WINC1500 MLA
User's Guide

Version 1.0

[This page left blank]

Table of Contents

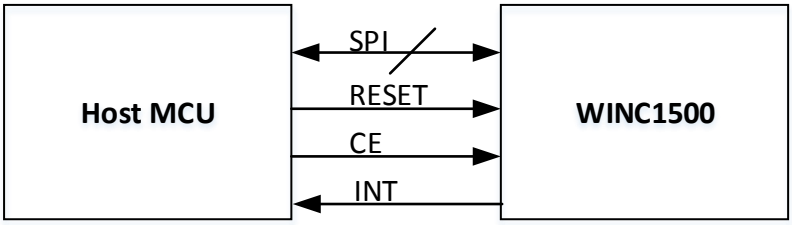
Revision History

Version	Changes	Release Date
1.0	• Initial Release	26 JAN 2017

1 Introduction

The WINC1500 MLA Driver supports the Microchip WINC1500 Wi-Fi Network Controller and allows one to easily create 802.11 wireless applications. The MLA Driver runs on a Host MCU that connects to the WINC1500, as shown below.

Figure 1: Host MCU Connected to WINC1500 Module



Features

- MCU-agnostic; requires only an SPI interface, a timer, 2 GPIO's, and an interrupt line
- Customizable via compiler switches to save memory
- Written in portable 'C' with all source code provided

2 WINC1500 MLA Software Overview

The WINC1500 MLA is a collection of 'C' modules that fall into one of two categories:

- Modules that do not need to be modified (the 'core' driver)
- Modules that contain stub functions requiring custom code

Below is a diagram showing the partitioning of the WINC1500 MLA Driver software on an MCU. Discussions that follow will reference this diagram.

Figure 2: WINC1500 MLA Driver Overview

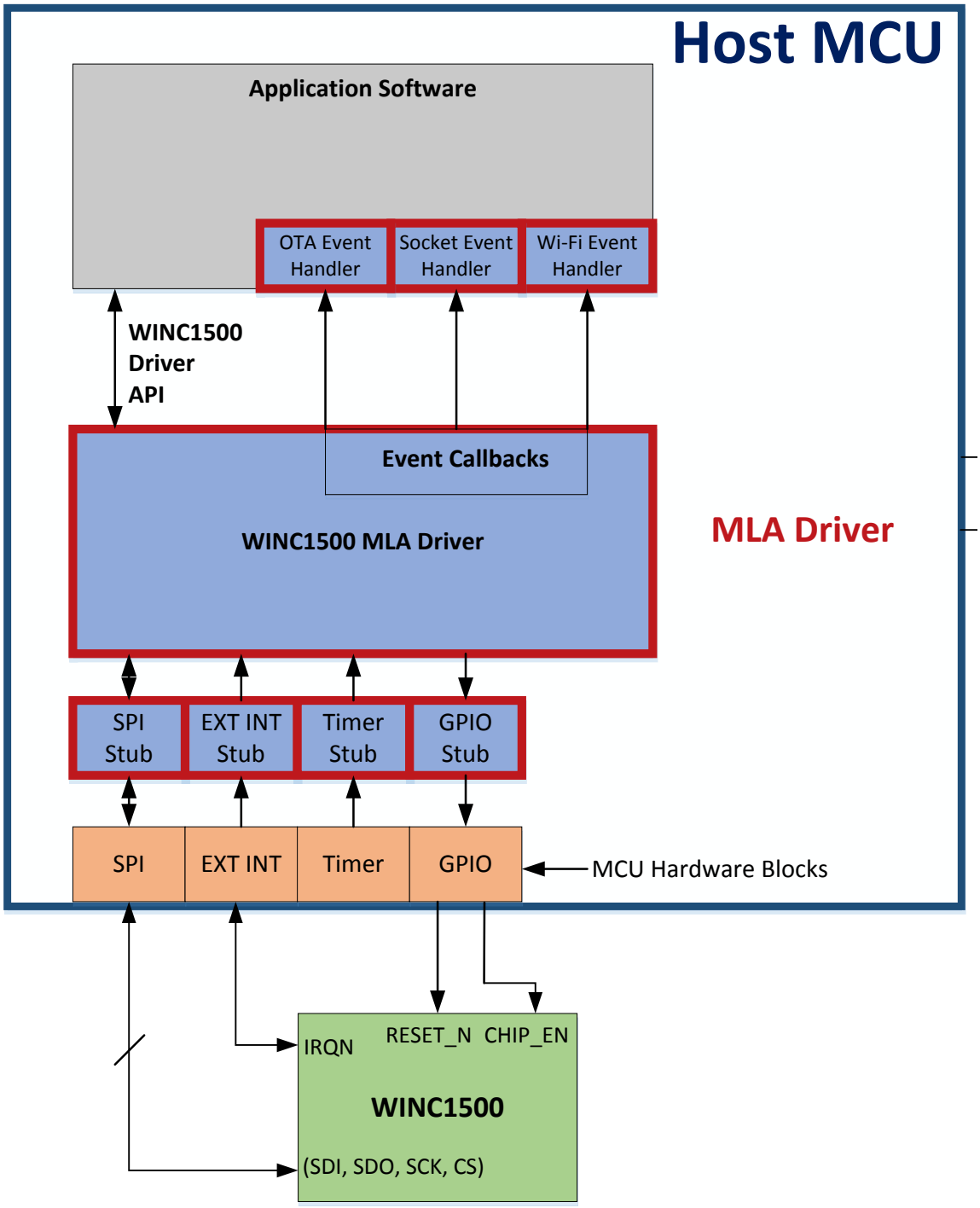


Figure 2 above shows the three major blocks of software that comprise a WINC1500 application. These are:

Application Software This is the application code. Note that three event handlers (OTA, Socket, and Wi-Fi) are part of this block. The event handlers contain callback functions that the driver calls and the application processes.

WINC1500 MLA Driver Software This is the core driver code and is supplied by Microchip. It should not need to be modified.

Stub Software The function prototypes are provided by Microchip. The MLA Driver will call these functions, but they must be coded by the user. The stub functions control MUC-specific hardware and event handling:

- SPI Interface
- GPIO control
- 1ms Timer
- Interrupt from WINC1500
- Wi-Fi, socket, and OTA event handling

3 Source Code

3.1 Data Types

The WINC1500 Driver source code uses data types as defined in stdint.h and stdbool.h (e.g. uint8_t, uint16_t, uint32_t, bool). The winc1500_api.h. file includes these standard header files. If your toolchain does not support stdint.h and stdbool.h then remove these includes and create typedef's for the data types. For example:

```
typedef unsigned char   uint8_t
typedef signed char     int8_t
typedef unsigned short  uint16_t
typedef unsigned long   uint32_t
typedef unsigned char   bool

#define false 0
#define true  1
```

4 Driver Stub API

Before one can use the WINC1500 Driver API there are some MCU-specific stub functions that must be coded. The WINC1500 Driver will call these functions during run-time.

This section discusses the stub functions that are required by the WINC1500 Driver, and must be customized for the Host MCU. There are example demos provided in the release with the stub functions filled in. The MCU driver stub functions are located in wf_mcu_driver.c

4.1 GPIO Stub Functions

The WINC1500 driver needs to control three GPIO outputs to the WINC1500. The GPIO's described in this section should be configured as outputs and defaulted high prior to the WINC1500 driver running.

4.1.1 m2mStub_PinSet_CE

Prototype	`void` **`m2mStub`**`(t_m2mWifiPinAction action);`
Description	The WINC1500 driver will call this function to set the WINC1500 CHIP_EN pin high or low. This is a host output GPIO.
Inputs	action M2M_WIFI_PIN_LOW or M2M_WIFI_PIN_HIGH
Returns	None

4.1.2 m2mStub_PinSet_RESET

Prototype	`void` **`m2mStub_PinSet_RESET`**`(t_m2mWifiPinAction action);`
Description	The WINC1500 driver will call this function to set the WINC1500 RESET_N pin high or low. This is a host output GPIO.
Inputs	action M2M_WIFI_PIN_LOW or M2M_WIFI_PIN_HIGH
Returns	None

4.1.3 m2mStub_PinSet_SPI_SS

Prototype	`void` **`m2mStub_PinSet_SPI_SS`**`(t_m2mWifiPinAction action);`
Description	The WINC1500 driver will call this function to set the WINC1500 SPI_SSN pin high or low. This is a host output GPIO.
Inputs	action M2M_WIFI_PIN_LOW or M2M_WIFI_PIN_HIGH
Returns	None

4.2 Interrupt Stub Functions

The WINC1500 will interrupt the host MCU when events occur by setting the IRQN line low. The Host MCU should be configured to trigger an interrupt on a falling edge.

The Host MCU interrupt handler must call **`m2m_EintHandler()`** each time a WINC1500 interrupt occurs, and then clear the interrupt. Thus, the interrupt handler will look something like:

```
void Winc1500Interrupt(void)
{
    m2m_EintHandler();
    MCU-specific code to clear the interrupt
}
```

The stub functions below allow the WINC1500 driver to enable or disable the WINC1500 interrupt.

4.2.1 m2mStub_EintDisable

Prototype	`void` **`m2mStub_EintDisable`**`(void);`
Description	Disables the host interrupt from the WINC1500.
Inputs	None
Returns	None

4.2.2 m2mStub_EintEnable

Prototype	`void` **`m2mStub_EintEnable`**`(void);`
Description	The WINC1500 driver will call this function to enable the WINC1500 interrupt. When the interrupt is initially configured it should be in a disabled state. The interrupt handler should call `m2m_EintHandler()` and clear the interrupt.
Inputs	None
Returns	None

4.3 Timer Stub Functions

The WINC1500 state machines require a timer with one millisecond resolution. The timer is a 32-bit counter that counts up starting at 0x00000000 and wraps back to 0 after reaching 0xffffffff.

4.3.1 m2mStub_GetOneMsTimer

Prototype	`uint32_t` **`m2mStub_GetOneMsTimer`**`(void)`
Description	Reads the 1ms timer value.
Inputs	None
Returns	Timer value

4.4 SPI Stub Functions

The Host MCU will communicate to the WINC1500 via the SPI interface. SPI supports four modes as shown in Table 4-1. The SPI mode should be set to SPI Mode 0. The SPI clock rate should be 8MHz or less.

Table 4-1: SPI Modes

SPI Mode	CPOL	CPHA	Description	Supported by WINC1500
0	0	0	• Idle state for clock is low • Data captured on clocks rising edge • Data propagated on clocks falling edge	Yes
1	0	1	• Idle state for clock is low • Data captured on clocks falling edge • Data propagated on clocks rising edge	No
2	1	0	• Idle state for clock is high • Data captured on clocks falling edge • Data propagated on clocks rising edge	No
3	1	1	• Idle state for clock is high • Data captured on clocks rising edge • Data propagated on clocks falling edge	No

4.4.1 m2mStub_SpiTxRx

Prototype	`void m2mStub_SpiTxRx(uint8_t *p_txBuf, uint16_t txLen, uint8_t *p_rxBuf, uint16_t rxLen);`
Description	Writes and reads bytes from the WINC1500 across the SPI interface. If txLen > rxLen then throw away the extra read bytes. Do NOT write the garbage read bytes to p_rxBuf. If rxLen is > txLen then write out filler bytes of 0x00 in order to get all the read bytes.
Inputs	p_txBuf — Pointer to tx data (data being clocked out to the WINC1500). This will be NULL if txLen is 0. txLen — Number of bytes to write. This will be 0 if only a read is being done. p_rxBuf — Pointer to rx data (data being clocked in from the WINC1500). This will be NULL if rxLen is 0. rxLen — Number of bytes to read. This will be 0 if only a write is being done.
Returns	None

4.5 Event Stub Functions

These are callback functions that the WINC1500 host driver calls to notify the application of events. Section 7, *Events* , covers all the events in detail.

4.5.1 m2m_wifi_handle_events

Prototype	`void` **`m2m_wifi_handle_events`**`(t_m2mWifiEventType eventCode, t_wifiEventData *p_eventData);`
Description	Called to notify the application of Wi-Fi events. See *Wi-Fi Events* in Section 7.1.
Inputs	eventCode type of event (see `t_m2mWifiEventType` in source code) p_eventData pointer to the `t_wifiEventData` union
Returns	None

4.5.2 m2m_socket_handle_events

Prototype	`void` **`m2m_socket_handle_events`**`(SOCKET sock, t_m2mSocketEventType eventCode, t_socketEventData *p_eventData);`
Description	Called to notify the application of Socket events. See *Socket Events* in Section 7.2.
Inputs	sock socket ID event is associated with eventCode type of event (see `t_m2mSocketEventType` in source code) p_eventData pointer to the `t_socketEventData` union
Returns	None

4.5.3 m2m_ota_handle_events

Prototype	`void` **`m2m_ota_handle_events`**`(t_m2mOtaEventType eventCode, t_m2mOtaEventData *p_eventData);`
Description	Called to notify the application of OTA events. See *OTA Events* in Section 7.3.
Inputs	eventCode type of event (see `t_wfOtaEvent` in source code) p_eventData pointer to the `t_otaEventData` union
Returns	None

4.5.4 m2m_error_handle_events

Prototype	`void` **`m2m_error_handle_events`**`(uint32_t errorCode);`
Description	Called to notify the application of error events. See *OTA Events* in Section 7.3. Error codes are described in wf_errors.h (see `t_m2mWifiErrorCodes`).
Inputs	errorCode Error code
Returns	None

5 Driver Customization

The WINC1500 Driver has several features that can be customized, and are described in this section. The goal of most the defines is to remove features not needed by an application in order to optimize memory usage. All of the #defines discussed in this section can be found in winc1500_driver_config.h.

#define	Description
M2M_POINTER_SIZE_IN_BYTES	Size of a 'C' pointer on the host MCU. Must be either 2 or 4 bytes.
M2M_ENABLE_ERROR_CHECKING	Comment out to remove parameter error checking
M2M_ENABLE_PRNG	Comment out to remove pseudo-random number functions
M2M_ENABLE_SOFT_AP_MODE	Comment out if not using Soft AP mode
M2M_ENABLE_WPS	Comment out if not using WPS
M2M_ENABLE_P2P	Comment out if not using P2P
M2M_ENABLE_HTTP_PROVISION_MODE	Comment out if not using HTTP provision mode
M2M_ENABLE_SCAN_MODE	Comment out if not using Wi-Fi scanning
M2M_ENABLE_SPI_FLASH	Comment out if not using PC WINC1500 firmware download utility via PC
M2M_DISABLE_FIRMWARE_LOG	Comment out if using WINC1500 firmware logging. See description in source code.

6 WINC1500 Driver API

This section covers all MLA Driver functions that the application can call.

6.1 Initialization Functions

The functions in this section are used to initialize the WINC1500 Driver, reset the WINC1500, and prepare for connectivity. The function prototypes are located in winc1500_api.h.

6.1.1 m2m_wifi_init

Prototype	`void` **`m2m_wifi_init`**`(void);`
Description	This function must be called before any other functions in the API; it starts the internal initialization state machine to initialize the driver and the WINC1500 for operations. No further API calls should be made (other than `m2m_wifi_task`) until the M2M_WIFI_DRIVER_INIT_EVENT has been generated. See *Wi-Fi Events*.
Inputs	None
Returns	None

6.1.2 m2m_wifi_task

Prototype	`void` **`m2m_wifi_task`**`(void);`
Description	After calling `m2m_wifi_init()` this function must be called periodically for the WINC1500 Driver to function.
Inputs	None
Returns	None

6.2 Wi-Fi API

The functions in this section describe how to create and monitor Wi-Fi connections. The function prototypes are located in winc1500_api.h.

6.2.1 m2m_wifi_connect

Prototype	`void m2m_wifi_connect(char *pcSsid, uint8_t u8SsidLen, uint8_t u8SecType, void *pvAuthInfo, uint16_t u16Ch);`
Description	Initiates a Wi-Fi connection using the input parameters. Upon the connection succeeding (or failing), the `M2M_WIFI_CONN_STATE_CHANGED_EVENT` is generated.

Inputs	pcSssid — SSID of AP to connect to (a null-terminated string)
	u8SsidLen — Length of SSID, in bytes (not including Null terminator)
	u8SecType — Must be one of the following:
	`M2M_WIFI_SEC_OPEN`
	`M2M_WIFI_SEC_WPA_PSK`
	`M2M_WIFI_SEC_WEP`
	`M2M_WIFI_SEC_802_1X`
	pvAuthInfo — Pointer to one of the following (see `t_m2mWifiAuth` in source):
	• security passphrase string if connecting with WPA
	• binary WPA PSK array if connecting with WPS
	• `t_m2mWifiWepSecurity` structure if connecting with WEP
	u16Ch — Wi-Fi channel

Returns	None

6.2.2 m2m_wifi_connect_sc

Prototype	`void` **`m2m_wifi_connect_sc`**`(char *pcSsid, uint8_t u8SsidLen, uint8_t u8SecType, void *pvAuthInfo, uint16_t u16Ch);`
Description	Initiates a Wi-Fi connection and saves connection config in FLASH. Identical to `m2m_wifi_connect()` except input connection parameters are saved to WINC1500 FLASH. A future call to `m2m_default_connect`() will use these parameters. Upon the connection succeeding (or failing), the `M2M_WIFI_CONN_STATE_CHANGED_EVENT` is generated.

Inputs	pcSsid — SSID of AP to connect to (a null-terminated string)
	u8SsidLen — Length of SSID, in bytes (not including Null terminator)
	u8SecType — Must be one of the following:
	`M2M_WIFI_SEC_OPEN`
	`M2M_WIFI_SEC_WPA_PSK`
	`M2M_WIFI_SEC_WEP`
	`M2M_WIFI_SEC_802_1X`
	pvAuthInfo — Pointer to one of the following (see `t_m2mWifiAuth` in source):
	• security passphrase string if connecting with WPA
	• binary WPA PSK array if connecting with WPS
	• `t_m2mWifiWepSecurity` structure if connecting with WEP
	u16Ch — Wi-Fi channel

Returns	None

6.2.3 m2m_default_connect

Prototype	`void` **`m2m_default_connect`**`(void)`
Description	Connects using the most recently saved connection profile from the previous call to `m2m_wifi_connect_sc()`. After this call the `M2M_WIFI_DEFAULT_CONNNECT_EVENT` is generated. See *Wi-Fi Events*.
Inputs	None
Returns	None

6.2.4 m2m_wifi_get_connection_info

Prototype	`void` **`m2m_wifi_get_connection_info`**`(void);`
Description	Requests the current connection information. The `M2M_WIFI_CONN_INFO_RESPONSE_EVENT` is generated when the information is ready. See *Wi-Fi Events*.
Inputs	None
Returns	None

6.2.5 m2m_wifi_disconnect

Prototype	`void` **`m2m_wifi_disconnect`**`(void);`
Description	Disconnects from the currently connected AP. If not connected this function has no effect. Only valid in station mode; should not be called when in SoftAP mode. When the disconnect is complete the `M2M_WIFI_CONN_STATE_CHANGED_EVENT` is generated. See *Wi-Fi Events*.
Inputs	None
Returns	None

6.2.6 m2m_wifi_wps

Prototype	`void` **`m2m_wifi_wps`**`(uint8_t u8TriggerType, const char *pcPinNumber);`
Description	Connects to an AP using WPS (Wi-Fi Protected Setup). If connecting to an AP using WPS then this function must be used instead of `m2m_wifi_connect()` or `m2m_wifi_connect_sc()`. The WINC1500 must be idle or in STA mode prior to calling this function. Upon success (or failure) the `M2M_WIFI_WPS_EVENT` is generated. See *Wi-Fi Events*.
Inputs	u8TriggerType `WPS_PIN_TRIGGER` for pin method or `WPS_PBC_TRIGGER` for push-button method. pcPinNumber Pointer to pin number string; only used if wpsMode is `WPS_PIN_TRIGGER`. Must be an ASCII decimal null-terminated string of 7 digits (e.g. "1234567").
Returns	None

6.2.7 m2m_wifi_wps_disable

Prototype	`void` **`m2m_wifi_wps_disable`**`(void);`
Description	Disables WPS mode if `m2m_wifi_wps()` was called previously.
Inputs	None
Returns	None

6.2.8 m2m_wifi_p2p

Prototype	`void` **`m2m_wifi_p2p`**`(uint8_t u8channel);`
Description	Enables Wi-Fi Direct mode (also known as P2P). The WINC supports P2P in device listening mode only (intent of 0). The WINC P2P implementation does not support P2P GO (Group Owner) mode. Active P2P devices (e.g. phones) can find the WINC1500 in the search list. When a device connects to the WINC1500 the `M2M_WIFI_CONN_STATE_CHANGED_EVENT` is generated. Shortly after, the `M2M_WIFI_IP_ADDRESS_ASSIGNED_EVENT` is generated. See *Wi-Fi Events*.
Inputs	u8channel P2P listen channel. It must be either 1, 6, or 11
Returns	None

6.2.9 m2m_wifi_p2p_disconnect

Prototype	`void` **`m2m_wifi_p2p_disconnect`**`(void);`
Description	Removes the WINC1500 from P2P mode. This should only be called if `m2m_wifi_p2p()` was called previously.
Inputs	None
Returns	None

6.2.10 m2m_wifi_set_device_name

Prototype	`void` **`m2m_wifi_set_device_name`**`(char *pu8DeviceName, uint8_t u8DeviceNameLength);`
Description	The device name is used in (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 bytes of the OTP MAC address. If OTP (eFuse) is not programmed then zeros will be used (e.g. "WINC-00-00"). See RFC 952 and 1123 for valid device name rules. **Note**: This function should only be called once after initialization.
Inputs	Pu8DeviceName — Pointer to null-terminated device name string. Max size is 48 characters including the string terminator. u8DeviceNameLength — Length of device name, in bytes. Does not include Null terminator.
Returns	None

6.2.11 m2m_wifi_enable_ap

Prototype	`void` **`m2m_wifi_enable_ap`**`(const tstrM2MAPConfig *pstrM2MAPConfig);`
Description	The WINC1500 starts a SoftAP network. Only a single client is supported. Once a client connects to the WINC1500 other clients attempting to connect will be rejected. The `M2M_WIFI_IP_ADDRESS_ASSIGNED_EVENT` is generated when a client connects. See *Wi-Fi Events*. **Note**: Power save features are not supported in Soft AP mode.
Inputs	pstrM2MAPConfig Pointer to Soft AP configuration. See *tstrM2MAPConfig* below.
Returns	None

6.2.11.1 tstrM2MAPConfig

```
typedef struct
{
    uint8_t     au8SSID [M2M_MAX_SSID_LEN];
    uint8_t     u8ListenChannel;
    uint8_t     u8KeyIndx;
    uint8_t     u8KeySz;
    uint8_t     au8WepKey [WEP_104_KEY_STRING_SIZE + 1];
    uint8_t     u8SecType;
    uint8_t     u8SsidHide;
    uint8_t     au8DHCPServerIP[4];
    uint8_t     au8Key[M2M_WIFI_MAX_PSK_LEN];
    uint8_t     padding[2];
} tstrM2MAPConfig;
```

Field	Description
au8SSID	SSID name of Soft AP network
u8ListenChannel	Channel to use for Soft AP
u8KeyIndx	WEP key index (only used if `securityType` is set to `M2M_WIFI_SEC_WEP`)
u8KeySz	If using WEP, then will be either `WEP_40_KEY_STRING_SIZE` or `WEP_104_KEY_STRING_SIZE`. If using `M2M_WIFI_SEC_WPA_PSK` then will be the size of wpaKey (not including string terminator). If security is open this field won't be used.
au8WepKey	WEP key string with string terminator (only used if security is WEP)
u8SecType	`M2M_WIFI_SEC_OPEN`, `M2M_WIFI_SEC_WEP`, or `M2M_WIFI_SEC_WPA_PSK`
u8SsidHide	`M2M_WIFI_SSID_MODE_VISIBLE` or `M2M_WIFI_SSID_MODE_HIDDEN`
au8DHCPServerIP	Array of 4 bytes with with the IP address for the server. For example, if the desired IP address is 192.168.1.2 then the array values must be: [0] = 192, [1] = 168, [2] = 1, [3] = 2
au8Key	WPA pass-phrase with string terminator (only valid if `securityType` is `M2M_WIFI_SEC_WPA_PSK`.
padding	Not used

6.2.12 m2m_wifi_disable_ap

Prototype	`void` **`m2m_wifi_disable_ap`**`(void);`
Description	Disables Soft AP mode. See `m2m_wifi_enable_ap()`.
Inputs	None
Returns	None

6.2.13 m2m_wifi_set_cust_InfoElement

Prototype	`void` **`m2m_wifi_set_cust_InfoElement`**`(uint8_t* pau8M2mCustInfoElement)`
Description	This function is only applicable in SoftAP mode. It allows adding a custom information element to the beacon or probe response. The function can be called more than once if an IE needs to be appended to previously defined IE's.

Inputs	pau8M2mCustInfoElement Pointer to array. The general format of the array is:
	`[0] -- total number of bytes in the array`
	`[1] -- Element ID for IE #1`
	`[2] -- Length of data in IE #1`
	`[3:N] -- Data for IE #1`
	`[N+1] -- Element ID for IE #2`
	The maximum size of the array cannot exceed 255 bytes.

Example	**Example 1**: Presume the application needs to add two information elements, the first element has an ID of 160, length 3, data of 'A', 'B', 'C'. The second element has an ID of 37, length 4, data of 'H', '2', '0', '0'. The array would be set up as:
	`[0] = 11 // 11 total bytes in array`
	`[1] = 160 // ID #1`
	`[2] = 3 // length of data for ID #1`
	`[3] = 'A' // data[0] for ID #1`
	`[4] = 'B' // data[1] for ID #1`
	`[5] = 'C' // data[2] for ID #1`
	`[6] = 37 // ID #2`
	`[7] = 3 // length of data for ID #2`
	`[7] = 'H' // data[0] for ID #2`
	`[8] = '2' // data[1] for ID #2`
	`[9] = '0' // data[2] for ID #2`
	`[10] = '0' // data[3] for ID #2`
	
	Example 2: Presume the IE's in Example 1 have been created. The application wants to append a new IE with ID of 50, length 2, data of '2', 'X':
	`[0] = 15 // cumulative total, 11 + 4`
	`[1:10] // Same values as from Example 1`
	`[11] = 50 // ID #3`
	`[12] = 2 // length of ID #3`
	`[13] = '2' // data[0] for ID #3`
	`[14] = 'X' // data[1] for ID #3`
	
	Example 3: To clear all information elements, the array has a single zero byte:
	`[0] = 0`

Returns	None

6.2.14 m2m_wifi_request_scan

Prototype	`void` **`m2m_wifi_request_scan`**`(uint8_t ch);`
Description	Requests an active Wi-Fi scan operation (the WINC1500 sends out probe requests). When the scan operation completes the `M2M_WIFI_SCAN_DONE_EVENT` is generated (see *Wi-Fi Events*). After this event the `m2m_wifi_req_scan_result()` is called as many times as needed to retrieve the scan results. A scan can be requested in STA mode whether connected or disconnected. A scan cannot be requested when in P2P or SoftAP modes.
Inputs	ch Desired channel, or all channels. See `t_m2mWifiScanChannels`.
Returns	None

6.2.15 m2m_wifi_request_scan_passive

Prototype	`void` **`m2m_wifi_request_scan_passive`**`(uint8_t ch, uint16_t scan_time);`
Description	This function requests a passive Wi-Fi scan operation. The WINC1500 does not send out probe requests, but passively listens. When the scan operation completes the `M2M_WIFI_SCAN_DONE_EVENT` is generated; after this event the `m2m_wifi_req_scan_result()` as many times as needed to retrieve the scan results. See *Wi-Fi Events*. A scan can be requested in STA mode whether connected or disconnected. A scan cannot be requested when in P2P or SoftAP modes.
Inputs	ch Desired channel, or all channels. See `t_m2mWifiScanChannels`. scan_time The time (in milliseconds) that passive scan is listening to beacons on each channel per one slot. A value of 0 uses the default settings.
Returns	None

6.2.16 m2m_wifi_req_scan_result

Prototype	`void` **`m2m_wifi_req_scan_result`**`(uint8_t index);`
Description	After a scan has been initiated the `M2M_WIFI_SCAN_DONE_EVENT` is generated when the scan is complete; the number of scan results are reported with this event. At that point this function is called to retrieve each scan result. When the scan result has been retrieved the `M2M_WIFI_SCAN_RESULT_EVENT` is generated. See *Wi-Fi Events*. This function must not be called prior to calling either `m2m_wifi_request_scan()`, `m2m_wifi_request_scan()`, or `m2m_wifi_req_hidden_ssid_scan()` with the result of at least one AP being found.
Inputs	index Index of scan result to retrieve
Returns	None

6.2.17 m2m_wifi_req_hidden_ssid_scan

Prototype	`void` **`m2m_wifi_req_hidden_ssid_scan`**`(uint8_t ch, uint8_t *p_ssidList);`
Description	For security reasons an AP can be configured to not broadcast its SSID. This function allows an application to specifically scan for one or more AP's with a hidden SSID. When the scan completes the `M2M_WIFI_SCAN_DONE_EVENT` is generated. See *Wi-Fi Events*.
Inputs	ch — Desired channel, or all channels. See `t_m2mWifiScanChannels` p_ssidList — Pointer to an array containing the list SSID's to look for. The format of the list is: `list[0]` Total number of SSID's in the list `list[1]` Length of first SSID `list[2]` First character of first SSID `list[2:N]` Remaining characters of first ID (do **not** include a '\0' terminator) `list[N+1]` Length of second SSID `list[N+2]` First character of second SSID … Restrictions to the list: • The maximum number of SSID's that can be in the list is `M2M_WIFI_MAX_HIDDEN_SITES` (4). • The maximum size of the buffer cannot exceed 133 bytes. • The maximum size of an SSID cannot exceed 32 bytes. • String terminators are NOT included in the list

Example	Presume a list of two SSID's, "AP_1", and "AP_20". To create the list:
	`uint8_t ssidList[12];`
	
	`ssidList[0] = 2;` `// Total number of SSID's in list is 2`
	`ssidList[1] = strlen("AP_1");` `// Length of "AP_1" in list is 4`
	`//  (do not count string terminator)`
	`// Bytes index 2-5 containing the string AP_1`
	`memcpy(&ssidList[2], "AP_1", strlen("AP_1"));`
	
	`// Length of "AP_20" in list is 5`
	`// (do not count string terminator)`
	`ssidList[6] = strlen("AP_20");`
	
	`// Bytes index 7-11 containing the string "AP_20"`
	`memcpy(&ssidList[7], "AP_20", strlen("AP_20"));`
	
	ssidList[] now looks like:

Returns	None

Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	2	4	'A'	'P'	'_'	'1'	5	'A'	'P'	'_'	'2'	'0'

6.2.18 m2m_wifi_set_scan_options

Prototype	`void` **`m2m_wifi_set_scan_options`**`(tstrM2MScanOption *ptstrM2MScanOption);`
Description	Sets the options for a Wi-Fi scan. Normally this function will not be needed as the default scan options are sufficient.
Inputs	`ptstrM2MScanOption` pointer to scan options (*tstrM2MScanOption* below).
Returns	None

6.2.18.1 tstrM2MScanOption

```
typedef struct

{
    uint8_t   u8NumOfSlot;
    uint8_t   u8SlotTime;
    uint8_t  u8ProbesPerSlot;
    int8_t   s8RssiThresh;
} tstrM2MScanOption;
```

Field	Description
`u8NumOfSlot`	The min number of slots is 2 for every channel. Every slot the WINC will send a probe request and then wait/listen for a probe response/beacon slotTime
`u8SlotTime`	The time (in ms) that the WINC will wait on every channel listening for AP's. The min time is 10ms; the max time is 250ms.
`u8ProbesPerSlot`	Number of probe requests to be sent per channel scan slot
`s8RssiThresh`	RSSI threshold of the AP

6.2.19 m2m_wifi_set_scan_region

Prototype	`void` **`m2m_wifi_set_scan_region`**`(uint16_t scanRegion);`
Description	Setting the scan region affects the range of channels that can be scanned. The default scan region is M2M_WIFI_NORTH_AMERICA_REGION.
Inputs	scanRegion M2M_WIFI_NORTH_AMERICA_REGION, M2M_WIFI_EUROPE_REGION or M2M_WIFI_NORTH_ASIA_REGION
Returns	None

6.2.20 m2m_wifi_set_static_ip

Prototype	`void` **`m2m_wifi_set_static+ip`**`(tstrM2MIPConfig * pstrStaticIPConf);`
Description	Assign a static IP address (and related parameters) to the WINC1500. This function is needed if the AP does not have a DHCP server, or, a known, static IP address is needed. Assigning a static IP address must be done with care to avoid a network conflict. If the WINC1500 detects a conflict the `M2M_WIFI_IP_CONFLICT_EVENT` will be generated. See *Wi-Fi Events*.
Inputs	pstrStaticIPConf Pointer to static IP configuration. See *tstrM2MIPConfig* below.
Returns	None

6.2.20.1 tstrM2MIPConfig

```
typedef struct
{
    uint32_t u32StaticIP;           // static IP address (little-endian)
    uint32_t u32Gateway;            // default gateway IP address (little-endian)
    uint32_t u32DNS;                // DNS server IP address (little-endian)
    uint32_t u32SubnetMask;         // subnet mask (little-endian)
    uint32_t u32DhcpLeaseTime;      // not used
} tstrM2MIPConfig;
```

6.2.21 m2m_wifi_get_mac_address

Prototype	`void` **`m2m_wifi_get_mac_address`**`(uint8_t * pu8MacAddr);`
Description	Reads the current MAC address on the WINC1500.
Inputs	pu8MacAddr Pointer to where MAC address is written
Returns	None

6.2.22 m2m_wifi_get_otp_mac_address

Prototype	`void` **`m2m_wifi_get_otp_mac_address`**`(uint8_t * pu8MacAddr, uint8_t pu8isValid);`
Description	Reads the hardware (OTP) MAC address on the WINC1500. If the OTP has been programmed with a valid MAC address this function returns that address. If the OTP has not been programmed with a MAC address this functions returns a MAC address of all zeros.
Inputs	pu8MacAddr Pointer to where MAC address is written pu8uisValid True if OTP MAC address has been programmed, else False
Returns	True if OTP MAC address has been programmed, else False

6.2.23 m2m_wifi_set_mac_address

Prototype	`bool` **`m2m_wifi_set_mac_address`**`(uint8_t *au8MacAddress);`
Description	Performs a software overrides of the WINC1500 MAC address. Typically only needed when testing. In production code the WINC1500 MAC address should be used.
Inputs	au8MacAddress Pointer to MAC address
Returns	None

6.2.24 m2m_wifi_req_curr_rssi

Prototype	`void` **`m2m_wifi_req_curr_rssi`**`(void);`
Description	Requests the RSSI value for the current connection. After this function is called the `M2M_WIFI_RSSI_EVENT` is generated. See the *Wi-Fi Events* section.
Inputs	None
Returns	None

6.2.25 m2m_wifi_set_sleep_mode

Prototype	`void` **`m2m_wifi_set_sleep_mode`**`(uint8_t PsTyp, uint8_t BcastEn);`
Description	Sets the power-save mode for WINC1500. This is one of the two power-save setting functions that allow an application to adjust WINC1500 power consumption (the other function is `m2m_wifi_set_lsn_int()`). Most of the power-save modes are performed automatically by the WINC1500. If `PsTyp` is set to `M2M_WIFI_PS_MANUAL` then `m2m_wifi_request_sleep()` is used to control the sleep times from the application. **Note**: This function should only be called once after initialization.
Inputs	PsTyp Range:

PsTyp	
`M2M_WIFI_PS_DISABLED`	Power save is disabled.
`M2M_WIFI_PS_AUTOMATIC`	Power save is done automatically by the WINC1500. This mode doesn't disable all of the WINC1500 and use higher levels of power than the `M2M_WIFI_PS_H_AUTOMATIC` and the `M2M_WIFI_PS_DEEP_AUTOMATIC` modes.
`M2M_WIFI_PS_H_AUTOMATIC`	Power save is done automatically by the WINC1500. Achieves higher power save than the `M2M_WIFI_PS_AUTOMATIC` mode by shutting down more parts of the WINC device.
`M2M_WIFI_PS_DEEP_AUTOMATIC`	Power save is done automatically by the WINC1500. Achieves the highest possible power save. This is equivalent to 802.11
`M2M_WIFI_PS_MANUAL`	Power save is done manually by host application. WINC1500 is not synchronized to beacons and could be sleeping when data is sent to it, thus losing frames.

BcastEn Range:

BcastEn	
0	Power save is disabled. The WINC1500 will not wake up at the DTIM interval and will not receive broadcast traffic. It will still wake up at the listen interval; see `m2m_wifi_set_lsn_int()`.
1	The WINC1500 will wake up at each DTIM interval to listen for broadcast traffic.

Returns	None

6.2.26 m2m_wifi_set_lsn_int

Prototype	`void` **`m2m_wifi_set_lsn_int`** `tstrM2mLsnInt pstrM2mLsnInt);`
Description	Sets the Wi-Fi listen interval in AP beacon intervals. This is one of the two power-save setting functions that allow an application to adjust WINC1500 power consumption (the other function is `m2m_wifi_set_sleep_mode()`). Typically a beacon interval on an AP is 100ms, but it can vary. The listen interval is the number of beacon periods the station sleeps before waking up to receive data the AP has buffered for it. **Note**: This function should only be called once after initialization.

Inputs	`pstrM2mLsnInt` Pointer to structure:
	`typedef struct`
	`{`
	`  uint16_t u16LsnInt; // Sleep interval, in beacon periods`
	`  uint8_t  padding;`
	`}tstrM2mLsnInt;`

Returns	None

6.2.27 m2m_wifi_request_sleep

Prototype	`void` **`m2m_wifi_request_sleep`**`(uint32_t u32SlpReqTime);`
Description	Puts the WINC1500 into the current sleep mode. This function should only be called if the previous call to `m2m_wifi_set_sleep_mode()` set the PsTyp to `M2M_WIFI_PS_MANUAL`. Essentially, this function is used for those applications that wish to control the WINC1500 sleep times manually as opposed to the other modes where the WINC1500 controls the sleep times automatically. **Note**: The host driver will automatically wake up the WINC1500 when any host driver API function (e.g. Wi-Fi or socket) is called which requires communication with the WINC1500.
Inputs	u32SlpReqTime Number of milliseconds that the sleep mode should be active.
Returns	None

6.2.28 m2m_wifi_get_sleep_mode

Prototype	`void` **`m2m_wifi_get_sleep_mode`**`(void);`
Description	Returns the current sleep mode. The possible sleep modes are described in `m2m_wifi_set_sleep_mode()`.
Inputs	None
Returns	None

6.2.29 m2m_wifi_set_tx_power

Prototype	`void` **`m2m_wifi_set_tx_power`**`(uint8_t u8TxPwrLevel);`
Description	Sets the Tx power level. If this function is to be used, then it must be called after initialization and before any connection request. This function can only be called once after initialization.
Inputs	u8TxPwrLevel Range: `M2M_WIFI_TX_PWR_HIGH` – PPA Gain 6dbm, PA Gain 18dbm `M2M_WIFI_TX_PWR_MED` – PPA Gain 6dbm, PA Gain 12dbm `M2M_WIFI_TX_PWR_LOW` – PPA Gain 6dbm, PA Gain 6dbm
Returns	None

6.2.30 m2m_wifi_set_power_profile

Prototype	`void` **`m2m_wifi_set_power_profile`**`(uint8_t u8PwrMode);`
Description	Sets the WINC1500 power profile. If this function is to be used, then it must be called after initialization and before any connection request. This function can only be called once after initialization.
Inputs	u8PwrMode Range: `M2M_WIFI_PWR_AUTO` – WINC1500 firmware will decide the best power mode to use internally `M2M_WIFI_PWR_LOW1` – Low power mode 1 `M2M_WIFI_PWR_LOW2` – Low power mode 2 `M2M_WIFI_PWR_HIGH` – High power mode
Returns	None

6.3 Socket API

6.3.1 socket

Prototype	`SOCKET socket(uint16_t domain, uint8_t type, uint8_t flags);`
Description	Creates a socket.
Inputs	domain — Always `AF_INET` type — Socket type (`SOCK_DGRAM` or `SOCK_STREAM`) flags — Used to modify socket creation. It should be set to 0 for normal TCP/UDP sockets. If creating an SSL session set this parameter to `SOCKET_FLAGS_SSL` (only allowed if *type* is `SOCK_STREAM`).
Returns	Socket ID. If this value is negative the function failed (see t_socketError in source code).

6.3.2 bind

Prototype	`int8_t bind(SOCKET sock, struct sockaddr *my_addr, uint8_t addrlen);`
Description	Associates the provided address and local port to a socket. The function must be used with both TCP and UDP sockets before starting any UDP or TCP server operation. Upon socket bind completion the application will receive a `M2M_SOCKET_BIND_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock — Socket ID returned from socket() my_addr — Pointer to sockaddr_in structure. See *struct sockaddr_in* in Section 6.3.16 addrlen — Size,in bytes, of sockaddr_in structure
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.3 listen

Prototype	`int8_t listen(SOCKET sock, uint8_t backlog);`
Description	Waits for an incoming connection. Only applies to a TCP socket. After the bind() call, this function can be called to listen (wait) for an incoming connection. Upon the listen succeeding, the application will receive a `M2M_SOCKET_LISTEN_EVENT` in the callback `m2m_socket_handle_events()`. If successful, the TCP server operation is active and is ready to accept connections.
Inputs	sock — Socket ID returned from socket() backlog — Not used, set to 0
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.4 accept

Prototype	`int8_t` **`accept`**`(SOCKET sock, struct sockaddr *addr, uint8_t *addrlen);`
Description	Place-holder for BSD accept function. The function, in the WINC1500 driver, does not perform any work. It is present in the driver to support legacy code, or, to write code that adheres to the 'normal' BSD socket interface. Normally, accept() is called after listen(). It is not required to call this function. When a client connects to the WINC1500, the application will receive a `M2M_SOCKET_ACCEPT_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock Socket ID returned from socket() addr Not used addrlen Not used
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.5 connect

Prototype	`int8_t` **`connect`**`(SOCKET sock, struct sockaddr *server_addr, uint8_t addrlen);`
Description	Connects to a remote server. Only applies to a TCP socket. Once a socket has been created this function can be called to connect to a remote server. If bind was not called previously, the socket is bound to the WINC1500 IP address and a random local port number. This is the typical usage. If bind is called prior to connect, then the client socket will use the designated IP address and port number. When the connection completes the application will receive a `M2M_SOCKET_CONNECT_EVENT` in the callback `m2m_socket_handle_events)`; at that point the TCP session is active and data can be sent and received.
Inputs	sock Socket ID returned from socket(). server_addr IP address and port number of server. Fill in '`struct sockaddr_in`' and cast it to '`struct sockaddr`'. See `struct sockaddr_in` in Section 6.3.16. addrlen Size, in bytes, of `struct sockaddr`
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.6 recv

Prototype	`int8_t` **`recv`**`(SOCKET sock, void *buf, uint16_t len, uint32_t timeout);`
Description	Receives data from a TCP socket. Once a TCP socket has connected, this function can be called to await incoming data from the remote server. When data is received the application will receive a `M2M_SOCKET_RECV_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock Socket ID returned from socket() buf Pointer to application buffer that the received data will be written to len Size of *buf*, in bytes timeout Time, in milliseconds, to wait for receive data. If the timeout parameter is set to 0 then the socket will wait forever for data to be received. If a timeout occurs, the `M2M_SOCKET_RECV_EVENT` is still generated, but, the event data field 'bufSize' (see `t_socketRecv`) will be a negative value equal to `SOCK_ERR_TIMEOUT`.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.7 recvfrom

Prototype	`int8_t recvfrom(SOCKET sock, void *buf, uint16_t len, uint32_t timeout);`
Description	Once a UDP socket has been created and bound (see bind()) this function can be called to await incoming data. When data is received the the application will receive a `M2M_SOCKET_RECVFROM_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock — Socket ID returned from socket() buf — Pointer to application buffer that the received data will be written to len — Size of *buf*, in bytes timeout — Time, in milliseconds, to wait for receive data. If the timeout parameter is set to 0 then the socket will wait forever for data to be received. If a timeout occurs, the `M2M_SOCKET_RECVFROM_EVENT` is still generated, but, the event data field 'bufSize' (see `t_socketRecv`) will be a negative value equal to `SOCK_ERR_TIMEOUT`.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.8 send

Prototype	`int8_t send(SOCKET sock, void *buf, uint16_t len, uint16_t flags);`
Description	Sends data from a locally TCP created socket to a remote TCP socket. Once a TCP socket has been created and connected to a remote host this function can be called to send data to the remote host. After the data is sent the application will receive a `M2M_SOCKET_SEND_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock — Socket ID returned from socket() buf — Pointer to application buffer containing data to be sent len — Number of bytes to send. Must be less than or equal to `SOCKET_BUFFER_MAX_LENGTH` flags — Not used by WINC1500 driver; set to 0.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.9 sendto

Prototype	`int8_t sendto(SOCKET sock, void *buf, uint16_t len, uint16_t flags, struct sockaddr *to, uint8_t tolen);`
Description	Sends data from a locally UDP created socket to a remote UDP socket. Once a UDP socket has been created and bound this function can be called to send data to the remote host. After the data is sent the application will receive a `M2M_SOCKET_SENDTO_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	sock — Socket ID returned from socket() buf — Pointer to application buffer containing data to be sent len — Number of bytes to send. Must be less than or equal to `SOCKET_BUFFER_MAX_LENGTH` flags — Not used by WINC1500 driver; set to 0. to — IP address and port number of remote UDP socket. Fill in '`struct sockaddr_in`' and cast it to '`struct sockaddr`'. See `struct sockaddr_in` in Section 6.3.16 tolen — *to* length in bytes. Not used by WINC1500 driver; only included for BSD compatibility.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.10 close

Prototype	`int8_t close(SOCKET sock);`
Description	Closes previously created socket. If close() is called while there are still pending messages (sent or received) they will be discarded.
Inputs	sock — Socket ID returned from socket()
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.11 gethostbyname

Prototype	`int8_t gethostbyname(const char *name);`
Description	Requests a DNS look-up to get the IP address of the specified host name. After the WINC1500 resolves the name the application will receive a `M2M_SOCKET_DNS_RESOLVE_EVENT` in the callback `m2m_socket_handle_events()`.
Inputs	name — Name to resolve (a null-terminated string). The name must be less than or equal to `M2M_HOSTNAME_MAX_SIZE`.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.12 setsockopt

Prototype	`int8_t` **`setsockopt`**`(SOCKET socket,uint8_t level, uint8_t optname, const void *optval, uint16_t optlen);`
Description	Sets socket options.
Inputs	sock — Socket ID returned from socket() level — Protocol level (must be set to `SOL_SOCKET` or `SOL_SSL_SOCKET`) optname — Option to be set. If *level* **is SOL_SOCKET** then options are: `SO_SET_UDP_SEND_CALLBACK` — Enable/disable `M2M_SOCKET_SEND_EVENT` for UDP sockets. Since UDP is unreliable the application may not be interested in this event. Enabled if optval points to a true value; disabled if *optval* points to a false value. `IP_ADD_MEMBERSHIP` — Applies only to UDP sockets. This option is used to receive frames sent to a multicast group. The desired multicast IP address should be in a uint32_t (big-endian) pointed to by *optval*. `IP_DROP_MEMBERSHIP` — Applies only to UDP sockets. This option is used to stop receiving frames from a multicast group. *optval* should point to a uint32_t multicast IP address (big-endian). If level is **SOL_SSL_SOCKET** then values are: `SO_SSL_BYPASS_X509_VERIF` — This option allows an opened SSL socket to bypass the X509 certificate verification process. It is highly recommended to NOT use this socket option in production software; it should only be used for debugging. The *optval* should point to an int boolean value. `SO_SSL_SNI` — This option sets the Server Name Indicator (SNI) for an SSL socket. *optval* should point to a null-terminated string containing the server name associated with the connection. The name length must be less than or equal to `M2M_HOSTNAME_MAX_SIZE`. `SO_SSL_ENABLE_SESSION_CACHING` — This option allows the TLS to cache the session information for faster TLS session establishment in future connections using the TLS Protocol session resume feature. optval — Pointer to option value optlen — Length of *optVal*, in bytes
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.13 getsockopt

Prototype	`int8_t` **`getsockopt`**`(SOCKET sock, uint8_t level, uint8_t optname, const void *optval, uint8_t *optlen);`
Description	Gets socket options.
Inputs	sock — Socket ID returned from socket() level — Protocol level (must be set to `SOL_SOCKET` or `SOL_SSL_SOCKET`) optname — Option to be get (see `setsockopt()`. optval — Pointer to buffer where option value will be written optlen — Size of buffer pointed to by *optval*,in bytes (used to prevent overflow)
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).

6.3.14 m2m_ping_req

Prototype	`void` **`m2m_ping_req`**`(uint32_t destIpAddress, uint8_t ttl);`
Description	Sends a ping request to the specified IP address.
Inputs	destIpAddress — Destination IP address in big-endian format ttl — IP TTL value for the ping request. If set to zero then default value will be used. TTL is the number of router hops allowed before discarding the packet.
Returns	None

6.3.15 sslEnableCertExpirationCheck

Prototype	`int8_t sslEnableCertExpirationCheck(uint8_t enable);`		
Description	Enable/Disable the WINC1500 SSL certificate expiration check.		
Inputs	enable	0	Ignore certificate expiration time check while handling TLS.Handshake.Certificate Message. If there is no system time available or the certificate is expired, the TLS Connection shall succeed.
		1	Validate the certificate expiration time. If there is no system time or the certificate expiration is detected, the TLS Connection shall fail.
Returns	`SOCK_ERR_NO_ERROR` on success, else a negative value (see `t_socketError` in source code).		

6.3.16 struct sockaddr_in

```
struct sockaddr_in
{
    uint16_t    sin_family;
    uint16_t    sin_port;
    in_addr     sin_addr;
    uint8_t     padding[8];
};
```

Field	Description
sin_family	Always `AF_INET`
sin_port	Port number of socket (cannot be 0)
sin_addr	This is a nested structure with one element, s_addr, which is the IP address of socket (must be in big-endian format). This value can be set to 0 to accept any IP address when using the socket as a server.
padding	Not used

6.4 Time API

6.4.1 m2m_wifi_enable_sntp

Prototype	`void` **`m2m_wifi_enable_sntp`**`(uint8_t bEnable);`
Description	Enables or disables the Simple Network Time Protocol(SNTP) client. The WINC1500 SNTP client is enabled by default and is used to set the WINC1500 system clock to UTC time from one of the 'well-known' timer servers (e.g. time-c.nist.gov). The default SNTP client updates the time once every 24 hours. The UTC (Coordinated Universal Time) is needed for checking the expiration date of X509 certificates when establishing TLS connections. If the host has a real-time clock (RTC) then the SNTP could be disabled and the application could set the system time via `m2m_wifi_set_system_time()`.
Inputs	bEnable Set to true to enable SNTP client; set to false to disable SNTP client
Returns	None

6.4.2 m2m_wifi_set_system_time

Prototype	`void` **`m2m_wifi_set_system_time`**`(uint32_t utcSeconds);`
Description	Sets the system time in UTC seconds. If the host MCU has a real-time clock then the SNTP client can be disabled (see `m2m_wifi_enable_sntp`()) and this function can be used to set the WINC1500 system time.
Inputs	utcSeconds UTC seconds (number of seconds elapsed since 00:00:00, January 1, 1970)
Returns	None

6.4.3 m2m_wifi_get_system_time

Prototype	`void` **`m2m_wifi_get_system_time`**`(void);`
Description	Issues a request to the WINC1500 for the current system time. When the time is available the `M2M_WIFI_SYS_TIME_EVENT` is generated. See the *Wi-Fi Events* section.
Inputs	None
Returns	None

6.5 Over-the-Air (OTA) API

The WINC1500 is capable of updating its firmware from a wireless network. This section describes the functions required to perform this action.

6.5.1 m2m_ota_start_update

Prototype	`void` **`m2m_ota_start_update`**`(char *downloadUrl);`
Description	Causes the WINC1500 to download the OTA image and ensure integrity of the image. Upon success (or failure) of the download, the `M2M_OTA_STATUS_EVENT` is generated with `updateStatusType` equal to `M2M_OTA_DOWNLOAD_STATUS_TYPE`. Switching to the new image is not automatic; the application must call `m2m_ota_switch_firmware()`.See *OTA Events* in Section 7.3. **Note**: A Wi-Fi connection is required prior to calling this function.
Inputs	downloadUrl – URL to retrieve the download image from (null-terminated string)
Returns	None

6.5.2 m2m_ota_switch_firmware

Prototype	`void` **`m2m_ota_switch_firmware`**`(void);`
Description	Switches to the new OTA firmware image. After a successful OTA update the application must call this function to have the WIN1500 switch to the new (OTA) image. Upon success (or failure), the `M2M_OTA_STATUS_EVENT` is generated with `updateStatusType` equal to `M2M_OTA_SOFTWARE_STATUS_TYPE`. See *OTA Events* in Section 7.3. If successful, a system restart is required.
Inputs	None
Returns	None

6.5.3 m2m_ota_rollback

Prototype	`void` **`m2m_ota_rollback`**`(void);`
Description	Request OTA Roll-back to the older (other) WINC1500 image. The WINC1500 will check the validity of the Roll-back image before switching to it. Upon success (or failure) of the roll-back the `M2M_OTA_STATUS_EVENT` is generated with `updateStatusType` equal to `M2M_OTA_ROLLBACK_STATUS_TYPE`. See *OTA Events* in Section 7.3. If successful, a system restart is required.
Inputs	None
Returns	None

6.5.4 m2m_ota_abort

Prototype	`void` **`m2m_ota_abort`**`(void);`
Description	Aborts an OTA download if one is in progress. The validity of the original image is checked and, if valid, the WINC1500 will switch to it. Upon success (or failure) of the abort the `M2M_OTA_STATUS_EVENT` is generated with `updateStatusType` equal to `M2M_OTA_ABORT_STATUS`. See *OTA Events* in Section 7.3.
Inputs	None
Returns	None

6.6 Provisioning API

6.6.1 m2m_wifi_start_provision_mode

Prototype	`void` **`m2m_wifi_start_provision_mode`**`(tstrM2MAPConfig *pstrM2MAPConfig, char * pcHttpServerDomainName, uint8_t enableHttpRedirect);`
Description	The WINC1500 will start in SoftAP mode and start the HTTP Provision WEB Server. When the provisioning is complete the M2M_WIFI_PROVISION_INFO_EVENT is generated..
Inputs	pstrM2MAPConfig — Configuration for the SoftAP (see *tstrM2MAPConfig* in Section 6.2.11.1) pcHttpServerDomainName — Domain name of the HTTP Provision WEB server which will be used to load the provisioning home page from a browser. The domain name can have one of the following 3 forms as shown in the examples below: 1: "wincprov.com" 2: "http://wincprov.com" 3: "https://wincprov.com" bEnableHttpRedirect — Enable or disable the HTTP redirect feature. This parameter is ignored for a secure provisioning session (e.g. using "https" in the prefix). Possible values are: 0: Do not use HTTP Redirect. The associated device can open the provisioning page only when the HTTP Provision URL of the WINC HTTP Server is correctly written on the browser. 1: Use HTTP Redirect. All HTTP traffic (http://URL) from the associated device (Phone, PC, etc) will be redirected to the WINC HTTP Provisioning home page.
Returns	None

6.6.2 m2m_wifi_stop_provision_mode

Prototype	`void` **`m2m_wifi_stop_provision_mode`**`(void);`
Description	Stops the HTPP provisioning mode
Inputs	None
Returns	None

6.7 Utility API

6.7.1 inet_ntop4

Prototype	`void inet_ntop4(uint32_t src, char *dest);`
Description	Converts a binary IPv4 address in big-endian order to a ASCII string.
Inputs	src IPv4 address in big-endian order dest Pointer to where ASCII string will be written. The buffer should be `M2M_INET4_ADDRSTRLEN` bytes in length
Returns	None

6.7.2 inet_pton4

Prototype	`void inet_pton4(char *src, uint32_t *dest);`
Description	Converts an ASCII IPv4 address to a binary IPv4 address in big-endian order.
Inputs	src Pointer to IPv4 ASCII string (null-terminated) dest Pointer to where binary IP address will be written in big-endian order
Returns	None

6.7.3 m2m_get_elapsed_time

Prototype	`uint32_t m2m_get_elapsed_time(uint32_t startTime);`
Description	Returns the number of milliseconds since the input start time. This function is using the 1ms timer, thus it could be off by 1ms.
Inputs	startTime Starting time from which to measure elapsed time. This parameter should be obtained from `m2mStub_GetOneMsTimer()`.
Returns	Elapsed time, in milliseconds.

6.7.4 m2m_wifi_prng_get_random_bytes

Prototype	`void m2m_wifi_prng_get_random_bytes(uint16_t size);`
Description	Issues a request to the WINC1500 to generate one or more psuedo-random byte values. When the bytes are ready the `M2M_WIFI_PRNG_EVENT` is generated. See the *Wi-Fi Events* section.
Inputs	size Number of psuedo-random bytes to generate. Must be between 1 and `M2M_MAX_PRNG_BYTES` (16).
Returns	None

6.7.5 m2m_wifi_enable_firmware_log

Prototype	`void` **`m2m_wifi_enable_firmware_log`**`(uint8_t enable);`
Description	The WINC1500 UART debug output is on by default. By `M2M_DISABLE_FIRMWARE_LOG` define is used to disable the debug output during host initialization. This function can be used to dynamically enable or disable WINC1500 debug output during runtime.
Inputs	enable 0 to disable, 1 to enable
Returns	None

6.7.6 m2m_wifi_send_crl

Prototype	`void` **`m2m_wifi_send_crl`**`(t_m2mWifiTlsCrlInfo *p_crl);`
Description	Notifies the WINC1500 with the Certificate Revocation List. Used with TLS.
Inputs	p_crl See `t_m2mWifiTlsCrlInfo` below
Returns	None

6.7.6.1 t_m2mWifiTlsCrlInfo

```
typedef struct
{
    uint8_t            crlType;
    uint8_t            padding[3];
    t_wfTlsCrlEntry    tlsCrl[TLS_CRL_MAX_ENTRIES];
} t_m2mWifiTlsCrlInfo;
```

Field	Description
`crlType`	Type of certificate contained in the list. Must be either: `M2M_TLS_CRL_TYPE_NONE` [No CRL check] `M2M_TLS_CRL_TYPE_CERT_HASHE` [CRL contains certificate hashes]
`padding`	Not used
`tlsCrl`	List of CRL's (see `t_m2mWifiTlsCrlEntry` below)

6.7.6.2 t_m2mWifiTlsCrlEntry

```
typedef struct
{
    uint8_t    dataLen;
    uint8_t    data[M2M_WIFI_TLS_CRL_DATA_MAX_LEN];
    uint8_t    padding[3];
} t_m2mWifiTlsCrlEntry;
```

Field	Description
`dataLen`	Length of certificate data (maximum is `M2M_WIFI_TLS_CRL_DATA_MAX_LEN`)
`data`	Certificate data
`padding`	Not used

6.7.7 m2m_wifi_init_download_mode

Prototype	`void` **`m2m_wifi_init_download_mode`**`(void);`
Description	Puts the WINC1500 in the firmware download mode allowing the host MCU to update WINC1500 firmware.
Inputs	None
Returns	None

6.8 Status API

6.8.1 nm_get_firmware_info

Prototype	`void` **`nm_get_firmware_info`**`(tstrM2mRev *p_revision);`
Description	Gets the WINC1500 Firmware version.
Inputs	p_revision Pointer to where version information is written. See *tstrM2mRev* below
Returns	None

6.8.1.1 tstrM2mRev

```
typedef struct
{
    uint32_t    u32Chipid;
    uint8_t     u8FirmwareMajor;
    uint8_t     u8FirmwareMinor;
    uint8_t     u8FirmwarePatch;
    uint8_t     u8DriverMajor;
    uint8_t     u8DriverMinor;
    uint8_t     u8DriverPatch;
    uint8_t     BuildDate [sizeof(__DATE__)];  // date of WINC1500 firmware build
    uint8_t     BuildTime [sizeof(__TIME__)];  // time of WINC1500 firmware build
    uint8_t     padding1;
    uint16_t    u16FirmwareSvnNum;     // not used
    uint16_t    padding2[2];
} tstrM2mRev;
```

6.8.2 nm_get_ota_firmware_info

Prototype	`void` **`nm_get_ota_firmware_info`** `(t_wfRevision *p_revision);`
Description	Gets the WINC1500 OTA Firmware version.
Inputs	p_revision Pointer to where version information is written. See *tstrM2mRev* above.
Returns	None

6.8.3 nmi_get_rfrevid

Prototype	`uint32_nmi_get_rfrevid(void);`
Description	Gets the WINC1500 RF version ID.
Inputs	None
Returns	RF Version ID.

7 Events

There are four categories of events:

- Wi-Fi events
- Socket events
- OTA (Over-The-Air) update events
- Error Events

7.1 Wi-Fi Events

Wi-Fi events must be customized to suit the application. The callback function is `m2m_wifi_handle_events()` in Section 4.5.1.

The WINC1500 driver calls the event callback function to notify the application of Wi-Fi events. The `eventCode` parameter is described in Table 7-1 below. The `p_eventData` parameter points to a 'C' union of containing all possible Wi-Fi event data (see *t_wifiEventData* in Section 7.1.1). Not all events have data associated with them – in this case the pointer will be NULL. When an event occurs, the event data should be read as soon as possible before another event occurs which will overwrite data from the previous event.

If the event data is to be retrieved outside the event handler function, the utility function `m2m_wifi_get_wifi_event_data()` returns a pointer to the `t_wifiEventData` union.

Table 7-1: Wi-Fi Event Codes

eventCode	Description
`M2M_WIFI_DRIVER_INIT_EVENT`	This event occurs after `m2m_wifi_init()` is called. It signals that the internal initialization is complete and normal operations can commence. There is no event data associated with this event.
`M2M_WIFI_IP_CONFLICT_EVENT`	Signals that the WINC1500 has detected an IP address conflict. See `m2m_wifi_set_static()`. See the `conflictedAddress` field in `t_wifiEventData`.
`M2M_WIFI_RSSI_EVENT`	This event occurs after calling `m2m_wifi_req_curr_rssi()` is called. See the `rssi` field in `t_wifiEventData`.
`M2M_WIFI_SCAN_DONE_EVENT`	Signals that a previous scan request has completed. This event occurs after calling either `m2m_wifi_request_scan`, `m2m_wifi_request_scan`, `m2m_wifi_req_hidden_ssid_scan`.
`M2M_WIFI_SCAN_RESULT_EVENT`	Signals that a previously requested scan result is ready. This event occurs after calling `m2m_wifi_request_scan`, `m2m_wifi_request_scan`, or `m2m_wifi_req_hidden_ssid_scan`. See the `scanResult` field in `t_wifiEventData`.
`M2M_WIFI_CONN_STATE_CHANGED_EVENT`	Signals that the Wi-Fi connection state has changed. This will occur after calling `m2m_wifi_connect()`,`m2m_wifi_connect_sc()`, or `m2m_default_connect()`. After a connection is established this event can also occur if the connection is lost. See the `connState` field in `t_wifiEventData`.
`M2M_WIFI_IP_ADDRESS_ASSIGNED_EVENT`	Signals that the WINC1500 DHCP client has succeeded and has received an IP address from a DHCP server. See the `ipconfig` field in

eventCode	Description
	t_wifiEventData.
M2M_WIFI_SYS_TIME_EVENT	Signals that the system time request has completed. This event is occurs after calling m2m_wifi_set_system_time(). See the sysTime field in t_wifiEventData.
M2M_WIFI_CONN_INFO_RESPONSE_EVENT	Signals that the connection information request has completed. This event occurs after calling m2m_wifi_get_connection_info(). See the connInfo field in t_wifiEventData.
M2M_WIFI_WPS_EVENT	Signals that a WPS event has occurred. This event occurs after calling m2m_wifi_wps(). See the wpsInfo field in t_wifiEventData.
M2M_WIFI_PROVISION_INFO_EVENT	Signals that the HTTP provisioning server has sent the provisioning information to the WINC1500. See the provisionInfo field in t_wifiEventData.
M2M_WIFI_DEFAULT_CONNNECT_EVENT	Signals that a default connection has occurred (or failed). This event occurs after a call to m2m_default_connect(). See the defaultConnInfo field in t_wifiEventData.
M2M_WIFI_PRNG_EVENT	Signals that pseudo-random bytes are ready. This event occurs after calling m2m_wifi_prng_get_random_bytes(). See the prng field in t_wifiEventData.

7.1.1 t_wifiEventData

This structure is a union of all possible Wi-Fi event data structures.

```
typedef union t_wifiEventData
{
    uint32_t                conflictedIpAddress;
    int8_t                  rssi;
    tstrM2mScanDone         scanDone;
    tstrM2mWifiscanResult   scanResult;
    tstrM2mWifiStateChanged connState;
    tstrM2MIPConfig         ipConfig;
    tstrSystemTime          sysTime;
    tstrM2MConnInfo         connInfo;
    tstrM2MWPSInfo          wpsInfo;
    tstrM2MProvisionInfo    provisionInfo;
    tstrM2MDefaultConnResp  defaultConnInfo;
    tstrM2MPrng             prng;
} t_wifiEventData;
```

Field	Description
conflictedIpAddress	This event data is associated with the M2M_WIFI_IP_CONFLICT_EVENT. When the WINC1500 detects an IP address conflict the IP address with the conflict is stored in this field. The IP address will be in big-endian format. The event data is a pointer to a uint32_t.
rssi	This event data is associated with the M2M_WIFI_RSSI_EVENT. The event is generated after calling m2m_wifi_req_curr_rssi(). The event data is a pointer to an int8_t.
scanDone	This event data is associated with the M2M_WIFI_SCAN_DONE_EVENT. The event is generated after calling one of the scan request functions. The event data is a pointer to: typedef struct

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>

```
{
    uint8_t     u8NumofCh;
    int8_t      s8ScanState;
    uint8_t     padding[2];
} tstrM2mScanDone;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>u8NumofCh</td><td>Number of AP's found during the scan.</td></tr>
<tr><td>s8ScanState</td><td>Not used</td></tr>
<tr><td>padding</td><td>Not used</td></tr>
</table>

</td></tr>
<tr><td>scan_result</td><td>

This event data is associated with the M2M_WIFI_SCAN_RESULT_EVENT. This event is generated after calling m2m_wifi_request_scan, m2m_wifi_request_scan, or m2m_wifi_req_hidden_ssid_scan. The event data is a pointer to:

```
typedef struct
{
    uint8_t     u8index;
    int8_t      s8rssi;
    uint8_t     u8AuthType;
    uint8_t     u8ch;
    uint8_t     au8BSSID [6];
    char        au8SSID [M2M_WIFI_MAX_SSID_LEN];
    uint8_t     padding;
} tstrM2mWifiscanResult;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>u8index</td><td>Index of AP in the scan result list</td></tr>
<tr><td>s8rssi</td><td>AP signal strength</td></tr>
<tr><td>u8AuthType</td><td>AP authentication type. Will be one of the following:
<table>
<tr><td>M2M_WIFI_SEC_OPEN</td><td>Open security</td></tr>
<tr><td>M2M_WIFI_SEC_WPA_PSK</td><td>WPA/WPA2 personal (PSK)</td></tr>
<tr><td>M2M_WIFI_SEC_WEP</td><td>WEP 40/104, Open or shard</td></tr>
<tr><td>M2M_WIFI_SEC_802_1X</td><td>WPA/WPA2 Enterprise.IEEE802.1x user-name/password authentication</td></tr>
</table>
</td></tr>
<tr><td>u8ch</td><td>AP RF channel number</td></tr>
<tr><td>au8BSSID</td><td>BSSID of AP</td></tr>
<tr><td>au8SSID</td><td>SSID of AP (null-terminated string)</td></tr>
<tr><td>padding</td><td>Not used</td></tr>
</table>

</td></tr>
<tr><td>connState</td><td>

This event data is associated with the M2M_WIFI_CONN_STATE_CHANGED_EVENT. The event is generated after calling m2m_wifi_connect() as well as after a successful connection is that connection is subsequently lost. The event data is a pointer to:

```
typedef struct
{
    uint8_t     u8CurrState;
    uint8_t     u8ErrCode;
    uint8_t     padding[2];
} tstrM2mWifiStateChanged;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
</table>

</td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td>u8CurrState</td><td>M2M_WIFI_DISCONNECTED or M2M_WIFI_CONNECTED</td></tr>
<tr><td>u8ErrCode</td><td>If state is M2M_WIFI_DISCONNECTED, the reason for the disconnect. See t_m2mWifiConnChangedErrCode in source code.</td></tr>
<tr><td>padding</td><td>Not used</td></tr>
</table>
</td></tr>
<tr><td>ipConfig</td><td>
This event data is associated with the M2M_WIFI_IP_ADDRESS_ASSIGNED_EVENT. This event is generated after the DHCP client succeeds in obtaining an IP address. The event data is a pointer to:
<pre>
typedef struct
{
    uint32_t u32StaticIP;
    uint32_t u32Gateway;
    uint32_t u32DNS;
    uint32_t u32SubnetMask;
    uint32_t u32DhcpLeaseTime;
} tstrM2MIPConfig;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>u32StaticIp</td><td>Static IP address assigned to device</td></tr>
<tr><td>u32Gateway</td><td>IP address of the default gateway</td></tr>
<tr><td>u32DNS</td><td>IP address for the DNS server</td></tr>
<tr><td>u32SubnetMask</td><td>Subnet mask for the local area network</td></tr>
<tr><td>u32DhcpLeaseTime</td><td>DHCP lease time, in seconds</td></tr>
</table>
Note: All IP addresses will be in big-endian format.
</td></tr>
<tr><td>sysTime</td><td>
This event is generated after calling m2m_wifi_set_system_time().The event data is a pointer to:
<pre>
typedef struct
{
    uint16_t   u16Year;
    uint8_t   u8Month;
    uint8_t   u8Day;
    uint8_t   u8Hour;
    uint8_t   u8Minute;
    uint8_t   u8Second;
    uint8_t   padding; // not used
} t_m2mWifiSysTime;
</pre>
</td></tr>
<tr><td>connInfo</td><td>
This event is generated after calling m2m_wifi_get_connection_info. The event data is a pointer to:
<pre>
typedef struct
{
    char   acSSID [M2M_WIFI_MAX_SSID_LEN];
    uint8_t   u8SecType;
    uint8_t   au8IPAddr [4];
    uint8_t   macAddress[6];
    int8_t   rssi;
    uint8_t   padding[3];
} tstrM2MConnInfo;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>acSSID</td><td>AP connection SSID name. Only valid in station mode. Will be NULL in SoftAP or P2P.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td></td><td></td></tr>
<tr><td>u8SecType</td><td>Security type; see t_m2mSecurityType</td></tr>
<tr><td>au8IPAddr</td><td>Connection IP address</td></tr>
<tr><td>au8MACAddress</td><td>MAC address of the peer Wi-Fi station</td></tr>
<tr><td>s8RSSI</td><td>Connection RSSI signal</td></tr>
</table>
</td></tr>
<tr><td>wpsInfo</td><td>
This event is generated after calling m2m_wifi_wps(). The event data is a pointer to:
<pre>
typedef struct
{
    uint8_t   u8AuthType;
    uint8_t   u8Ch;
    uint8_t   au8SSID[M2M_WIFI_MAX_SSID_LEN];
    uint8_t   au8PSK[M2M_WIFI_MAX_PSK_LEN];
} tstrM2MWPSInfo;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>u8AuthType</td><td>Network authentication type. Will be one of the following:
<table>
<tr><td>M2M_WIFI_SECURITY_INVALID</td><td>WPS failed</td></tr>
<tr><td>M2M_WIFI_SEC_OPEN</td><td>Open security</td></tr>
<tr><td>M2M_WIFI_SEC_WPA_PSK</td><td>WPA/WPA2 personal (PSK)</td></tr>
<tr><td>M2M_WIFI_SEC_WEP</td><td>WEP 40/104, Open or shared</td></tr>
<tr><td>M2M_WIFI_SEC_802_1X</td><td>WPA/WPA2 Enterprise. IEEE802.1x user-name and password authentication</td></tr>
</table>
</td></tr>
<tr><td>u8Ch</td><td>RF channel for the AP</td></tr>
<tr><td>au8SSID</td><td>SSID obtained from WPS</td></tr>
<tr><td>au8PSK</td><td>PSK obtained from WPS</td></tr>
</table>
</td></tr>
<tr><td>provisionInfo</td><td>
This event is generated when the HTTP provisioning server sends the provision information to the WINC1500. The event data is a pointer to:
<pre>
typedef struct
{
    uint8_t   au8SSID [M2M_WIFI_MAX_SSID_LEN];
    uint8_t   au8Password [M2M_WIFI_MAX_PSK_LEN];
    uint8_t   u8SecType;
    uint8_t   status;
} tstrM2MProvisionInfo;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>au8SSID</td><td>Provisioned SSID</td></tr>
<tr><td>au8Password</td><td>Provisioned password</td></tr>
<tr><td>u8SecType</td><td>Provisioned security type (see t_wfSecurityType in source code).</td></tr>
<tr><td>u8Status</td><td>Provisioning status – true if provisioning is valid, else false. If false, the above data is invalid.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>defaultConnInfo</td><td>This event is generated after calling m2m_default_connect(). The event data is a pointer to:
<pre>typedef struct
{
    int8_t   s8ErrorCode;
    uint8_t   padding[3];
} tstrM2MDefaultConnResp;</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>errorCode</td><td>See t_m2mWifiDefaultConnectErrorCode in the source code.</td></tr>
<tr><td>padding</td><td>Not used</td></tr>
</table></td></tr>
<tr><td>prng</td><td>This event is generated after calling m2m_wifi_prng_get_random_bytes. The event data is a pointer to:
<pre>typedef struct
{
    uint8_t   buf[M2M_MAX_PRNG_BYTES]; // return buffer
    uint16_t   size;   // PRNG size requested
} tstrM2MPrng;</pre></td></tr>
</table>

7.2 Socket Events

Socket events are handled must be customized to suit the application. The callback function is `m2m_socket_handle_events()` in Section 4.5.2.

The WINC1500 driver calls the socket event callback function to notify the application of socket events. The `eventCode` parameter is described in Table 7-2: Socket Event Codes below. The `p_eventData` parameter points to a 'C' union of containing all possible socket event data (see *t_socketEventData* in Section 7.2.1). Not all events have data associated with them – in this case the pointer will be NULL. When an event occurs, the event data should be read as soon as possible before another event occurs which will overwrite data from the previous event.

If the event data is to be retrieved outside the event handler function, the utility function `m2m_wifi_get_socket_event_data()` returns a pointer to the `t_socketEventData` union.

Table 7-2: Socket Event Codes

eventCode	Description
`M2M_SOCKET_BIND_EVENT`	This event signals a bind has completed; it occurs after a call to `bind()`. See the `bindStatus` field in `t_socketEventData`.
`M2M_SOCKET_LISTEN_EVENT`	This event signals a listen has completed; it occurs after a call to `listen()`. See the `listenStatus` field in `t_socketEventData`.
`M2M_SOCKET_SEND_EVENT`	This event signals that data has been sent to the remote TCP socket; it occurs after a call to `send()`. See the `numSendBytes` field in `t_socketEventData`.
`M2M_SOCKET_SENDTO_EVENT`	This event signals that data has been sent to the remote UDP socket; it occurs after a call to `sendto()`. See the `numSendBytes` field in `t_socketEventData`.
`M2M_SOCKET_ACCEPT_EVENT`	This event signals an accept has completed; it occurs after a call to `accept()`. See the `acceptResponse` in `t_socketEventData`.
`M2M_SOCKET_CONNECT_EVENT`	This event signals a TCP connect has occurred; it occurs after a call to `connect()`. See the `connectResponse` field in `t_socketEventData`:
`M2M_SOCKET_DNS_RESOLVE_EVENT`	This event signals that a host name has been resolved to an IP address; it occurs after a call to `gethostbyname()`. See the `dnsReply` field in `t_socketEventData`:
`M2M_SOCKET_RECV_EVENT`	This event signals that data has been received on a TCP socket; it occurs after a call to `recv()` when the remote peer sends data to the WINC1500. See the `recvMsg` field in `t_socketEventData`.
`M2M_SOCKET_RECVFROM_EVENT`	This event signals that data has been received on a UDP socket; it occurs after a call to `recvfrom()` when the remote peer sends data to the WINC1500. See the `recvMsg` field in `t_socketEventData`.
`M2M_SOCKET_PING_RESPONSE_EVENT`	This event signals that a ping response or a timeout to a ping request has been received; it occurs after a call to `m2m_ping_req()`. See the `pingReply` field in `t_socketEventData`.

7.2.1 t_socketEventData

This structure is a union of all possible socket event data structures.

```
typedef union t_socketEventData
{
    int8_t            bindStatus;
    int8_t            listenStatus;
    int16_t           numSendBytes;
    t_socketAccept    acceptResponse;
    t_socketConnect   connectResponse;
    t_dnsReply        dnsReply;
    t_socketRecv      recvMsg;
    t_pingReply       pingReply;
} t_socketEventData;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>bindStatus</td><td>This event data is associated with the M2M_SOCKET_BIND_EVENT. The value will be SOCK_ERR_NO_ERROR on success, else a negative value (see t_socketError in source code).</td></tr>
<tr><td>listenStatus</td><td>The event data is associated with the M2M_SOCKET_LISTEN_EVENT. The value will be SOCK_ERR_NO_ERROR on success, else a negative value (see t_socketError in source code).</td></tr>
<tr><td>numSendBytes</td><td>This event data is associated with the M2M_SOCKET_SEND_EVENT or M2M_SOCKET_SENDTO_EVENT. The value is the number of bytes that were sent. If the value is negative an error occurred (see t_socketError in source code).</td></tr>
<tr><td>acceptResponse</td><td>This event data is associated the M2M_SOCKET_ACCEPT_EVENT. The event data is a pointer to the acceptResponse in t_socketEventData:
<pre>typedef struct
{
    SOCKET   sock;
    struct sockaddr_in strAddr;
} t_socketAccept;</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>sock</td><td>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 (see t_socketError in source code).</td></tr>
<tr><td>strAddr</td><td>IP address and port number of the remote peeer. See struct sockaddr_in in Section 6.3.16.</td></tr>
</table></td></tr>
<tr><td>connectResponse</td><td>This event data is associated with the M2M_SOCKET_CONNECT_EVENT. The event data is a pointer to
<pre>typedef struct
{
    SOCKET   sock;
    int8_t   error;
} t_socketConnect;</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>sock</td><td>On a successful connect operation, the return information is the socket ID for the connecting socket. Otherwise a negative error code is returned to indicate failure of the accept operation (see t_socketError in source code).</td></tr>
<tr><td>error</td><td>SOCK_ERR_NO_ERROR if successful, else a negative number. See</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td></td><td>t_socketError in source code.</td></tr>
</table>
</td></tr>
<tr><td>dnsReply</td><td>
The event data is associated with the M2M_SOCKET_DNS_RESOLVE_EVENT. The event data is a pointer to:
<pre>
typedef struct
{
    char   hostName[M2M_HOSTNAME_MAX_SIZE];
    uint32_t   hostIp;
} t_dnsReply;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>hostName</td><td>Host name that was resolved</td></tr>
<tr><td>hostIp</td><td>IP address of resolved host name (big-endian)</td></tr>
</table>
</td></tr>
<tr><td>recvMsg</td><td>
This event data is associated with the M2M_SOCKET_RECV_EVENT and the M2M_SOCKET_RECVFROM_EVENT. The event data is a pointer to:
<pre>
typedef struct
{
    uint8_t   *p_rxBuf;
    int16_t   bufSize;
    uint16_t   remainingSize;
    struct sockaddr_in ai_addr;
} t_socketRecv;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>p_rxBuf</td><td>Pointer to the application buffer (this was a parameter to recv()) containing the Rx data.</td></tr>
<tr><td>bufSize</td><td>Length of data in p_rxBuf. If this value is negative than an error occurred (see t_socketError in source code). If the value is 0 then the socket connection was closed.</td></tr>
<tr><td>remainingSize</td><td>The number of bytes remaining in the current recv operation. This can occur is the data sent from the remote host is larger than the buffer size created for this socket. In this case, another event will occur to retrieve the next chunk of data.</td></tr>
<tr><td>struct sockaddr_in</td><td>Only valid for UDP sockets. Not used for TCP sockets. Contains the IP address and port number of the remote UDP peer.</td></tr>
</table>
</td></tr>
<tr><td>pingReply</td><td>
This event data is associated with the M2M_SOCKET_PING_RESPONSE_EVENT. The event data is a pointer to:
<pre>
typedef struct
{
    uint32_t   ipAddress;
    uint32_t   rtt;
    t_m2mPingErrorCode   errorCode;
} t_pingReply;
</pre>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>ipAddress</td><td>IP address of the ping respondent (big-endian)</td></tr>
<tr><td>rtt</td><td>Round trip time in milliseconds</td></tr>
<tr><td>errorCode</td><td>One of the following:
M2M_PING_SUCCESS</td></tr>
</table>
</td></tr>
</table>

Field	Description
	M2M_PING_DEST_UNREACHABLE M2M_PING_TIMEOUT

7.3 OTA Events

OTA events are associated with downloading and switching to a new WINC1500 firmware image downloaded via the Wi-Fi network. The callback function is `m2m_ota_handle_events()` in Section 4.5.3.

The WINC1500 driver calls the OTA event callback function to notify the application of OTA events. The eventCode parameter is described in Table 7-3: OTA Event Codes below. The `p_eventData` parameter points to a 'C' structure containing the OTA event data (see *t_otaEventData* in Section 7.3.1).

If the event data is to be retrieved outside the event handler function, the utility function `m2m_wifi_get_ota_event_data()` returns a pointer to the `t_otaEventData` structure.

Table 7-3: OTA Event Codes

eventCode	Description
`M2M_OTA_STATUS_EVENT`	This event code is used for all OTA events. The event data is a pointer to the `t_otaEventData` structure described below.

7.3.1 t_otaEventData

```
typedef struct t_otaEventData
{
    t_m2mOtaUpdateStatus  otaUpdateStatus;  // see t_m2mOtaUpdateStatus below

} t_otaEventData;
```

7.3.2 t_m2mOtaUpdateStatus

```
typedef struct
{
    uint8_t     updateStatusType;
    uint8_t     updateStatus;
    uint8_t     padding[2];
} t_m2mOtaUpdateStatus;
```

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>updateStatusType</td><td>Indicates what status is being reported. The range is:
<table>
<tr><td>M2M_OTA_DOWNLOAD_STATUS_TYPE</td><td>Status of an OTA download; will occur after calling m2m_ota_start().</td></tr>
<tr><td>M2M_OTA_SOFTWARE_STATUS_TYPE</td><td>Status of an OTA switch to the new firmware. Will occur after a call to m2m_ota_switch_firmware().</td></tr>
<tr><td>M2M_OTA_ROLLBACK_STATUS_TYPE</td><td>Status of a rollback; will occur after calling m2m_ota_rollback().</td></tr>
<tr><td>M2M_OTA_ABORT_STATUS</td><td>Status of a download abort; will occur after calling m2m_ota_abort().</td></tr>
</table></td></tr>
<tr><td>updateStatus</td><td>Indicates the result of the operation described in updateStatusType. Will be one of the following:
<table>
<tr><td>OTA_STATUS_SUCCESS</td><td>OTA operation was successful.</td></tr>
<tr><td>OTA_STATUS_FAIL</td><td>Generic failure</td></tr>
<tr><td>OTA_STATUS_INVALID_ARG</td><td>Invalid or malformed download URL</td></tr>
<tr><td>OTA_STATUS_INVALID_RB_IMAGE</td><td>Invalid rollback image</td></tr>
</table></td></tr>
</table>

Field	Description
	OTA_STATUS_INVALID_FLASH_SIZE — Flash size on device is not enough for OTA OTA_STATUS_AlREADY_ENABLED — An OTA operation is already enabled OTA_STATUS_UPDATE_IN_PROGRESS — An OTA operation update is in progress OTA_STATUS_IMAGE_VERIFY_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 operation aborted
padding	Not used

7.4 Error Events

The application is notified of error events via the callback function `m2m_error_handle_events()` (see Section 4.5.4). Error codes are defined in wf_errors.h.