot be completed successfully without specifying a specific address For example: bind is called without specifying a port number

◆ SOCK_ERR_ADDR_ALREADY_IN_USE

```
#define SOCK_ERR_ADDR_ALREADY_IN_USE -2
```

Socket operation cannot bind on the given address. With socket operations, only one IP address per socket is permitted. Any attempt for a new socket to bind with an IP address already bound to another open socket, will return the following error code. States that bind operation failed.

- ◆ SOCK_ERR_MAX_TCP_SOCK

```
#define SOCK_ERR_MAX_TCP_SOCK -3
```

Exceeded the maximum number of TCP sockets. A maximum number of TCP sockets opened simultaneously is defined through TCP_SOCK_MAX. It is not permitted to exceed that number at socket creation. Identifies that [socket](#) operation failed.

- ◆ SOCK_ERR_MAX_UDP_SOCK

```
#define SOCK_ERR_MAX_UDP_SOCK -4
```

Exceeded the maximum number of UDP sockets. A maximum number of UDP sockets opened simultaneously is defined through UDP_SOCK_MAX. It is not permitted to exceed that number at socket creation. Identifies that [socket](#) operation failed

- ◆ SOCK_ERR_INVALID_ARG

```
#define SOCK_ERR_INVALID_ARG -6
```

An invalid argument is passed to a function.

- ◆ SOCK_ERR_MAX_LISTEN_SOCK

```
#define SOCK_ERR_MAX_LISTEN_SOCK -7
```

Exceeded the maximum number of TCP passive listening sockets. Identifies that [listen](#) operation failed.

- ◆ SOCK_ERR_INVALID

```
#define SOCK_ERR_INVALID -9
```

The requested socket operation is not valid in the current socket state. For example: [accept](#) is called on a TCP socket before [bind](#) or [listen](#).

◆ SOCK_ERR_ADDR_IS_REQUIRED

```
#define SOCK_ERR_ADDR_IS_REQUIRED -11
```

Destination address is required. Failure to provide the socket address required for the socket operation to be completed. It is generated as an error to the [sendto](#) function when the address required to send the data to is not known.

◆ SOCK_ERR_CONN_ABORTED

```
#define SOCK_ERR_CONN_ABORTED -12
```

The socket is closed by the peer. The local socket is closed also.

◆ SOCK_ERR_TIMEOUT

```
#define SOCK_ERR_TIMEOUT -13
```

The socket pending operation has Timedout.

◆ SOCK_ERR_BUFFER_FULL

```
#define SOCK_ERR_BUFFER_FULL -14
```

No buffer space available to be used for the requested socket operation.

◆ _htonl

```
#define _htonl ((uint32)((uint32)(m << 24) | ((uint32)((m & 0x0000FF00) << 8)) | ((uint32)(m & 0x00FF0000) >> 8)) | ((uint32)(m >> 24)))
```

Convert a 4-byte integer from the host representation to the Network byte order representation.

◆ _htons

```
#define _htons( A ) (uint16)((((uint16)(A)) << 8) | (((uint16)(A)) >> 8))
```

Convert a 2-byte integer (short) from the host representation to the Network byte order representation.

◆ _ntohl

```
#define _ntohl _htonl
```

Convert a 4-byte integer from the Network byte order representation to the host representation

◆ _ ntohs

```
#define _ ntohs _ htons
```

Convert a 2-byte integer from the Network byte order representation to the host representation



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

[Modules](#) | [Data Structures](#) | [Typedefs](#)

DataTypes

[Socket](#)

Modules

[Asynchronous Events](#)

Data Structures

`struct in_addr`

`struct sockaddr`

`struct sockaddr_in`

Typedefs

`typedef sint8 SOCKET`

Detailed Description

Specific Enumeration-typedefs used for socket operations

Typedef Documentation

◆ SOCKET

SOCKET

Definition for socket handler data type. Socket ID, used with all socket operations to uniquely identify the socket handler. Such an ID is uniquely assigned at socket creation when calling **socket** operation.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



Asynchronous Events

Socket » DataTypes

[Data Structures](#) | [Typedefs](#) | [Enumerations](#)

Data Structures

struct **tstrSocketBindMsg**

struct **tstrSocketListenMsg**

struct **tstrSocketAcceptMsg**

struct **tstrSocketConnectMsg**

struct **tstrSocketRecvMsg**

Typedefs

typedef void(*) **tpfAppSocketCb**) (**SOCKET** sock, **uint8** u8Msg, void *pvMsg)

typedef void(*) **tpfAppResolveCb**) (**uint8** *pu8DomainName, **uint32** u32ServerIP)

typedef void(*) **tpfPingCb**) (**uint32** u32IPAddr, **uint32** u32RTT, **uint8** u8ErrorCode)

Enumerations

enum **tenuSocketCallbackMsgType** {
 SOCKET_MSG_BIND = 1, **SOCKET_MSG_LISTEN**,
 SOCKET_MSG_DNS_RESOLVE, **SOCKET_MSG_ACCEPT**,
 SOCKET_MSG_CONNECT, **SOCKET_MSG_RECV**, **SOCKET_MSG_SEND**,
 SOCKET_MSG_SENDTO,
 SOCKET_MSG_RECVFROM

```
}
```

Detailed Description

Specific Enumeration used for asynchronous operations

Typedef Documentation

◆ tpfAppSocketCb

```
tpfAppSocketCb
```

The main socket application callback function. Applications register their main socket application callback through this function by calling [registerSocketCallback](#). In response to events received, the following callback function is called to handle the corresponding asynchronous function called. Example: [bind](#), [connect](#),...etc.

Parameters

[in] **sock** Socket ID for the callback.

The socket callback function is called whenever a new event is received in response to socket operations.

Parameters

[in] **u8Msg** Socket event type. Possible values are:

- [SOCKET_MSG_BIND](#)
- [SOCKET_MSG_LISTEN](#)
- [SOCKET_MSG_ACCEPT](#)
- [SOCKET_MSG_CONNECT](#)
- [SOCKET_MSG_RECV](#)
- [SOCKET_MSG_SEND](#)
- [SOCKET_MSG_SENDTO](#)
- [SOCKET_MSG_RECVFROM](#)

[in] **pvMsg** Pointer to message structure. Existing types are:

- [tstrSocketBindMsg](#)
- [tstrSocketListenMsg](#)
- [tstrSocketAcceptMsg](#)
- [tstrSocketConnectMsg](#)
- [tstrSocketRecvMsg](#)

See also

[tenuSocketCallbackMsgType](#) [tstrSocketRecvMsg](#) [tstrSocketConnectMsg](#)
[tstrSocketAcceptMsg](#) [tstrSocketListenMsg](#) [tstrSocketBindMsg](#)

◆ tpfAppResolveCb

tpfAppResolveCb

DNS resolution callback function.

Applications requiring DNS resolution should register their callback through this function by calling [registerSocketCallback](#). The following callback is triggered in response to asynchronous call to the [gethostbyname](#) function (DNS Resolution callback).

Parameters

[in] **pu8DomainName** Domain name of the host.

[in] **u32ServerIP** Server IPv4 address encoded in NW byte order format. If it is Zero, then the DNS resolution failed.

◆ tpfPingCb

tpfPingCb

PING Callback.

The function delivers the ping statistics for the sent ping triggered by calling `m2m_ping_req`.

Parameters

[in] **u32IPAddr** Destination IP.

[in] **u32RTT** Round Trip Time.

[in] **u8ErrorCode** Ping error code. It may be one of:

- PING_ERR_SUCCESS
- PING_ERR_DEST_UNREACH
- PING_ERR_TIMEOUT

Enumeration Type Documentation

◆ tenuSocketCallbackMsgType

enum tenuSocketCallbackMsgType

Asynchronous APIs, make use of callback functions, in-order to return back the results once the corresponding socket operation is completed. Hence resuming the normal execution of the application code while the socket operation returns the results. Callback functions expect event messages to be passed in, in-order to identify the operation they're returning the results for. The following enum identifies the type of events that are received in the callback function.

Application Use: In order for application developers to handle the pending events from the network controller through the callback functions. A function call must be made to the function [m2m_wifi_handle_events](#) at least once for each socket operation.

See also

[bind](#) [listen](#) [accept](#) [connect](#) [send](#) [recv](#)

Enumerator	
SOCKET_MSG_BIND	Bind socket event.
SOCKET_MSG_LISTEN	Listen socket event.
SOCKET_MSG_DNS_RESOLVE	DNS Resolution event.
SOCKET_MSG_ACCEPT	Accept socket event.
SOCKET_MSG_CONNECT	Connect socket event.
SOCKET_MSG_RECV	Receive socket event.
SOCKET_MSG_SEND	Send socket event.
SOCKET_MSG_SENTO	sendto socket event.
SOCKET_MSG_RECVFROM	Recvfrom socket event.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrSocketBindMsg Struct Reference

[Socket](#) » [DataTypes](#) » [Asynchronous Events](#)

Data Fields

`sint8 status`

Detailed Description

Socket bind status.

An asynchronous call to the `bind` socket operation, returns information through this structure in response. This structure together with the event `SOCKET_MSG_BIND` are passed in paramters to the callback function.

See also

`bind`

Field Documentation

◆ `status`

`sint8 status`

The result of the bind operation. Holding a value of ZERO for a successful bind or otherwise a

negative error code corresponding to the type of error.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

tstrSocketListenMsg Struct Reference

[Socket](#) » [DataTypes](#) » [Asynchronous Events](#)

Data Fields

`sint8 status`

Detailed Description

Socket listen status.

Socket listen information is returned through this structure in response to the asynchronous call to the [listen](#) function. This structure together with the event [SOCKET_MSG_LISTEN](#) are passed-in parameters to the callback function.

See also

[listen](#)

Field Documentation

◆ `status`

`sint8 status`

Holding a value of ZERO for a successful listen or otherwise a negative error code

corresponding to the type of error.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



tstrSocketAcceptMsg Struct Reference

[Socket](#) » [DataTypes](#) » [Asynchronous Events](#)

Data Fields

SOCKET sock

struct **sockaddr_in** strAddr

Detailed Description

Socket accept status.

Socket accept information is returned through this structure in response to the asynchronous call to the [accept](#) function. This structure together with the event **SOCKET_MSG_ACCEPT** are passed-in parameters to the callback function.

Field Documentation

◆ SOCK

SOCKET sock

On a successful [accept](#) operation, the return information is the socket ID for the accepted connection with the remote peer. Otherwise a negative error code is returned to indicate

failure of the accept operation.

◆ strAddr

```
struct sockaddr_in strAddr
```

Socket address structure for the remote peer.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



tstrSocketConnectMsg Struct Reference

[Socket](#) » [DataTypes](#) » [Asynchronous Events](#)

Data Fields

Data Fields

SOCKET sock

sint8 s8Error

Detailed Description

Socket connect status.

Socket connect information is returned through this structure in response to the asynchronous call to the **connect** socket function. This structure together with the event **SOCKET_MSG_CONNECT** are passed-in parameters to the callback function.

Field Documentation

◆ SOCK

SOCKET sock

Socket ID referring to the socket passed to the connect function call.

◆ s8Error

sint8 s8Error

Connect error code. Holding a value of ZERO for a successful connect or otherwise a negative error code corresponding to the type of error.



tstrSocketRecvMsg Struct Reference

[Socket](#) » [DataTypes](#) » [Asynchronous Events](#)

Data Fields

Data Fields

`uint8 * pu8Buffer`

`sint16 s16BufferSize`

`uint16 u16RemainingSize`

`struct sockaddr_in strRemoteAddr`

Detailed Description

Socket recv status.

Socket receive information is returned through this structure in response to the asynchronous call to the `recv` or `recvfrom` socket functions. This structure together with the events `SOCKET_MSG_RECV` or `SOCKET_MSG_RECVFROM` are passed-in parameters to the callback function.

Remarks

In case the received data from the remote peer is larger than the USER buffer size defined during the asynchronous call to the `recv` function, the data is delivered to the user in a number of consecutive chunks according to the USER Buffer size. a negative or zero buffer size indicates an error with the following code: `SOCK_ERR_NO_ERROR` : Socket connection closed `SOCK_ERR_CONN_ABORTED` : Socket connection aborted : Socket

receive timed out

Field Documentation

- ◆ pu8Buffer

```
uint8* pu8Buffer
```

Pointer to the USER buffer (passed to [recv](#) and [recvfrom](#) function) containing the received data chunk.

- ◆ s16BufferSize

```
sint16 s16BufferSize
```

The received data chunk size. Holds a negative value if there is a receive error or ZERO on success upon reception of close socket message.

- ◆ u16RemainingSize

```
uint16 u16RemainingSize
```

The number of bytes remaining in the current [recv](#) operation.

- ◆ strRemoteAddr

```
struct sockaddr_in strRemoteAddr
```

Socket address structure for the remote peer. It is valid for [SOCKET_MSG_RECVFROM](#) event.



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

in_addr Struct Reference

[Socket](#) » [DataTypes](#)

Data Fields

`uint32 s_addr`

Detailed Description

IPv4 address representation.

This structure is used as a placeholder for IPV4 address in other structures.

See also

[sockaddr_in](#)

Field Documentation

◆ `s_addr`

`uint32 s_addr`

Network Byte Order representation of the IPv4 address. For example, the address "192.168.0.10" is represented as 0x0A00A8C0.



sockaddr Struct Reference

[Socket](#) » [DataTypes](#)

Data Fields

Data Fields

`uint16 sa_family`

`uint8 sa_data [14]`

Detailed Description

Generic socket address structure.

See also

[sockaddr_in](#)

Field Documentation

◆ `sa_family`

`uint16 sa_family`

Socket address family.

◆ `sa_data`

```
uint8 sa_data[14]
```

Maximum size of all the different socket address structures.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Fields

sockaddr_in Struct Reference

[Socket](#) » [DataTypes](#)

Data Fields

`uint16 sin_family`

`uint16 sin_port`

`in_addr sin_addr`

`uint8 sin_zero [8]`

Detailed Description

Socket address structure for IPV4 addresses. Used to specify socket address information to which to connect to. Can be cast to [sockaddr](#) structure.

Field Documentation

◆ `sin_family`

`uint16 sin_family`

Specifies the address family(AF). Members of AF_INET address family are IPv4 addresses.

Hence, the only supported value for this is AF_INET.

◆ sin_port

uint16 sin_port

Port number of the socket. Network sockets are identified by a pair of IP addresses and port number. It must be set in the Network Byte Order format , [_htons](#) (e.g. [_htons\(80\)](#)). Can NOT have zero value.

◆ sin_addr

in_addr sin_addr

IP Address of the socket. The IP address is of type [in_addr](#) structure. Can be set to "0" to accept any IP address for server operation. non zero otherwise.

◆ sin_zero

uint8 sin_zero[8]

Padding to make structure the same size as [sockaddr](#).



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Modules

Function

Socket

Modules

socketInit

registerSocketCallback

socket

bind

listen

accept

connect

recv

recvfrom

send

sendto

close

nmi_inet_addr

gethostbyname

sslEnableCertExpirationCheck

setsockopt

getsockopt

m2m_ping_req

Detailed Description

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



socketInit

Socket » Function

Functions

Functions

NMI_API void socketInit (void)

NMI_API void socketDeinit (void)

Detailed Description

The function performs the necessary initializations for the socket library through the following steps:

- A check made by the global variable gbSocketInit, ensuring that initialization for sockets is performed only once, in-order to prevent resetting the socket instances already created in the global socket array (gastrSockets).
- Zero initializations to the global socket array (gastrSockets), which holds the list of TCP sockets.
- Registers the socket (Host Interface)hif callback function through the call to the hif_register_cb function. This facilitates handling all of the socket related functions received through interrupts from the firmware.

Function Documentation

◆ **socketInit()**

NMI_API void socketInit (void)

Parameters

[in] **void**

Returns

void

Remarks

This initialization function must be invoked before any socket operation is performed. No error codes from this initialization function since the socket array is statically allocated based in the maximum number of sockets **MAX_SOCKET** based on the systems capability.

Example

This example demonstrates the use of the socketinit for socket initialization for an mqtt chat application.

```
tstrWifiInitParam param;
int8_t ret;
char topic[strlen(MAIN_CHAT_TOPIC) + MAIN_CHAT_USER_NAME_SIZE + 1];

//Initialize the board.
system_init();

//Initialize the UART console.
configure_console();

// Initialize the BSP.
nm_bsp_init();

-----
// Initialize socket interface.
socketInit();
registerSocketCallback(socket_event_handler, socket_resolve_handler);

// Connect to router.
m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID),
                  MAIN_WLAN_AUTH, (char *)MAIN_WLAN_PSK, M2M_WIFI_CH_ALL);
```

◆ socketDeinit()

NMI_API void socketDeinit (void)

Socket Layer De-initialization.

The function performs the necessary cleanup for the socket library static data It must be invoked as the last any socket operation is performed on any active sockets.



registerSocketCallback

Socket » Function

Functions

Functions

```
NMI_API void registerSocketCallback (tpfAppSocketCb socket_cb, tpfAppResolveCb  
resolve_cb)
```

Detailed Description

Register two callback functions one for asynchronous socket events and the other one for DNS callback registering function. The registered callback functions are used to retrieve information in response to the asynchronous socket functions called.

Function Documentation

♦ registerSocketCallback()

```
NMI_API void registerSocketCallback ( tpfAppSocketCb socket_cb,  
                                    tpfAppResolveCb resolve_cb  
                                )
```

Parameters

[in] **tpfAppSocketCb** Assignment of callback function to the global callback
tpfAppSocketCb gpfAppSocketCb. Delivers socket

messages to the host application. In response to the asynchronous function calls, such as **bind** **listen** **accept** **connect**

[in] **tpfAppResolveCb** Assignment of callback function to the global callback **tpfAppResolveCb** `gpfAppResolveCb`. Used for DNS resolving functionalities. The DNS resolving technique is determined by the application registering the callback. NULL is assigned when, DNS resolution is not required.

Returns

`void`

Remarks

If any of the socket functionalities is not to be used, NULL is passed in as a parameter. It must be invoked after `socketinit` and before other socket layer operations.

Example

This example demonstrates the use of the `registerSocketCallback` to register a socket callback function with DNS resolution CB set to null for a simple UDP server example.

```
tstrWifiInitParam param;
int8_t ret;
struct sockaddr_in addr;

// Initialize the board
system_init();

//Initialize the UART console.
configure_console();

// Initialize the BSP.
nm_bsp_init();

// Initialize socket address structure.
addr.sin_family = AF_INET;
addr.sin_port = _htons(MAIN_WIFI_M2M_SERVER_PORT);
addr.sin_addr.s_addr = _htonl(MAIN_WIFI_M2M_SERVER_IP);

// Initialize Wi-Fi parameters structure.
memset((uint8_t *)&param, 0, sizeof(tstrWifiInitParam));

// Initialize Wi-Fi driver with data and status callbacks.
param.pfAppWifiCb = wifi_cb;
ret = m2m_wifi_init(&param);
if (M2M_SUCCESS != ret) {
    printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
    while (1) {
    }
}

// Initialize socket module
socketInit();
registerSocketCallback(socket_cb, NULL);
```

```
// Connect to router.  
m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID),  
    MAIN_WLAN_AUTH, (char *)MAIN_WLAN_PSK, M2M_WIFI_CH_ALL);
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



socket

Socket » Function

Functions

Functions

NMI_API SOCKET socket (uint16 u16Domain, uint8 u8Type, uint8 u8Flags)

Detailed Description

Synchronous socket allocation function based on the specified socket type. Created sockets are non-blocking and their possible types are either TCP or a UDP sockets. The maximum allowed number of TCP sockets is **TCP_SOCK_MAX** sockets while the maximum number of UDP sockets that can be created simultaneously is **UDP_SOCK_MAX** sockets.

Function Documentation

◆ socket()

```
NMI_API SOCKET socket ( uint16 u16Domain,  
                         uint8 u8Type,  
                         uint8 u8Flags  
                     )
```

Parameters

[in] **u16Domain** Socket family. The only allowed value is AF_INET (IPv4.0) for

TCP/UDP sockets.

[in] **u8Type** Socket type. Allowed values are:

- **SOCK_STREAM**
- **SOCK_DGRAM**

[in] **u8Flags** Used to specify the socket creation flags. It shall be set to zero for normal TCP/UDP sockets. It could be **SOCKET_FLAGS_SSL** if the socket is used for SSL session. The use of the flag **SOCKET_FLAGS_SSL** has no meaning in case of UDP sockets.

Precondition

The **socketInit** function must be called once at the beginning of the application to initialize the socket handler. before any call to the socket function can be made.

See also

**connect bind listen accept recv recvfrom send sendto close setsockopt
getsockopt**

Returns

On successful socket creation, a non-blocking socket type is created and a socket ID is returned In case of failure the function returns a negative value, identifying one of the socket error codes defined. For example: **SOCK_ERR_INVALID** for invalid argument or **SOCK_ERR_MAX_TCP_SOCK** if the number of TCP allocated sockets exceeds the number of available sockets.

Remarks

The socket function must be called a priori to any other related socket functions "e.g. send, recv, close ..etc"

Example

This example demonstrates the use of the socket function to allocate the socket, returning the socket handler to be used for other socket operations. Socket creation is dependent on the socket type.

UDP example

```
SOCKET UdpServerSocket = -1;  
  
UdpServerSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

TCP example

```
static SOCKET tcp_client_socket = -1;  
  
tcp_client_socket = socket(AF_INET, SOCK_STREAM, 0));
```

SSL example

```
static SOCKET ssl_socket = -1;  
ssl_socket = socket(AF_INET, SOCK_STREAM, SOCK_FLAGS_SSL));
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



bind

Socket » Function

Functions

Functions

NMI_API sint8 bind (SOCKET sock, struct sockaddr *pstrAddr, uint8 u8AddrLen)

Detailed Description

Asynchronous bind function associates the provided address and local port to the socket. The function can be used with both TCP and UDP sockets it's mandatory to call the **bind** function before starting any UDP or TCP server operation. Upon socket bind completion, the application will receive a **SOCKET_MSG_BIND** message in the socket callback.

Function Documentation

◆ bind()

```
NMI_API sint8 bind ( SOCKET          sock,
                      struct sockaddr * pstrAddr,
                      uint8            u8AddrLen
                    )
```

Parameters

[in] **sock** Socket ID, must hold a non negative value. A negative value will

return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.

[in] **pstrAddr** Pointer to socket address structure "sockaddr_in" **sockaddr_in**
[in] **u8AddrLen** Size of the given socket address structure in bytes.

Precondition

The socket function must be called to allocate a socket before passing the socket ID to the bind function.

See also

[socket](#) [connect](#) [listen](#) [accept](#) [recv](#) [recvfrom](#) [send](#) [sendto](#)

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID or NULL socket address structure.
- **SOCK_ERR_INVALID** Indicate socket bind failure.

Example

This example demonstrates the call of the bind socket operation after a successful socket operation.

```
struct sockaddr_in addr;
SOCKET udpServerSocket = -1;
int ret = -1;

if(udpServerSocket == -1)
{
    udpServerSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if(udpServerSocket >= 0)
    {
        addr.sin_family      = AF_INET;
        addr.sin_port        = htons(1234);
        addr.sin_addr.s_addr = 0;
        ret = bind(udpServerSocket,(struct sockaddr*)&addr,sizeof(addr));

        if(ret != 0)
        {
            printf("Bind Failed. Error code = %d\n",ret);
            close(udpServerSocket);
        }
    }
    else
    {
        printf("UDP Server Socket Creation Failed\n");
    }
}
```




listen

Socket » Function

Functions

Functions

NMI_API sint8 **listen** (**SOCKET** sock, **uint8** backlog)

Detailed Description

After successful socket binding to an IP address and port on the system, start listening on a passive socket for incoming connections. The socket must be bound on a local port or the listen operation fails. Upon the call to the asynchronous listen function, response is received through the event **SOCKET_MSG_BIND** in the socket callback. A successful listen means the TCP server operation is active. If a connection is accepted, then the application socket callback function is notified with the new connected socket through the event **SOCKET_MSG_ACCEPT**. Hence there is no need to call the **accept** function after calling **listen**.

After a connection is accepted, the user is then required to call the **recv** to receive any packets transmitted by the remote host or to receive notification of socket connection termination.

Function Documentation

◆ **listen()**

NMI_API sint8 **listen** (**SOCKET** sock,
 uint8 backlog

)

Parameters

- [in] **sock** Socket ID, must hold a non negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
- [in] **backlog** Not used by the current implementation.

Precondition

The bind function must be called to assign the port number and IP address to the socket before the listen operation.

See also

[bind](#) [accept](#) [recv](#) [recvfrom](#) [send](#) [sendto](#)

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID.
- **SOCK_ERR_INVALID** Indicate socket listen failure.

Example

This example demonstrates the call of the listen socket operation after a successful socket operation.

```
static void TCP_Socketcallback(SOCKET sock, uint8 u8Msg, void * pvMsg)
{
    int ret = -1;

    switch(u8Msg)
    {
        case SOCKET_MSG_BIND:
            {
                tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg*)pvMsg;
                if(pstrBind != NULL)
                {
                    if(pstrBind->status == 0)
                    {
                        ret = listen(sock, 0);

                    if(ret <0)
                        printf("Listen failure! Error = %d\n",ret);
                }
                else
                {
                    M2M_ERR("bind Failure!\n");
                }
            }
        close(sock);
    }
}
```

```

        }

break;

case SOCKET_MSG_LISTEN:
{
    tstrSocketListenMsg *pstrListen = (tstrSocketListenMsg*)pvMsg;
    if(pstrListen != NULL)
    {
        if(pstrListen->status == 0)
        {
            ret = accept(sock,NULL,0);
        }
    else
        {
            M2M_ERR("listen Failure!\n");
close(sock);
        }
    }
}
break;

case SOCKET_MSG_ACCEPT:
{
    tstrSocketAcceptMsg *pstrAccept = (tstrSocketAcceptMsg*)pvMsg;
    if(pstrAccept->sock >= 0)
    {
        TcpNotificationSocket = pstrAccept->sock;
recv(pstrAccept-
    >sock,gau8RxBuffer,sizeof(gau8RxBuffer),TEST_RECV_TIMEOUT);
    }
else
    {
        M2M_ERR("accept failure\n");
    }
}
break;

default:
break;
}

```



accept

Socket » Function

Functions

Functions

NMI_API sint8 accept (SOCKET sock, struct sockaddr *addr, uint8 *addrlen)

Detailed Description

The function has no current implementation. An empty deceleration is used to prevent errors when legacy application code is used. For recent application use, the accept function can be safer as it has no effect and could be safely removed from any application using it.

Function Documentation

◆ accept()

```
NMI_API sint8 accept ( SOCKET          sock,
                      struct sockaddr * addr,
                      uint8 *           addrlen
                    )
```

Parameters

[in] **sock** Socket ID, must hold a non negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid

argument is passed in.

[in] **addr** Not used in the current implementation.

[in] **addrlen** Not used in the current implementation.

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID.



connect

Socket » Function

Functions

Functions

NMI_API sint8 connect (SOCKET sock, struct sockaddr *pstrAddr, uint8 u8AddrLen)

Detailed Description

Establishes a TCP connection with a remote server. The asynchronous connect function must be called after receiving a valid socket ID from the **socket** function. The application socket callback function is notified of a successful new socket connection through the event **SOCKET_MSG_CONNECT**. A successful connect means the TCP session is active. The application is then required to make a call to the **recv** to receive any packets transmitted by the remote server, unless the application is interrupted by a notification of socket connection termination.

Function Documentation

◆ connect()

```
NMI_API sint8 connect ( SOCKET          sock,
                        struct sockaddr * pstrAddr,
                        uint8            u8AddrLen
                      )
```

Parameters

- [in] **sock** Socket ID, must hold a non negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
- [in] **pstrAddr** Address of the remote server.
- [in] **pstrAddr** Pointer to socket address structure "sockaddr_in" **sockaddr_in**
- [in] **u8AddrLen** Size of the given socket address structure in bytes. Not currently used, implemented for BSD compatibility only.

Precondition

The socket function must be called to allocate a TCP socket before passing the socket ID to the bind function. If the socket is not bound, you do NOT have to call bind before the "connect" function.

See also

[socket](#) [recv](#) [send](#) [close](#)

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID or NULL socket address structure.
- **SOCK_ERR_INVALID** Indicate socket connect failure.

Example

The example demonstrates a TCP application, showing how the asynchronous call to the connect function is made through the main function and how the callback function handles the **SOCKET_MSG_CONNECT** event.

UDP example

```
struct sockaddr_in Serv_Addr;
SOCKET TcpClientSocket = -1;
int ret = -1

TcpClientSocket = socket(AF_INET, SOCK_STREAM, 0);
Serv_Addr.sin_family = AF_INET;
Serv_Addr.sin_port = htons(1234);
Serv_Addr.sin_addr.s_addr = inet_addr(SERVER);
printf("Connected to server via socket %u\n", TcpClientSocket);

do
{
    ret = connect(TcpClientSocket,
        (sockaddr_in*)&Serv_Addr, sizeof(Serv_Addr));
    if(ret != 0)
    {
```

```

        printf("Connection Error\n");
    }
else
{
    printf("Connection successful.\n");
    break;
}
}while(1)

```

TCP example

```

if(u8Msg == SOCKET_MSG_CONNECT)
{
    tstrSocketConnectMsg *pstrConnect = (tstrSocketConnectMsg*)pvMsg;
    if(pstrConnect->s8Error == 0)
    {
        uint8 acBuffer[GROWL_MSG_SIZE];
        uint16 u16MsgSize;

        printf("Connect success!\n");

        u16MsgSize = FormatMsg(u8ClientID, acBuffer);
        send(sock, acBuffer, u16MsgSize, 0);
        recv(pstrNotification->Socket,
            (void*)au8Msg, GROWL_DESCRIPTION_MAX_LENGTH, GROWL_RX_TIMEOUT);
        u8Retry = GROWL_CONNECT_RETRY;
    }
else
{
    M2M_DBG("Connection Failed, Error: %d\n",pstrConnect->s8Error);
    close(pstrNotification->Socket);
}
}

```



recv

Socket » Function

Functions

Functions

```
NMI_API sint16 recv (SOCKET sock, void *pvRecvBuf, uint16 u16BufLen, uint32  
u32Timeoutmsec)
```

Detailed Description

An asynchronous receive function, used to retrieve data from a TCP stream. Before calling the `recv` function, a successful socket connection status must have been received through any of the two socket events [`SOCKET_MSG_CONNECT`] or [`SOCKET_MSG_ACCEPT`], from the socket callback. Hence, indicating that the socket is already connected to a remote host. The application receives the required data in response to this asynchronous call through the reception of the event `SOCKET_MSG_RECV` in the socket callback.

Receiving the `SOCKET_MSG_RECV` message in the callback with zero or negative buffer length indicates the following:

- `SOCK_ERR_NO_ERROR` : Socket connection closed
- `SOCK_ERR_CONN_ABORTED` : Socket connection aborted
- `SOCK_ERR_TIMEOUT` : Socket receive timed out The application code is expected to close the socket through the call to the `close` function upon the appearance of the above mentioned errors.

Function Documentation

◆ recv()

```
NMI_API sint16 recv ( SOCKET sock,  
                      void *    pvRecvBuf,  
                      uint16     u16BufLen,  
                      uint32     u32Timeoutmsec  
)
```

Parameters

[in] sock	Socket ID, must hold a non negative value. A negative value will return a socket error SOCK_ERR_INVALID_ARG . Indicating that an invalid argument is passed in.
[in] pvRecvBuf	Pointer to a buffer that will hold the received data. The buffer is used in the recv callback to deliver the received data to the caller. The buffer must be resident in memory (heap or global buffer).
[in] u16BufLen	The buffer size in bytes.
[in] u32Timeoutmsec	Timeout for the recv function in milli-seconds. If the value is set to ZERO, the timeout will be set to infinite (the recv function waits forever). If the timeout period is elapsed with no data received, the socket will get a timeout error.

Precondition

- The socket function must be called to allocate a TCP socket before passing the socket ID to the recv function.
- The socket in a connected state is expected to receive data through the socket interface.

See also

[socket](#) [connect](#) [bind](#) [listen](#) [recvfrom](#) [close](#)

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID or NULL Recieve buffer.
- **SOCK_ERR_BUFFER_FULL** Indicate socket receive failure.

Example

The example demonstrates a code snippet for the calling of the recv function in the socket

callback upon notification of the accept or connect events, and the parsing of the received data when the SOCKET_MSG_RECV event is received.

```
switch(u8Msg)
{
    case SOCKET_MSG_ACCEPT:
        {
            tstrSocketAcceptMsg *pstrAccept = (tstrSocketAcceptMsg*)pvMsg;
            if(pstrAccept->sock >= 0)
            {
                recv(pstrAccept->sock,gau8RxBuffer,sizeof(gau8RxBuffer),TEST_RECV_TIMEOUT);
            }
            else
            {
                M2M_ERR("accept\n");
            }
        }
        break;

    case SOCKET_MSG_RECV:
        {
            tstrSocketRecvMsg *pstrRx = (tstrSocketRecvMsg*)pvMsg;
            if(pstrRx->s16BufferSize > 0)
            {
                recv(sock,gau8RxBuffer,sizeof(gau8RxBuffer),TEST_RECV_TIMEOUT);
            }
            else
            {
                printf("Socet recv Error: %d\n",pstrRx->s16BufferSize);
                close(sock);
            }
        }
        break;

    default:
        break;
}
```

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



recvfrom

Socket » Function

Functions

Functions

```
NMI_API sint16 recvfrom (SOCKET sock, void *pvRecvBuf, uint16 u16BufLen, uint32  
u32Timeoutmsec)
```

Detailed Description

Receives data from a UDP Socket.

The asynchronous recvfrom function is used to retrieve data from a UDP socket. The socket must already be bound to a local port before a call to the recvfrom function is made (i.e message **SOCKET_MSG_BIND** is received with successful status in the socket callback).

Upon calling the recvfrom function with a successful return code, the application is expected to receive a notification in the socket callback whenever a message is received through the **SOCKET_MSG_RECVFROM** event.

Receiving the **SOCKET_MSG_RECVFROM** message in the callback with zero, indicates that the socket is closed. Whereby a negative buffer length indicates one of the socket error codes such as socket timeout error :

The recvfrom callback can also be used to show the IP address of the remote host that sent the frame by using the "strRemoteAddr" element in the **tstrSocketRecvMsg** structure. (refer to the code example)

Function Documentation

◆ recvfrom()

```
NMI_API sint16 recvfrom ( SOCKET sock,
                           void *     pvRecvBuf,
                           uint16      u16BufLen,
                           uint32      u32TimeoutSeconds
                         )
```

Parameters

[in] sock	Socket ID, must hold a non negative value. A negative value will return a socket error SOCK_ERR_INVALID_ARG . Indicating that an invalid argument is passed in.
[in] pvRecvBuf	Pointer to a buffer that will hold the received data. The buffer shall be used in the recv callback to deliver the received data to the caller. The buffer must be resident in memory (heap or global buffer).
[in] u16BufLen	The buffer size in bytes.
[in] u32TimeoutSeconds	Timeout for the recv function in milli-seconds. If the value is set to ZERO, the timeout will be set to infinite (the recv function waits forever).

Precondition

- The socket function must be called to allocate a UDP socket before passing the socket ID to the recvfrom function.
- The socket corresponding to the socket ID must be successfully bound to a local port through the call to a **bind** function.

See also

[socket bind close](#)

Returns

The function returns ZERO for successful operations and a negative value otherwise.

The possible error values are:

- **SOCK_ERR_NO_ERROR** Indicating that the operation was successful.
- **SOCK_ERR_INVALID_ARG** Indicating passing invalid arguments such as negative socket ID or NULL Receive buffer.
- **SOCK_ERR_BUFFER_FULL** Indicate socket receive failure.

Example

The example demonstrates a code snippet for the calling of the recvfrom function in the socket callback upon notification of a successful bind event, and the parsing of the received data when the SOCKET_MSG_RECVFROM event is received.

```

switch(u8Msg)
{
    case SOCKET_MSG_BIND:
        {
            tstrSocketBindMsg *pstrBind = (tstrSocketBindMsg*)pvMsg;
            if(pstrBind != NULL)
                {
                    if(pstrBind->status == 0)
                        {
                            recvfrom(sock, gau8SocketTestBuffer, TEST_BUFFER_SIZE, 0);
                        }
                    else
                        {
                            M2M_ERR("bind\n");
                        }
                }
            break;

    case SOCKET_MSG_RECVFROM:
        {
            tstrSocketRecvMsg *pstrRx = (tstrSocketRecvMsg*)pvMsg;
            if(pstrRx->s16BufferSize > 0)
                {
                    //get the remote host address and port number
                    uint16 u16port = pstrRx->strRemoteAddr.sin_port;
                    uint32 strRemoteHostAddr = pstrRx->strRemoteAddr.sin_addr.s_addr;

                    printf("Received frame with size = %d.\nHost address=%x,
                           Port number = %d\n\n",pstrRx->s16BufferSize,strRemoteHostAddr,
                           u16port);

                    ret =
                    recvfrom(sock,gau8SocketTestBuffer,sizeof(gau8SocketTestBuffer),TEST_RECV_TIMEOUT);
                }
            else
                {
                    printf("Socet recv Error: %d\n",pstrRx->s16BufferSize);
                    ret = close(sock);
                }
            break;
        default:
            break;
        }
}

```



send

Socket » Function

Functions

Functions

```
NMI_API sint16 send (SOCKET sock, void *pvSendBuffer, uint16 u16SendLength, uint16  
u16Flags)
```

Detailed Description

Asynchronous sending function, used to send data on a TCP/UDP socket.

Called by the application code when there is outgoing data available required to be sent on a specific socket handler. The only difference between this function and the similar [sendto](#) function, is the type of socket the data is sent on and the parameters passed in. [send](#) function is most commonly called for sockets in a connected state. After the data is sent, the socket callback function registered using [registerSocketCallback\(\)](#), is expected to receive an event of type [SOCKET_MSG_SEND](#) holding information containing the number of data bytes sent.

Function Documentation

◆ send()

```
NMI_API sint16 send ( SOCKET sock,  
void * pvSendBuffer,  
uint16 u16SendLength,
```

```
    uint16  u16Flags
```

```
)
```

Parameters

- [in] **sock** Socket ID, must hold a non negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.
- [in] **pvSendBuffer** Pointer to a buffer holding data to be transmitted.
- [in] **u16SendLength** The buffer size in bytes.
- [in] **u16Flags** Not used in the current implementation.

Precondition

Sockets must be initialized using `socketInit`.

For TCP Socket:

Must use a successfully connected Socket (so that the intended recipient address is known ahead of sending the data). Hence this function is expected to be called after a successful socket connect operation(in client case or accept in the the server case).

For UDP Socket:

UDP sockets most commonly use **sendto** function, where the destination address is defined. However, in-order to send outgoing data using the **send** function, at least one successful call must be made to the **sendto** function a priori the consecutive calls to the **send** function, to ensure that the destination address is saved in the firmware.

See also

[socketInit](#) [recv](#) [sendto](#) [socket](#) [connect](#) [accept](#) [sendto](#)

Warning

`u16SendLength` must not exceed **SOCKET_BUFFER_MAX_LENGTH**.

Use a valid socket identifier through the a prior call to the **socket** function. Must use a valid buffer pointer. Successful completion of a call to **send()** does not guarantee delivery of the message, A negative return value indicates only locally-detected errors

Returns

The function shall return **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.



sendto

Socket » Function

Functions

Functions

```
NMI_API sint16 sendto (SOCKET sock, void *pvSendBuffer, uint16 u16SendLength, uint16
flags, struct sockaddr *pstrDestAddr, uint8 u8AddrLen)
```

Detailed Description

Asynchronous sending function, used to send data on a UDP socket. Called by the application code when there is data required to be sent on a UDP socket handler. The application code is expected to receive data from a successful bounded socket node. The only difference between this function and the similar [send](#) function, is the type of socket the data is received on. This function works only with UDP sockets. After the data is sent, the socket callback function registered using [registerSocketCallback\(\)](#), is expected to receive an event of type [SOCKET_MSG_SENDTO](#).

Function Documentation

◆ sendto()

```
NMI_API sint16 sendto ( SOCKET sock,
void *pvSendBuffer,
uint16 u16SendLength,
```

```
        uint16          flags,  
        struct sockaddr * pstrDestAddr,  
        uint8           u8AddrLen  
    )
```

Parameters

[in] sock	Socket ID, must hold a non negative value. A negative value will return a socket error SOCK_ERR_INVALID_ARG . Indicating that an invalid argument is passed in.
[in] pvSendBuffer	Pointer to a buffer holding data to be transmitted. A NULL value will return a socket error SOCK_ERR_INVALID_ARG . Indicating that an invalid argument is passed in.
[in] u16SendLength	The buffer size in bytes. It must not exceed SOCKET_BUFFER_MAX_LENGTH .
[in] flags	Not used in the current implementation
[in] pstrDestAddr	The destination address.
[in] u8AddrLen	Destination address length in bytes. Not used in the current implementation, only included for BSD compatibility.

Precondition

Sockets must be initialized using `socketInit`.

See also

`socketInit` `recvfrom` `sendto` `socket` `connect` `accept` `send`

Warning

`u16SendLength` must not exceed **SOCKET_BUFFER_MAX_LENGTH**. Use a valid socket (returned from `socket`). A valid buffer pointer must be used (not NULL). Successful completion of a call to `sendto()` does not guarantee delivery of the message, A negative return value indicates only locally-detected errors

Returns

The function returns **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.



close

Socket » Function

Functions

Functions

NMI_API sint8 close (SOCKET sock)

Detailed Description

Synchronous close function, releases all the socket assigned resources.

Function Documentation

◆ close()

NMI_API sint8 close (SOCKET sock)

Parameters

[in] **sock** Socket ID, must hold a non negative value. A negative value will return a socket error **SOCK_ERR_INVALID_ARG**. Indicating that an invalid argument is passed in.

Precondition

Sockets must be initialized through the call of the socketInit function. **close** is called only for valid socket identifiers created through the **socket** function.

Warning

If **close** is called while there are still pending messages (sent or received) they will be discarded.

See also

[socketInit](#) [socket](#)

Returns

The function returned **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.



nmi_inet_addr

Socket » Function

Functions

Functions

NMI_API uint32 nmi_inet_addr (char *pclpAddr)

Detailed Description

Synchronous function which returns a BSD socket compliant Internet Protocol (IPv4) socket address. This IPv4 address in the input string parameter could either be specified as a host name, or as a numeric string representation like n.n.n.n known as the IPv4 dotted-decimal format (i.e. "192.168.10.1"). This function is used whenever an ip address needs to be set in the proper format (i.e. for the **tstrM2MIPConfig** structure).

Function Documentation

◆ nmi_inet_addr()

NMI_API uint32 nmi_inet_addr (char * pclpAddr)

Parameters

[in] pclpAddr A null terminated string containing the IP address in IPv4 dotted-decimal address.

Returns

Unsigned 32-bit integer representing the IP address in Network byte order (eg. "192.168.10.1" will be expressed as 0x010AA8C0).



gethostbyname

Socket » Function

Functions

Functions

NMI_API sint8 gethostbyname (uint8 *pcHostName)

Detailed Description

Asynchronous DNS resolving function. This function use DNS to resolve a domain name into the corresponding IP address. A call to this function will cause a DNS request to be sent and the response will be delivered to the DNS callback function registered using [registerSocketCallback\(\)](#)

Function Documentation

- [gethostbyname\(\)](#)

NMI_API sint8 gethostbyname (uint8 * pcHostName)

Parameters

[in] **pcHostName** NULL terminated string containing the domain name for the remote host. Its size must not exceed [HOSTNAME_MAX_SIZE](#).

See also

[registerSocketCallback](#)

Warning

Successful completion of a call to [gethostbyname\(\)](#) does not guarantee success of the DNS request, a negative return value indicates only locally-detected errors

Returns

- [SOCK_ERR_NO_ERROR](#)
- [SOCK_ERR_INVALID_ARG](#)

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by





sslEnableCertExpirationCheck

Socket » Function

Functions

Functions

NMI_API sint8 sslEnableCertExpirationCheck (tenuSslCertExpSettings
enuValidationSetting)

Detailed Description

Configure the behavior of the SSL Library for Certificate Expiry Validation.

Function Documentation

◆ sslEnableCertExpirationCheck()

NMI_API sint8
sslEnableCertExpirationCheck (tenuSslCertExpSettings enuValidationSetting)

Parameters

[in] **enuValidationSetting** See **tenuSslCertExpSettings** for details.

Returns

- **SOCK_ERR_NO_ERROR** for successful operation and negative error code otherwise.

See also

[tenuSslCertExpSettings](#)

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by

 1.8.13



setsockopt

Socket » Function

Functions

Functions

```
NMI_API sint8 setsockopt (SOCKET socket, uint8 u8Level, uint8 option_name, const void  
*option_value, uint16 u16OptionLen)
```

Detailed Description

The **setsockopt()** function shall set the option specified by the option_name argument, at the protocol level specified by the level argument, to the value pointed to by the option_value argument for the socket specified by the socket argument.

Possible protocol level values supported are **SOL_SOCKET** and **SOL_SSL_SOCKET**. Possible options when the protocol level is **SOL_SOCKET** :

SO_SET_UDP_SEND_CALLBACK Enable/Disable callback messages for **sendto()**. Since UDP is unreliable by default the user maybe interested (or not) in receiving a message of **SOCKET_MSG_SENDTO** for each call of **sendto()**. Enabled if option value equals TRUE, disabled otherwise.

IP_ADD_MEMBERSHIP Valid for UDP sockets. This option is used to receive frames sent to a multicast group. option_value shall be a pointer to Unsigned 32-bit integer containing the multicast IPv4 address.

Valid for UDP sockets. This option is used to stop receiving frames sent to a multicast group. option_value shall be a

pointer to Unsigned 32-bit integer containing the multicast IPv4 address.

Possible options when the protocol level is **SOL_SSL_SOCKET**

SO_SSL_BYPASS_X509_VERIF

Allow an opened SSL socket to bypass the X509 certificate verification process. It is highly recommended **NOT** to use this socket option in production software applications. The option is supported for debugging and testing purposes. The option value should be casted to int type and it is handled as a boolean flag.

SO_SSL_SNI

Set the Server Name Indicator (SNI) for an SSL socket. The SNI is a null terminated string containing the server name associated with the connection. It must not exceed the size of **HOSTNAME_MAX_SIZE**.

SO_SSL_ENABLE_SESSION_CACHING

This option allows the TLS to cache the session information for fast TLS session establishment in future connections using the TLS Protocol session resume features.

Function Documentation

◆ setsockopt()

```
NMI_API sint8 setsockopt( SOCKET    socket,
                           uint8      u8Level,
                           uint8      option_name,
                           const void * option_value,
                           uint16     u16OptionLen
                         )
```

Parameters

- [in] **sock** Socket handler.
- [in] **level** protocol level. See description above.
- [in] **option_name** option to be set. See description above.
- [in] **option_value** pointer to user provided value.
- [in] **option_len** length of the option value in bytes.

[in] **option_len** length of the option value in bytes.

Returns

The function shall return **SOCK_ERR_NO_ERROR** for successful operation and a negative value (indicating the error) otherwise.

See also

[SOL_SOCKET](#), [SOL_SSL_SOCKET](#), [IP_ADD_MEMBERSHIP](#),
[IP_DROP_MEMBERSHIP](#)



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Functions

getsockopt

Socket » Function

Functions

```
NMI_API sint8 getsockopt (SOCKET sock, uint8 u8Level, uint8 u8OptName, const void  
*pvOptValue, uint8 *pu8OptLen)
```

Detailed Description

Get socket options retrieves This Function isn't implemented yet but this is the form that will be released later.

Function Documentation

♦ getsockopt()

```
sint8 getsockopt ( SOCKET sock,  
                    uint8      u8Level,  
                    uint8      u8OptName,  
                    const void * pvOptValue,  
                    uint8 *    pu8OptLen  
                )
```

Parameters

[in] **sock** Socket Identifier.

[in] **u8Level** The protocol level of the option.

[in] **u8OptName** The u8OptName argument specifies a single option to get.

[out] **pvOptValue** The pvOptValue argument contains pointer to a buffer containing the option value.

[out] **pu8OptLen** Option value buffer length.

Returns

The function shall return ZERO for successful operation and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



m2m_ping_req

Socket » Function

Functions

Functions

NMI_API sint8 m2m_ping_req (uint32 u32DstIP, uint8 u8TTL, tpfPingCb fpPingCb)

Detailed Description

The function sends ping request to the given IP Address.

Function Documentation

◆ m2m_ping_req()

```
NMI_API sint8 m2m_ping_req ( uint32      u32DstIP,  
                           uint8       u8TTL,  
                           tpfPingCb  fpPingCb  
                         )
```

Parameters

[in] **u32DstIP** Target Destination IP Address for the ping request. It must be represented in Network byte order. The function nmi_inet_addr could be used to translate the dotted decimal notation IP to its Network bytes order integer representative.

[in] **u8TTL** IP TTL value for the ping request. If set to ZERO, the default value SHALL be used.

[in] **fpPingCb** Callback will be called to deliver the ping statistics.

See also

[nmi_inet_addr](#)

Returns

The function returns **M2M_SUCCESS** for successful operations and a negative value otherwise.

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Data Structures

Here are the data structures with brief descriptions:

C in_addr	IPv4 address representation
C sockaddr	Generic socket address structure
C sockaddr_in	Socket address structure for IPV4 addresses. Used to specify socket address information to which to connect to. Can be cast to sockaddr structure
C tstr1xAuthCredentials	Credentials for the user to authenticate with the AAA server (WPA-Enterprise Mode IEEE802.1x)
C tstrCryptoResp	Crypto response
C tstrEthInitParam	Structure to hold Ethernet interface parameters. Structure is to be defined and have its attributes set, based on the application's functionality before a call is made to initialize the Wi-Fi operations by calling the m2m_wifi_init function. This structure is part of the Wi-Fi configuration structure tstrWifiInitParam . Applications shouldn't need to define this structure, if the bypass mode is not defined
C tstrM2MAPConfig	AP Configuration
C tstrM2mBatteryVoltage	Battery Voltage
C tstrM2mClientState	PS Client State
C tstrM2MConnInfo	M2M Provisioning Information obtained from the HTTP Provisioning server
C tstrM2MDefaultConnResp	Response error of the m2m_default_connect
C tstrM2MDeviceNameConfig	Device name
C tstrM2mEnableLogs	Enable Firmware logs
C tstrM2MIPConfig	Static IP configuration
C tstrM2mlpCtrlBuf	Structure holding the incoming buffer's data size information, indicating the data size of the buffer and the remaining buffer's data size . The data of the buffer which holds the packet sent to the host when in the bypass mode, is placed in the tstrEthInitParam

	structure in the au8ethRcvBuf attribute. This following information is retrieved in the host when an event M2M_WIFI_RESP_ETHERNET_RX_PACKET is received in the Wi-Fi callback function tpfAppWifiCb
C tstrM2mlpRsvdPkt	Received Packet Size and Data Offset
C tstrM2mLsnInt	Listen interval
C tstrM2MMulticastMac	M2M add/remove multi-cast mac address
C tstrM2MP2PConnect	Set the device to operate in the Wi-Fi Direct (P2P) mode
C tstrM2MProvisionInfo	M2M Provisioning Information obtained from the HTTP Provisioning server
C tstrM2MProvisionModeConfig	M2M Provisioning Mode Configuration
C tstrM2mPsType	Power Save Configuration
C tstrM2mPwrMode	
C tstrM2mPwrState	Power Mode
C tstrM2mReqScanResult	Scan Result Request
C tstrM2MScan	Wi-Fi Scan Request
C tstrM2mScanDone	Wi-Fi Scan Result
C tstrM2MScanOption	Scan options and configurations
C tstrM2MScanRegion	Wi-Fi channel regulation region information
C tstrM2Mservercmd	PS Server CMD
C tstrM2mServerInit	PS Server initialization
C tstrM2mSetMacAddress	Sets the MAC address from application. The WINC load the mac address from the effuse by default to the WINC configuration memory, but that function is used to let the application overwrite the configuration memory with the mac address from the host
C tstrM2mSlpReqTime	Manual power save request sleep time
C tstrM2mTxPwrLevel	Tx power level
C tstrM2mWifiConnect	Wi-Fi Connect Request
C tstrM2mWifiGainsParams	Gain Values
C tstrM2MWifiMonitorModeCtrl	Wi-Fi Monitor Mode Filter
C tstrM2MWifiRxPacketInfo	Wi-Fi RX Frame Header
C tstrM2mWifiscanResult	Wi-Fi Scan Result
C tstrM2MWifiSecInfo	Authentication credentials to connect to a Wi-Fi network
C tstrM2mWifiStateChanged	Wi-Fi Connection State
C tstrM2MWifiTxPacketInfo	Wi-Fi TX Packet Info
C tstrM2mWifiWepParams	WEP security key parameters
C tstrM2MWPSConnect	WPS Configuration parameters
C tstrM2MWPSInfo	WPS Result
C tstrOtaControlSec	Control section structure is used to define the working

	image and the validity of the roll-back image and its offset, also both firmware versions is kept in that structure
C tstrOtaInitHdr	OTA Image Header
C tstrOtaUpdateInfo	OTA Update Information
C tstrOtaUpdateStatusResp	OTA Update Information
C tstrPrng	M2M Request PRNG
C tstrSocketAcceptMsg	Socket accept status
C tstrSocketBindMsg	Socket bind status
C tstrSocketConnectMsg	Socket connect status
C tstrSocketListenMsg	Socket listen status
C tstrSocketRecvMsg	Socket recv status
C tstrSslSetActiveCsList	
C tstrSystemTime	Used for time storage
C tstrTlsCrlEntry	Certificate data for inclusion in a revocation list (CRL)
C tstrTlsCrlInfo	Certificate revocation list details
C tstrTlsSrvSecFileEntry	This struct contains a TLS certificate
C tstrTlsSrvSecHdr	This struct contains a set of TLS certificates
C tstrWifiInitParam	Structure, holding the Wi-fi configuration attributes such as the wi-fi callback , monitoring mode callback and Ethernet parameter initialization structure. Such configuration parameters are required to be set before calling the wi-fi initialization function m2m_wifi_init . pfAppWifiCb attribute must be set to handle the wi-fi callback operations. pfAppMonCb attribute, is optional based on whether the application requires the monitoring mode configuration, and can there not be set before the initialization. strEthInitParam structure, is another optional configuration based on whether the bypass mode is set
C tuniM2MWifiAuth	Wi-Fi Security Parameters for all supported security modes



tstrEthInitParam Struct Reference

Data Fields

Data Fields

tpfAppWifiCb **pfAppWifiCb**

tpfAppEthCb **pfAppEthCb**

uint8 * au8ethRcvBuf

uint16 u16ethRcvBufSize

uint8 u8EthernetEnable

uint8 __PAD8__

Detailed Description

Structure to hold Ethernet interface parameters. Structure is to be defined and have its attributes set, based on the application's functionality before a call is made to initialize the Wi-Fi operations by calling the [m2m_wifi_init](#) function. This structure is part of the Wi-Fi configuration structure [tstrWifilnParam](#). Applications shouldn't need to define this structure, if the bypass mode is not defined.

See also

[tpfAppEthCb](#) [tpfAppWifiCb](#) [m2m_wifi_init](#)

Warning

Make sure that application defines ETH_MODE before using [tstrEthInitParam](#).

Field Documentation

- ◆ **pfAppWifiCb**

```
tpfAppWifiCb pfAppWifiCb
```

Callback for wifi notifications.

- ◆ **pfAppEthCb**

```
tpfAppEthCb pfAppEthCb
```

Callback for Ethernet interface.

- ◆ **au8ethRcvBuf**

```
uint8* au8ethRcvBuf
```

Pointer to Receive Buffer of Ethernet Packet

- ◆ **u16ethRcvBufSize**

```
uint16 u16ethRcvBufSize
```

Size of Receive Buffer for Ethernet Packet

- ◆ **u8EthernetEnable**

```
uint8 u8EthernetEnable
```

Enable Ethernet mode flag

- ◆ **__PAD8__**

`uint8 __PAD8__`

Padding

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by  1.8.13



tstrM2mlpCtrlBuf Struct Reference

Data Fields

Data Fields

`uint16 u16DataSize`

`uint16 u16RemainigDataSize`

Detailed Description

Structure holding the incoming buffer's data size information, indicating the data size of the buffer and the remaining buffer's data size . The data of the buffer which holds the packet sent to the host when in the bypass mode, is placed in the [tstrEthInitParam](#) structure in the au8ethRcvBuf attribute. This following information is retrieved in the host when an event

[M2M_WIFI_RESP_ETHERNET_RX_PACKET](#) is received in the Wi-Fi callback function tpfAppWifiCb.

The application is expected to use this structure's information to determine if there is still incoming data to be received from the firmware.

See also

`tpfAppEthCb tstrEthInitParam`

Warning

Make sure that ETHERNET/bypass mode is defined before using [tstrM2mlpCtrlBuf](#)

Field Documentation

- ◆ **u16DataSize**

```
uint16 u16DataSize
```

Size of the received data in bytes.

- ◆ **u16RemainingDataSize**

```
uint16 u16RemainingDataSize
```

Size of the remaining data bytes to be delivered to host.



WINC1500

IoT

Software

APIs 19.5.2

WINC Software API Reference
Manual

Data Fields

tstrWifiInitParam Struct Reference

Data Fields

tpfAppWifiCb [pfAppWifiCb](#)

tpfAppMonCb [pfAppMonCb](#)

[tstrEthInitParam](#) [strEthInitParam](#)

Detailed Description

Structure, holding the Wi-fi configuration attributes such as the wi-fi callback , monitoring mode callback and Ethernet parameter initialization structure. Such configuration parameters are required to be set before calling the wi-fi initialization function [m2m_wifi_init](#). [pfAppWifiCb](#) attribute must be set to handle the wi-fi callback operations. [pfAppMonCb](#) attribute, is optional based on whether the application requires the monitoring mode configuration, and can there not be set before the initialization. [strEthInitParam](#) structure, is another optional configuration based on whether the bypass mode is set.

See also

[tpfAppEthCb](#) [tpfAppMonCb](#) [tstrEthInitParam](#)

Field Documentation

- ◆ pfAppWifiCb

```
tpfAppWifiCb pfAppWifiCb
```

Callback for Wi-Fi notifications.

- ◆ pfAppMonCb

```
tpfAppMonCb pfAppMonCb
```

Callback for monitoring interface.

- ◆ strEthInitParam

```
tstrEthInitParam strEthInitParam
```

Structure to hold Ethernet interface parameters.



Here is a list of all struct and union fields with links to the structures/unions they belong to:

- `__PAD16__` : [tstrM2mBatteryVoltage](#) , [tstrM2mLsnInt](#) , [tstrM2mPsType](#) ,
[tstrM2mScanDone](#) , [tstrM2MScanRegion](#) , [tstrM2mSetMacAddress](#) ,
[tstrM2MWifiMonitorModeCtrl](#) , [tstrM2mWifiStateChanged](#) , [tstrPrng](#)
- `__PAD24__` : [tstrCryptoResp](#) , [tstrM2MAPConfig](#) , [tstrM2mClientState](#) ,
[tstrM2MConnInfo](#) , [tstrM2MDefaultConnResp](#) , [tstrM2mEnableLogs](#) ,
[tstrM2MP2PConnect](#) , [tstrM2MProvisionModeConfig](#) , [tstrM2mPwrMode](#) ,
[tstrM2mReqScanResult](#) , [tstrM2Mservercmd](#) , [tstrM2mServerInit](#) ,
[tstrM2mTxPwrLevel](#) , [tstrM2MWifiRxPacketInfo](#) , [tstrM2mWifiWepParams](#) ,
[tstrM2MWPSConnect](#) , [tstrTlsCrlEntry](#)
- `__PAD8__` : [tstrEthInitParam](#) , [tstrM2MMulticastMac](#) , [tstrOtaUpdateInfo](#) ,
[tstrSystemTime](#)
- `__PAD__` : [tstrM2mWifiConnect](#) , [tstrM2MWifiSecInfo](#)
- `__RSVD8__` : [tstrM2MScan](#)
- `__PAD16__` : [tstrOtaUpdateStatusResp](#)
- `__PAD8__` : [tstrM2mWiifiscanResult](#)



Here is a list of all struct and union fields with links to the structures/unions they belong to:

- a -

- acFileName : [tstrTlsSrvSecFileEntry](#)
- acHttpServerDomainName : [tstrM2MProvisionModeConfig](#)
- acPinNumber : [tstrM2MWPSConnect](#)
- acSSID : [tstrM2MConnInfo](#)
- astrEntries : [tstrTlsSrvSecHdr](#)
- astrTlsCrl : [tstrTlsCrlInfo](#)
- au8BSSID : [tstrM2MWifiMonitorModeCtrl](#) , [tstrM2MWifiRxPacketInfo](#) , [tstrM2mWifiscanResult](#)
- au8Data : [tstrTlsCrlEntry](#)
- au8DeviceName : [tstrM2MDeviceNameConfig](#)
- au8DHCPServerIP : [tstrM2MAPConfig](#)
- au8DstMacAddress : [tstrM2MWifiMonitorModeCtrl](#) , [tstrM2MWifiRxPacketInfo](#)
- au8ethRcvBuf : [tstrEthInitParam](#)
- au8IPAddr : [tstrM2MConnInfo](#)
- au8Key : [tstrM2MAPConfig](#)
- au8Mac : [tstrM2mSetMacAddress](#)
- au8MACAddress : [tstrM2MConnInfo](#)
- au8macaddress : [tstrM2MMulticastMac](#)
- au8Passwd : [tstr1xAuthCredentials](#)
- au8Password : [tstrM2MProvisionInfo](#)
- au8PSK : [tstrM2MWPSInfo](#) , [tuniM2MWifiAuth](#)
- au8SecStartPattern : [tstrTlsSrvSecHdr](#)
- au8SrcMacAddress : [tstrM2MWifiMonitorModeCtrl](#) , [tstrM2MWifiRxPacketInfo](#)
- au8SSID : [tstrM2MAPConfig](#) , [tstrM2MProvisionInfo](#) , [tstrM2mWifiConnect](#) , [tstrM2mWifiscanResult](#) , [tstrM2MWPSInfo](#)
- au8UserName : [tstr1xAuthCredentials](#)
- au8WepKey : [tstrM2MAPConfig](#) , [tstrM2mWifiWepParams](#)



Here is a list of all struct and union fields with links to the structures/unions they belong to:

- p -

- pfAppEthCb : [tstrEthInitParam](#)
- pfAppMonCb : [tstrWifiInitParam](#)
- pfAppWifiCb : [tstrEthInitParam](#) , [tstrWifiInitParam](#)
- pu8Buffer : [tstrSocketRecvMsg](#)
- pu8RngBuff : [tstrPrng](#)

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by [doxygen](#) 1.8.13



Here is a list of all struct and union fields with links to the structures/unions they belong to:

- S -

- s16BufferSize : [tstrSocketRecvMsg](#)
- s8Error : [tstrSocketConnectMsg](#)
- s8ErrorCode : [tstrM2MDefaultConnResp](#)
- s8Resp : [tstrCryptoResp](#)
- s8RSSI : [tstrM2MConnInfo](#) , [tstrM2MWifiRxPacketInfo](#)
- s8rssi : [tstrM2mWifiscanResult](#)
- s8RssiThresh : [tstrM2MScanOption](#)
- s8ScanState : [tstrM2mScanDone](#)
- s_addr : [in_addr](#)
- sa_data : [sockaddr](#)
- sa_family : [sockaddr](#)
- sin_addr : [sockaddr_in](#)
- sin_family : [sockaddr_in](#)
- sin_port : [sockaddr_in](#)
- sin_zero : [sockaddr_in](#)
- sock : [tstrSocketAcceptMsg](#) , [tstrSocketConnectMsg](#)
- status : [tstrSocketBindMsg](#) , [tstrSocketListenMsg](#)
- strAddr : [tstrSocketAcceptMsg](#)
- strApConfig : [tstrM2MProvisionModeConfig](#)
- strCred1x : [tuniM2MWifiAuth](#)
- strEthInitParam : [tstrWifiInitParam](#)
- strRemoteAddr : [tstrSocketRecvMsg](#)
- strSec : [tstrM2mWifiConnect](#)
- strWepInfo : [tuniM2MWifiAuth](#)



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

Here is a list of all struct and union fields with links to the structures/unions they belong to:

- u -

- u16BattVolt : [tstrM2mBatteryVoltage](#)
- u16Ch : [tstrM2mWifiConnect](#)
- u16DataLength : [tstrM2MWifiRxPacketInfo](#)
- u16DataSize : [tstrM2mIpCtrlBuf](#)
- u16ethRcvBufSize : [tstrEthInitParam](#)
- u16FrameLength : [tstrM2MWifiRxPacketInfo](#)
- u16HeaderLength : [tstrM2MWifiTxPacketInfo](#)
- u16LsnInt : [tstrM2mLsnInt](#)
- u16PacketSize : [tstrM2MWifiTxPacketInfo](#)
- u16PassiveScanTime : [tstrM2MScan](#)
- u16PktOffset : [tstrM2mIpRsvdPkt](#)
- u16PktSz : [tstrM2mIpRsvdPkt](#)
- u16PrngSize : [tstrPrng](#)
- u16RemainigDataSize : [tstrM2mIpCtrlBuf](#)
- u16RemainingSize : [tstrSocketRecvMsg](#)
- u16ScanRegion : [tstrM2MScanRegion](#)
- u16Year : [tstrSystemTime](#)
- u32CsBMP : [tstrSslSetActiveCsList](#)
- u32DataRateKbps : [tstrM2MWifiRxPacketInfo](#)
- u32DhcpLeaseTime : [tstrM2MIPConfig](#)
- u32DNS : [tstrM2MIPConfig](#)
- u32FileAddr : [tstrTlsSrvSecFileEntry](#)
- u32FileSize : [tstrTlsSrvSecFileEntry](#)
- u32Gateway : [tstrM2MIPConfig](#)
- u32nEntries : [tstrTlsSrvSecHdr](#)
- u32NextWriteAddr : [tstrTlsSrvSecHdr](#)
- u32OtaControlSecCrc : [tstrOtaControlSec](#)
- u32OtaCortusAppRollbackOffset : [tstrOtaControlSec](#)

- u32OtaCortusAppRollbackValidSts : **tstrOtaControlSec**
- u32OtaCortusAppRollbackVer : **tstrOtaControlSec**
- u32OtaCortusAppWorkingOffset : **tstrOtaControlSec**
- u32OtaCortusAppWorkingValidSts : **tstrOtaControlSec**
- u32OtaCortusAppWorkingVer : **tstrOtaControlSec**
- u32OtaCurrentworkingImgFirmwareVer : **tstrOtaControlSec**
- u32OtaCurrentworkingImgOffset : **tstrOtaControlSec**
- u32OtaFormatVersion : **tstrOtaControlSec**
- u32OtaLastCheckTime : **tstrOtaControlSec**
- u32OtaMagicValue : **tstrOtaControlSec , tstrOtaInitHdr**
- u32OtaPayloadSzie : **tstrOtaInitHdr**
- u32OtaRollbackImageOffset : **tstrOtaControlSec**
- u32OtaRollbackImageValidStatus : **tstrOtaControlSec**
- u32OtaRollbackImgFirmwareVer : **tstrOtaControlSec**
- u32OtaSequenceNumber : **tstrOtaControlSec**
- u32SleepTime : **tstrM2mSlpReqTime**
- u32StaticIP : **tstrM2MIPConfig**
- u32SubnetMask : **tstrM2MIPConfig**
- u8AddRemove : **tstrM2MMulticastMac**
- u8AuthType : **tstrM2mWifiscanResult , tstrM2MWPSInfo**
- u8BcastEn : **tstrM2mPsType**
- u8ch : **tstrM2mWifiscanResult**
- u8Ch : **tstrM2MWPSInfo**
- u8Channel : **tstrM2mServerInit**
- u8ChannelID : **tstrM2MWifiMonitorModeCtrl**
- u8ChNum : **tstrM2MScan**
- u8CipherType : **tstrM2MWifiRxPacketInfo**
- u8cmd : **tstrM2Mservercmd**
- u8CrlType : **tstrTlsCrlInfo**
- u8CurrState : **tstrM2mWifiStateChanged**
- u8DataLen : **tstrTlsCrlEntry**
- u8Day : **tstrSystemTime**
- u8DownloadUrlOffset : **tstrOtaUpdateInfo**
- u8DownloadUrlSize : **tstrOtaUpdateInfo**
- u8Enable : **tstrM2mEnableLogs**
- u8EnableRedirect : **tstrM2MProvisionModeConfig**
- u8EnRecvHdr : **tstrM2MWifiMonitorModeCtrl**
- u8ErrCode : **tstrM2mWifiStateChanged**
- u8EthernetEnable : **tstrEthInitParam**
- u8FrameSubtype : **tstrM2MWifiMonitorModeCtrl , tstrM2MWifiRxPacketInfo**
- u8FrameType : **tstrM2MWifiMonitorModeCtrl , tstrM2MWifiRxPacketInfo**
- u8HeaderLength : **tstrM2MWifiRxPacketInfo**
- u8Hour : **tstrSystemTime**
- u8Index : **tstrM2mReqScanResult**

- u8index : **tstrM2mWifiscanResult**
- u8KeyIndx : **tstrM2MAPConfig** , **tstrM2mWifiWepParams**
- u8KeySz : **tstrM2MAPConfig** , **tstrM2mWifiWepParams**
- u8ListenChannel : **tstrM2MAPConfig** , **tstrM2MP2PConnect**
- u8Minute : **tstrSystemTime**
- u8Month : **tstrSystemTime**
- u8NcdRequiredUpgrade : **tstrOtaUpdateInfo**
- u8NcdUpgradeVersion : **tstrOtaUpdateInfo**
- u8NcfCurrentVersion : **tstrOtaUpdateInfo**
- u8NcfUpgradeVersion : **tstrOtaUpdateInfo**
- u8NoSaveCred : **tstrM2mWifiConnect**
- u8NumofCh : **tstrM2mScanDone**
- u8NumOfSlot : **tstrM2MScanOption**
- u8OtaUpdateStatus : **tstrOtaUpdateStatusResp**
- u8OtaUpdateStatusType : **tstrOtaUpdateStatusResp**
- u8PPAGFor11B : **tstrM2mWifiGainsParams**
- u8PPAGFor11GN : **tstrM2mWifiGainsParams**
- u8Priority : **tstrM2MWifiRxPacketInfo**
- u8ProbesPerSlot : **tstrM2MScanOption**
- u8PsType : **tstrM2mPsType**
- u8PwrMode : **tstrM2mPwrMode**
- u8Rsv1 : **tstrTlsCrlInfo**
- u8Rsv2 : **tstrTlsCrlInfo**
- u8Rsv3 : **tstrTlsCrlInfo**
- u8Second : **tstrSystemTime**
- u8SecType : **tstrM2MAPConfig** , **tstrM2MConnInfo** , **tstrM2MProvisionInfo** , **tstrM2MWifiSecInfo**
- u8ServiceClass : **tstrM2MWifiRxPacketInfo**
- u8SlotTime : **tstrM2MScanOption**
- u8SsidHide : **tstrM2MAPConfig**
- u8State : **tstrM2mClientState**
- u8Status : **tstrM2MProvisionInfo**
- u8TriggerType : **tstrM2MWPSConnect**
- u8TxPwrLevel : **tstrM2mTxPwrLevel**
- uniAuth : **tstrM2MWifiSecInfo**



- `__PAD16__` : `tstrM2mBatteryVoltage` , `tstrM2mLsnInt` , `tstrM2mPsType` ,
`tstrM2mScanDone` , `tstrM2MScanRegion` , `tstrM2mSetMacAddress` ,
`tstrM2MWifiMonitorModeCtrl` , `tstrM2mWifiStateChanged` , `tstrPrng`
- `__PAD24__` : `tstrCryptoResp` , `tstrM2MAPConfig` , `tstrM2mClientState` ,
`tstrM2MConnInfo` , `tstrM2MDefaultConnResp` , `tstrM2mEnableLogs` ,
`tstrM2MP2PConnect` , `tstrM2MProvisionModeConfig` , `tstrM2mPwrMode` ,
`tstrM2mReqScanResult` , `tstrM2Mservercmd` , `tstrM2mServerInit` ,
`tstrM2mTxPwrLevel` , `tstrM2MWifiRxPacketInfo` , `tstrM2mWifiWepParams` ,
`tstrM2MWPSConnect` , `tstrTlsCrlEntry`
- `__PAD8__` : `tstrEthInitParam` , `tstrM2MMulticastMac` , `tstrOtaUpdateInfo` ,
`tstrSystemTime`
- `__PAD__` : `tstrM2mWifiConnect` , `tstrM2MWifiSecInfo`
- `__RSVD8__` : `tstrM2MScan`
- `_PAD16_` : `tstrOtaUpdateStatusResp`
- `_PAD8_` : `tstrM2mWiifiscanResult`

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by 1.8.13



- a -

- acFileName : **tstrTlsSrvSecFileEntry**
- acHttpServerDomainName : **tstrM2MProvisionModeConfig**
- acPinNumber : **tstrM2MWPSConnect**
- acSSID : **tstrM2MConnInfo**
- astrEntries : **tstrTlsSrvSecHdr**
- astrTlsCrl : **tstrTlsCrlInfo**
- au8BSSID : **tstrM2MWifiMonitorModeCtrl** , **tstrM2MWifiRxPacketInfo** ,
tstrM2mWifiscanResult
- au8Data : **tstrTlsCrlEntry**
- au8DeviceName : **tstrM2MDeviceNameConfig**
- au8DHCPServerIP : **tstrM2MAPConfig**
- au8DstMacAddress : **tstrM2MWifiMonitorModeCtrl** , **tstrM2MWifiRxPacketInfo**
- au8ethRcvBuf : **tstrEthInitParam**
- au8IPAddr : **tstrM2MConnInfo**
- au8Key : **tstrM2MAPConfig**
- au8Mac : **tstrM2mSetMacAddress**
- au8MACAddress : **tstrM2MConnInfo**
- au8macaddress : **tstrM2MMulticastMac**
- au8Passwd : **tstr1xAuthCredentials**
- au8Password : **tstrM2MProvisionInfo**
- au8PSK : **tstrM2MWPSInfo** , **tuniM2MWifiAuth**
- au8SecStartPattern : **tstrTlsSrvSecHdr**
- au8SrcMacAddress : **tstrM2MWifiMonitorModeCtrl** , **tstrM2MWifiRxPacketInfo**
- au8SSID : **tstrM2MAPConfig** , **tstrM2MProvisionInfo** , **tstrM2mWifiConnect** ,
tstrM2mWifiscanResult , **tstrM2MWPSInfo**
- au8UserName : **tstr1xAuthCredentials**
- au8WepKey : **tstrM2MAPConfig** , **tstrM2mWifiWepParams**



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

- p -

- pfAppEthCb : **tstrEthInitParam**
- pfAppMonCb : **tstrWifiInitParam**
- pfAppWifiCb : **tstrEthInitParam** , **tstrWifiInitParam**
- pu8Buffer : **tstrSocketRecvMsg**
- pu8RngBuff : **tstrPrng**

Generated on Thu Jan 26 2017 22:15:21 for WINC1500 IoT Software APIs by doxygen 1.8.13



- S -

- s16BufferSize : **tstrSocketRecvMsg**
- s8Error : **tstrSocketConnectMsg**
- s8ErrorCode : **tstrM2MDefaultConnResp**
- s8Resp : **tstrCryptoResp**
- s8RSSI : **tstrM2MConnInfo** , **tstrM2MWifiRxPacketInfo**
- s8rssi : **tstrM2mWifiscanResult**
- s8RssiThresh : **tstrM2MScanOption**
- s8ScanState : **tstrM2mScanDone**
- s_addr : **in_addr**
- sa_data : **sockaddr**
- sa_family : **sockaddr**
- sin_addr : **sockaddr_in**
- sin_family : **sockaddr_in**
- sin_port : **sockaddr_in**
- sin_zero : **sockaddr_in**
- sock : **tstrSocketAcceptMsg** , **tstrSocketConnectMsg**
- status : **tstrSocketBindMsg** , **tstrSocketListenMsg**
- strAddr : **tstrSocketAcceptMsg**
- strApConfig : **tstrM2MProvisionModeConfig**
- strCred1x : **tuniM2MWifiAuth**
- strEthInitParam : **tstrWifiInitParam**
- strRemoteAddr : **tstrSocketRecvMsg**
- strSec : **tstrM2mWifiConnect**
- strWepInfo : **tuniM2MWifiAuth**



WINC1500 IoT Software APIs

19.5.2

WINC Software API Reference
Manual

- u -

- u16BattVolt : [tstrM2mBatteryVoltage](#)
- u16Ch : [tstrM2mWifiConnect](#)
- u16DataLength : [tstrM2MWifiRxPacketInfo](#)
- u16DataSize : [tstrM2mIpCtrlBuf](#)
- u16ethRcvBufSize : [tstrEthInitParam](#)
- u16FrameLength : [tstrM2MWifiRxPacketInfo](#)
- u16HeaderLength : [tstrM2MWifiTxPacketInfo](#)
- u16LsnInt : [tstrM2mLsnInt](#)
- u16PacketSize : [tstrM2MWifiTxPacketInfo](#)
- u16PassiveScanTime : [tstrM2MScan](#)
- u16PktOffset : [tstrM2mIpRsvdPkt](#)
- u16PktSz : [tstrM2mIpRsvdPkt](#)
- u16PrngSize : [tstrPrng](#)
- u16RemainigDataSize : [tstrM2mIpCtrlBuf](#)
- u16RemainingSize : [tstrSocketRecvMsg](#)
- u16ScanRegion : [tstrM2MScanRegion](#)
- u16Year : [tstrSystemTime](#)
- u32CsBMP : [tstrSslSetActiveCsList](#)
- u32DataRateKbps : [tstrM2MWifiRxPacketInfo](#)
- u32DhcpLeaseTime : [tstrM2MIPConfig](#)
- u32DNS : [tstrM2MIPConfig](#)
- u32FileAddr : [tstrTlsSrvSecFileEntry](#)
- u32FileSize : [tstrTlsSrvSecFileEntry](#)
- u32Gateway : [tstrM2MIPConfig](#)
- u32nEntries : [tstrTlsSrvSecHdr](#)
- u32NextWriteAddr : [tstrTlsSrvSecHdr](#)
- u32OtaControlSecCrc : [tstrOtaControlSec](#)
- u32OtaCortusAppRollbackOffset : [tstrOtaControlSec](#)

- u32OtaCortusAppRollbackValidSts : **tstrOtaControlSec**
- u32OtaCortusAppRollbackVer : **tstrOtaControlSec**
- u32OtaCortusAppWorkingOffset : **tstrOtaControlSec**
- u32OtaCortusAppWorkingValidSts : **tstrOtaControlSec**
- u32OtaCortusAppWorkingVer : **tstrOtaControlSec**
- u32OtaCurrentworkingImgFirmwareVer : **tstrOtaControlSec**
- u32OtaCurrentworkingImgOffset : **tstrOtaControlSec**
- u32OtaFormatVersion : **tstrOtaControlSec**
- u32OtaLastCheckTime : **tstrOtaControlSec**
- u32OtaMagicValue : **tstrOtaControlSec , tstrOtaInitHdr**
- u32OtaPayloadSzie : **tstrOtaInitHdr**
- u32OtaRollbackImageOffset : **tstrOtaControlSec**
- u32OtaRollbackImageValidStatus : **tstrOtaControlSec**
- u32OtaRollbackImgFirmwareVer : **tstrOtaControlSec**
- u32OtaSequenceNumber : **tstrOtaControlSec**
- u32SleepTime : **tstrM2mSlpReqTime**
- u32StaticIP : **tstrM2MIPConfig**
- u32SubnetMask : **tstrM2MIPConfig**
- u8AddRemove : **tstrM2MMulticastMac**
- u8AuthType : **tstrM2mWifiscanResult , tstrM2MWPSInfo**
- u8BcastEn : **tstrM2mPsType**
- u8ch : **tstrM2mWifiscanResult**
- u8Ch : **tstrM2MWPSInfo**
- u8Channel : **tstrM2mServerInit**
- u8ChannelID : **tstrM2MWifiMonitorModeCtrl**
- u8ChNum : **tstrM2MScan**
- u8CipherType : **tstrM2MWifiRxPacketInfo**
- u8cmd : **tstrM2Mservercmd**
- u8CrlType : **tstrTlsCrlInfo**
- u8CurrState : **tstrM2mWifiStateChanged**
- u8DataLen : **tstrTlsCrlEntry**
- u8Day : **tstrSystemTime**
- u8DownloadUrlOffset : **tstrOtaUpdateInfo**
- u8DownloadUrlSize : **tstrOtaUpdateInfo**
- u8Enable : **tstrM2mEnableLogs**
- u8EnableRedirect : **tstrM2MProvisionModeConfig**
- u8EnRecvHdr : **tstrM2MWifiMonitorModeCtrl**
- u8ErrCode : **tstrM2mWifiStateChanged**
- u8EthernetEnable : **tstrEthInitParam**
- u8FrameSubtype : **tstrM2MWifiMonitorModeCtrl , tstrM2MWifiRxPacketInfo**
- u8FrameType : **tstrM2MWifiMonitorModeCtrl , tstrM2MWifiRxPacketInfo**
- u8HeaderLength : **tstrM2MWifiRxPacketInfo**
- u8Hour : **tstrSystemTime**
- u8Index : **tstrM2mReqScanResult**

- u8index : **tstrM2mWifiscanResult**
- u8KeyIndx : **tstrM2MAPConfig , tstrM2mWifiWepParams**
- u8KeySz : **tstrM2MAPConfig , tstrM2mWifiWepParams**
- u8ListenChannel : **tstrM2MAPConfig , tstrM2MP2PConnect**
- u8Minute : **tstrSystemTime**
- u8Month : **tstrSystemTime**
- u8NcdRequiredUpgrade : **tstrOtaUpdateInfo**
- u8NcdUpgradeVersion : **tstrOtaUpdateInfo**
- u8NcfCurrentVersion : **tstrOtaUpdateInfo**
- u8NcfUpgradeVersion : **tstrOtaUpdateInfo**
- u8NoSaveCred : **tstrM2mWifiConnect**
- u8NumofCh : **tstrM2mScanDone**
- u8NumOfSlot : **tstrM2MScanOption**
- u8OtaUpdateStatus : **tstrOtaUpdateStatusResp**
- u8OtaUpdateStatusType : **tstrOtaUpdateStatusResp**
- u8PPAGFor11B : **tstrM2mWifiGainsParams**
- u8PPAGFor11GN : **tstrM2mWifiGainsParams**
- u8Priority : **tstrM2MWifiRxPacketInfo**
- u8ProbesPerSlot : **tstrM2MScanOption**
- u8PsType : **tstrM2mPsType**
- u8PwrMode : **tstrM2mPwrMode**
- u8Rsv1 : **tstrTlsCrlInfo**
- u8Rsv2 : **tstrTlsCrlInfo**
- u8Rsv3 : **tstrTlsCrlInfo**
- u8Second : **tstrSystemTime**
- u8SecType : **tstrM2MAPConfig , tstrM2MConnInfo , tstrM2MProvisionInfo , tstrM2MWifiSecInfo**
- u8ServiceClass : **tstrM2MWifiRxPacketInfo**
- u8SlotTime : **tstrM2MScanOption**
- u8SsidHide : **tstrM2MAPConfig**
- u8State : **tstrM2mClientState**
- u8Status : **tstrM2MProvisionInfo**
- u8TriggerType : **tstrM2MWPSConnect**
- u8TxPwrLevel : **tstrM2mTxPwrLevel**
- uniAuth : **tstrM2MWifiSecInfo**

Table of Contents

License	1
Modules	2
SSL	3
Enumeration/Typedefs	4
Functions	5
WLAN	10
Defines	11
CommonDefines	30
Function	42
m2m_ota_init	46
m2m_ota_notif_set_url	48
m2m_ota_notif_check_for_update	50
m2m_ota_notif_sched	52
m2m_ota_start_update	54
m2m_ota_rollback	57
m2m_ota_abort	59
m2m_ota_switch_firmware	60
m2m_wifi_download_mode	62
m2m_wifi_init	64
m2m_wifi_deinit	66
m2m_wifi_handle_events	68
m2m_wifi_send_crl	70
m2m_wifi_default_connect	72
m2m_wifi_connect	74
m2m_wifi_disconnect	78
m2m_wifi_start_provision_mode	80
m2m_wifi_stop_provision_mode	84
m2m_wifi_get_connection_info	86
m2m_wifi_set_mac_address	89
m2m_wifi_wps	91
m2m_wifi_wps_disable	94
m2m_wifi_p2p	96
m2m_wifi_p2p_disconnect	99

m2m_wifi_enable_ap	101
m2m_wifi_disable_ap	104
m2m_wifi_set_static_ip	106
m2m_wifi_request_dhcp_client	108
m2m_wifi_request_dhcp_server	110
m2m_wifi_enable_dhcp	111
m2m_wifi_set_scan_options	113
m2m_wifi_set_scan_region	114
m2m_wifi_request_scan	116
m2m_wifi_get_num_ap_found	123
m2m_wifi_req_scan_result	126
m2m_wifi_req_curr_rssi	129
m2m_wifi_get_otp_mac_address	131
m2m_wifi_get_mac_address	133
m2m_wifi_set_sleep_mode	135
m2m_wifi_request_sleep	137
m2m_wifi_get_sleep_mode	139
m2m_wifi_req_client_ctrl	140
m2m_wifi_req_server_init	142
m2m_wifi_set_device_name	144
m2m_wifi_set_lsn_int	146
m2m_wifi_enable_monitoring_mode	148
m2m_wifi_disable_monitoring_mode	151
m2m_wifi_send_wlan_pkt	153
m2m_wifi_send_ethernet_pkt	155
m2m_wifi_enable_sntp	157
m2m_wifi_set_sytem_time	159
m2m_wifi_get_sytem_time	161
m2m_wifi_set_cust_InfoElement	163
m2m_wifi_set_power_profile	165
m2m_wifi_set_tx_power	167
m2m_wifi_enable_firmware_logs	169
m2m_wifi_set_battery_voltage	171
m2m_wifi_set_gains	172
m2m_wifi_get_firmware_version	174
m2m_wifi_prng_get_random_bytes	176

DataTypes	178
tstrM2mPwrMode	200
tstrM2mTxPwrLevel	202
tstrM2mEnableLogs	204
tstrM2mBatteryVoltage	206
tstrM2mWifiGainsParams	208
tstrM2mWifiWepParams	210
tstr1xAuthCredentials	212
tuniM2MWifiAuth	214
tstrM2MWifiSecInfo	216
tstrM2mWifiConnect	218
tstrM2MWPSConnect	220
tstrM2MWPSInfo	222
tstrM2MDefaultConnResp	224
tstrM2MScanOption	226
tstrM2MScanRegion	228
tstrM2MScan	230
tstrCryptoResp	232
tstrM2mScanDone	234
tstrM2mReqScanResult	236
tstrM2mWifiscanResult	238
tstrM2mWifiStateChanged	241
tstrM2mPsType	243
tstrM2mSlpReqTime	245
tstrM2mLsnInt	246
tstrM2MWifiMonitorModeCtrl	248
tstrM2MWifiRxPacketInfo	251
tstrM2MWifiTxPacketInfo	255
tstrM2MP2PConnect	257
tstrM2MAPConfig	259
tstrM2mServerInit	262
tstrM2mClientState	264
tstrM2Mservercmd	266
tstrM2mSetMacAddress	268
tstrM2MDeviceNameConfig	270
tstrM2MIPConfig	271

tstrM2mIpRsvdPkt	273
tstrM2MProvisionModeConfig	275
tstrM2MProvisionInfo	277
tstrM2MConnInfo	279
tstrOtaInitHdr	281
tstrOtaControlSec	283
tstrOtaUpdateStatusResp	288
tstrOtaUpdateInfo	290
tstrSystemTime	293
tstrM2MMulticastMac	295
tstrPrng	297
tstrTlsCrlEntry	299
tstrTlsCrlInfo	301
tstrTlsSrvSecFileEntry	303
tstrTlsSrvSecHdr	305
tstrSslSetActiveCsList	307
tstrM2mPwrState	308
BSP	309
Defines	310
DataTypes	312
Function	315
nm_bsp_init	316
nm_bsp_deinit	318
nm_bsp_reset	320
nm_bsp_sleep	322
nm_bsp_register_isr	324
nm_bsp_interrupt_ctrl	326
Socket	328
Defines	329
TCP/IP Defines	330
TLS Defines	334
TLS Socket Options	335
Legacy names for TLS Cipher Suite IDs	337
TLS Cipher Suite IDs	340
Error Codes	346
DataTypes	351

Asynchronous Events	353
tstrSocketBindMsg	357
tstrSocketListenMsg	359
tstrSocketAcceptMsg	361
tstrSocketConnectMsg	363
tstrSocketRecvMsg	365
in_addr	367
sockaddr	369
sockaddr_in	371
Function	373
socketInit	375
registerSocketCallback	378
socket	381
bind	384
listen	387
accept	390
connect	392
recv	395
recvfrom	398
send	401
sendto	403
close	405
nmi_inet_addr	407
gethostbyname	409
sslEnableCertExpirationCheck	411
setsockopt	413
getsockopt	416
m2m_ping_req	418
Data Structures	420
Data Structures	420
in_addr	367
sockaddr	369
sockaddr_in	371
tstr1xAuthCredentials	212
tstrCryptoResp	232

tstrEthInitParam	423
tstrM2MAPConfig	259
tstrM2mBatteryVoltage	206
tstrM2mClientState	264
tstrM2MConnInfo	279
tstrM2MDefaultConnResp	224
tstrM2MDeviceNameConfig	270
tstrM2mEnableLogs	204
tstrM2MIPConfig	271
tstrM2mIpCtrlBuf	426
tstrM2mIpRsvdPkt	273
tstrM2mLsnInt	246
tstrM2MMulticastMac	295
tstrM2MP2PConnect	257
tstrM2MProvisionInfo	277
tstrM2MProvisionModeConfig	275
tstrM2mPsType	243
tstrM2mPwrMode	200
tstrM2mPwrState	308
tstrM2mReqScanResult	236
tstrM2MScan	230
tstrM2mScanDone	234
tstrM2MScanOption	226
tstrM2MScanRegion	228
tstrM2Mservercmd	266
tstrM2mServerInit	262
tstrM2mSetMacAddress	268
tstrM2mSlpReqTime	245
tstrM2mTxPwrLevel	202
tstrM2mWifiConnect	218
tstrM2mWifiGainsParams	208
tstrM2MWifiMonitorModeCtrl	248
tstrM2MWifiRxPacketInfo	251
tstrM2mWifiscanResult	238
tstrM2MWifiSecInfo	216
tstrM2mWifiStateChanged	241

tstrM2MWifiTxPacketInfo	255
tstrM2mWifiWepParams	210
tstrM2MWPSConnect	220
tstrM2MWPSInfo	222
tstrOtaControlSec	283
tstrOtaInitHdr	281
tstrOtaUpdateInfo	290
tstrOtaUpdateStatusResp	288
tstrPrng	297
tstrSocketAcceptMsg	361
tstrSocketBindMsg	357
tstrSocketConnectMsg	363
tstrSocketListenMsg	359
tstrSocketRecvMsg	365
tstrSslSetActiveCsList	307
tstrSystemTime	293
tstrTlsCrlEntry	299
tstrTlsCrlInfo	301
tstrTlsSrvSecFileEntry	303
tstrTlsSrvSecHdr	305
tstrWifiInitParam	428
tuniM2MWifiAuth	214
Data Fields	430
All	430
—	430
a	431
p	433
s	434
u	435
Variables	438
—	438
a	439
p	441
s	442
u	443