MediaTek LinkIt™ Connect 7681 API Reference

Version: 1.1

Release date: 13th May 2016

Specifications are subject to change without notice.

Document Revision History

Revision	Date	Description
1.0	3rd Jan, 2015	Initial Release
1.1	13th May, 2016	Replaced long dashes with short dashes in commands

Table of Contents

Lists of tables and figures

1. Introduction

This document is a reference to the AT commands and APIs available for the MediaTek LinkIt Connect 7681 development platform.

AT commands are described in section 2.

The LinkIt Connect 7681 APIs are described in sections:

- 3, Interface API
- 4, Wi-Fi API
- 5, Control API
- 6, Security API

The LinkIt Connect 7681 APIs are in C/C++ style and data types follow C++ conventions and common extensions.

2. AT Commands

This section describes the AT commands that can be sent to the MT7861 over the UART.

2.1. Display Version

Send system version details as a message to UART.

Command	`Ver`
Argument Descriptions	None
Example	`AT#Ver`

2.2. Reboot

Reboot the MT7681.

Command	`Reboot`
Argument Descriptions	None
Example	`AT#Reboot`

2.3. Restore Default Settings

Reset the system to the default values defined in `iot_custom.c`.

Command	`Default`
Argument Descriptions	None
Example	`AT#Default`

The default settings, located in `iot_custom.c`, are specified in a common configuration structure `IOT_COM_CFG` as follows:

```
File : iot_custom.c

/*this is the Common CFG region default table */
IOT_COM_CFG Com_Cfg = {
                        DEFAULT_BOOT_FW_IDX,
                        DEFAULT_RECOVERY_MODE_STATUS,
                        DEFAULT_IO_MODE,

                        DEFAULT_UART_BAUDRATE,
                        DEFAULT_UART_DATA_BITS,
                        DEFAULT_UART_PARITY,
                        DEFAULT_UART_STOP_BITS,

                        DEFAULT_TCP_UDP_CS,
                        DEFAULT_IOT_TCP_SRV_PORT,
                        DEFAULT_LOCAL_TCP_SRV_PORT,
                        DEFAULT_IOT_UDP_SRV_PORT,
                        DEFAULT_LOCAL_UDP_SRV_PORT,
                        DEFAULT_USE_DHCP,
                        DEFAULT_STATIC_IP,
                        DEFAULT_SUBNET_MASK_IP,
                        DEFAULT_DNS_IP,
                        DEFAULT_GATEWAY_IP,
                        DEFAULT_IOT_SERVER_IP,
                        DEFAULT_IOT_CMD_PWD
                     };

/*this is the User CFG region default table */
IOT_USR_CFG Usr_Cfg = {
                        DEFAULT_VENDOR_NEME,
                        DEFAULT_PRODUCT_TYPE,
                        DEFAULT_PRODUCT_NAME
                     };
```

The default Station mode settings (such as the connected SSID, PMK, AuthMode and alike) are reset to NULL/zero as a result of running this command. These values are set again when the MT7681 finishes the MediaTek Smart Connection process and has obtained an IP address from the wireless AP.
The default AP settings are not set by this command, to do that use `AT#SoftApConf -d0`.

2.4. Switch Channel

Switch the channel used for Wi-Fi communications.

Command	`Channel`
Argument Descriptions	-b <Bandwidth> (0 for BW_20, 1 for BW_40) -c <channel number> (1 to 14)
Example	`AT#Channel -b0 -c6`

When `-b` (bandwidth) is set to `1 (BW 40)`, `-c` (channel number) defines the central channel for BW40.

2.5. Configure UART

Configure the settings of the UART port.

Command	`Uart`
Argument Descriptions	`-b <baud rate>` (57600, 115200, 230400 ...) `-w <data bits>` (5, 6, 7, 8) `-p <parity>` (0 for no parity, 1 for odd, 2 for even) `-s <stop bits>` (1 for 1bit, 2 for 2bits, 3 for 1.5bits)
Example	`AT#Uart -b 57600 -w 7 -p 1 -s 1`

This command effects the UART settings only temporarily: the settings aren't saved to Flash and stored settings are reapplied when the board is restarted or rebooted.

For more information on working with UART, see section 3.2, "UART".

2.6. Update Firmware from UART

Set the MT7681 to Recovery mode to accept new firmware on the UART port using x-Modem protocol.

Command	`UpdateFW`
Argument Descriptions	None
Example	`AT#UpdateFW`

2.7. Enter Smart Connection State

Force the MT7681 to enter the Smart Connection state and wait for an over-the-air package containing wireless-AP settings.

Command	`Smnt`
Argument Descriptions	None
Example	`AT#Smnt`

When this command is sent the MT7681 changes its state machine to Smart Connection mode and listens for an over the air Smart Connection packet.
This command is available in Station mode only.

2.8. Power Saving Level

Request the MT7681 to enter a sleep state at a particular power saving level.

Command	`PowerSaving`
Argument Descriptions	`-l <PowerSaving Level>` (1~5) `-t <PowerSleep Time>` (0~0xFFFFFF (unit: us)) `-r <read PowerSaving Level>`
Example	`AT#PowerSaving -l1 -t0xFFFFFF` `AT#PowerSaving -r<Space>`

MT7681 will go into deep sleep when it receives the command `AT#PowerSaving -l1 -t0xFFFFFF`, and will wake after `0xFFFFFF` micro seconds (16.777215 seconds).
This command is available in Station mode only.

There are 5 power saving levels for MT7681 station mode: in each power saving level the MT7681 disables more internal modules (Blocks) as listed in Table 1.

Level	Disabled Blocks	Wakeup Time
Level 5	Switch Regulator, LDO_DIG, Crystal, BBPLL and CPU clock	3 to 20ms
Level 4	LDO_DIG, Crystal, BBPLL and CPU clock	1 to 20ms
Level 3	Crystal, BBPLL and CPU clock	1 to 20ms
Level 2	BBPLL and CPU clock	200us
Level 1	CPU clock	2us

Table 1 Power saving levels of the MT7681

Level 5 offers the most power saving option and has the longest wakeup period. The wakeup procedure from a power saving state is handled by an internal power saving counter. When the power saving level is 1 or 2 the MT7681 will wake and handle any UART or GPIO interrupt. However, when in power saving levels of 3, 4 or 5 the MT7681 will NOT wake on a UART or GPIO interrupt.

At the time of writing, power saving isn't available while the MT7681 is in AP mode.

2.9. Read or Write Flash

Read or write data to Flash.

Command	FLASH
Argument Descriptions	-l <Offset> read data at the specified offset from Flash -s <Offset> save data to the specified offset on Flash -v <Value> value of the data
Example	AT#FLASH -l6 AT#FLASH -s6 -v56

For information on data types and their storage layout in the external Flash, please see Appendix A, "Flash Layout", in the MediaTek LinkIt Connect 7681 Developer's Guide.

2.10. Configure SoftAP

Configure the wireless-AP settings for SoftAP mode.

Command	SoftAPConf
Argument Descriptions	-s<SSID name> (maximum 32 characters) -c <AP channel> (1~14) -a <AP Auth Mode> (0:Open, 4:WPA-PSK, 7:WPA2-PSK, 9:WPA/WPA2-PSK) -p<AP Password> (maximum 32 characters) -m (Store current AP config into FLASH) -d (Clears current AP config from FLASH)
Example	1: AT#SoftAPConf -s[ssid] -c[channel] -a[auth_mode] -p[password] 2: AT#SoftAPConf -m0 3: AT#SoftAPConf -d0

This command is available in AP mode only.

The default AP settings, located in `ap_pub.c`, are as follows:

```
#define AP_MODE_OPEN                     0
#define AP_MODE_WPAPSK_TKIP              1
#define AP_MODE_WPA2PSK_AES              2
#define AP_MODE_WPA1WPA2PSK_TKIPAESMIX   3

#define DEFAULT_AP_MODE     AP_MODE_WPA1WPA2PSK_TKIPAESMIX //AP_MODE_OPEN
#define Default_Ssid        "MT7681_AP1"
#define Default_Password    "12345678"
```

3. Interface API

This section describes the APIs that provide features for manipulating the interfaces available on MT7681.

3.1. SPI Flash

MT7681 has an SPI interface to connect to 1MB of serial Flash memory, for the storage of firmware and configuration data. The Flash SPI interface has:

- a 10MHz clock.
- default support for serial Flash with 4 kB sectors and 64 kB blocks.
- access to 4 kB region for ad-hoc use.

The SPI Flash APIs provide convenient wrappers to the SPI commands for reading and writing to and from Flash memory. As a reference the native commands are listed in section 3.1.7, 'SPI Command'.

3.1.1. spi_flash_read

This function returns a pointer to the data at a specified Flash memory location.

Syntax	`spi_flash_read(addr, data, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`addr`	`uint32`	The offset at which the data being read is stored on the Flash
	IN	`len`	`uint16`	The length of data to be read
	OUT	`data`	`uint8*`	The pointer indicating the read data
Return Value	`(int32)` returning values: • `0`, read succeeded • non-zero, read failed			

This API is slightly faster than `spi_flash_read_m2`.

3.1.2. spi_flash_read_m2

This function is used to return a pointer to data at a specified location in Flash using the SPI command method. This function is similar to `spi_flash_read()` with a source code available in `spi-flash_pub.c`.The source code enables extending function support to other Flash specific commands.

Syntax	`spi_flash_read_m2(addr, data, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`addr`	`uint32`	The offset at which the data being read is stored on the Flash
	IN	`len`	`uint16`	The data length to be read
	OUT	`data`	`uint8*`	The pointer to the read data
Return Value	(`int32`) returning values: • `0` — successful read • non-zero — read failed			

3.1.3. spi_flash_write_func

This function is used to write a block of data to Flash, without erasing the existing data first.

Syntax	`spi_flash_write_func(addr, data, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`addr`	`uint32`	The offset at which the data will be written to the Flash
	IN	`len`	`uint16`	The data length to be written
	IN	`data`	`uint8`	The pointer indicating the data to be written
Return Value	`0` (`int32`)— successful write non-zero (`int32`) — write failed			

In order to change small units of data within a block it's recommended that `spi_flash_write` is used. This function resolves inefficient allocation of 4 kB of RAM in the MT7681.`spi_flash_erase_sector` or `spi_flash_erase_block` should be used to erase any existing data before using `spi_flash_write` function, or the write will be unsuccessful.

3.1.4. spi_flash_write

This function is used to write data to Flash. Unlike `spi_flash_write_func`, this function reads the existing sector, erases the sector, adds the new data to the read sector and writes the updated sector back to Flash. The operation of this function is shown in Figure 1.

Syntax	`spi_flash_write(addr, data, len)`
Parameters	[IN] `addr (uint32)` — The offset at which the data will be written to the Flash [IN] `len (uint16)` — The data length to be written [IN] `data (uint8)` — The pointer indicating the data to be written
Return Value	`0 (int32)` — write succeeded non-zero (`int32`) —write failed

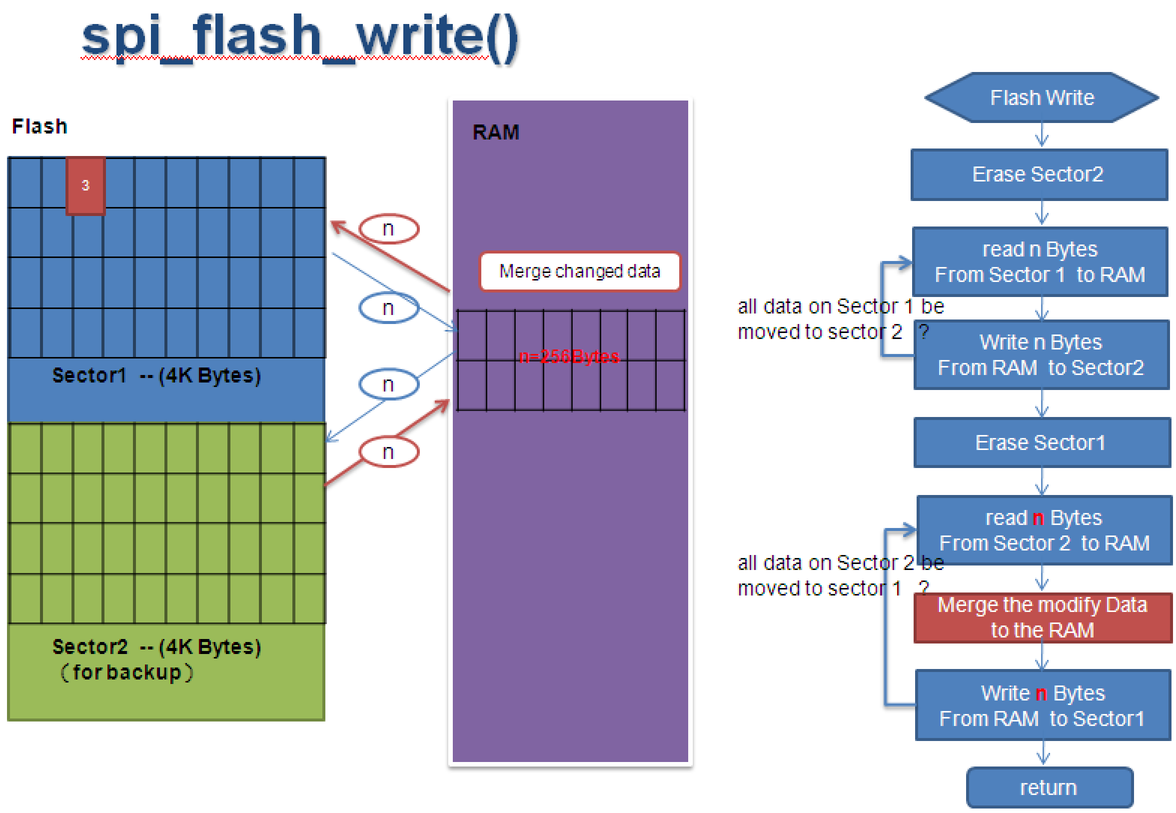


Figure 1 The behavior of spi_flash_write

A limitation of `spi_flash_write` is that `len` must be <= `FLASH_OFFESET_WRITE_BUF` (4kB).
Due to the limited read/write life cycle of Flash memory, consider not to overuse this API by making frequent small data changes (particularly modifying whole sectors at a time). Where possible, batch the data changes.

3.1.5. spi_flash_erase_sector

This function is used to erase the sector at the specified address. The return address of an erased content becomes `0xFF`.

Syntax	`spi_flash_erase_sector(addr)`
Parameters	`[IN] addr (uint32)` — the address of the sector in Flash to be erased
Return Value	`(void)` — None

- The default size of sectors is 4 kB.
- Due to the characteristics of Flash, it's mandatory to erase a sector before writing to it.
- This function erases all data in the sector, the data erased cannot be restored, use with caution.

3.1.6. spi_flash_erase_block

This function is used to erase the block at the specified address. The return address of an erased content becomes `0xFF`.

Syntax	`spi_flash_erase_block(addr)`
Parameters	[IN] addr `(uint32)` — the address of the block in Flash to be erased
Return Value	`(void)` — None

- The default size of block is 64 kB.
- Due to the characteristics of Flash, it's mandatory to erase a block before writing to it.
- This function erases all data from the block, the data erased cannot be restored, use with caution.

3.1.7. SPI Commands

Reference Table 2 shows the SPI commands available for reading/writing to/from Flash memory.

Type	Offset	Bits	Type	Description	Initial value
PUB_SPICMD_WR_BYTE	0x0000	[31:8]	-	Reserved	-
		[7:0]	WO	Write 1 byte on SPI	8'b0
PUB_SPICMD_WR_LAST_BYTE	0x0004	[31:8]	-	Reserved	-
		[7:0]	WO	Write the last byte on SPI	8'b0
PUB_SPICMD_RD_BYTE	0x0018	[31:1]	-	Reserved	-
		0	WO	Read 1 byte on SPI	1'b0
PUB_SPICMD_RD_LAST_BYTE	0x001C	[31:1]	-	Reserved	-
		0	WO	Read the last byte on SPI	1'b0

Table 2 The SPI commands for reading and writing Flash

3.2. UART

MT7681 provides a single UART port with a 16Byte UART transmitter/receiver (Tx/Rx) FIFO buffer. The UART API enables use of this port to exchange data between PCs and other devices, with the same UART baud rate.

The (40 MHz) system clock in the MT7681 restricts the use of exact values of common baud rates. The actual baud rate for commonly used baud rates is calculated as follows:

`dlr` [rounded divisor latch] `= round (systemclock / (16 * baudrate), 0)`

`actual_baudrate = systemclock / (16 * dlr)`

where `systemclock = 40*1000000 (Hz)`

Table 3 shows some examples of the baud rate calculation for common baud rates

target baud rate	16 x target baud rate	dl = systemclock / (16 x target baud rate)	round(dl)	output = systemclock / (16 x dlr)
57600	921600	43.40277778	43	58139.53488
115200	1843200	21.70138889	22	113636.3636
230400	3686400	10.85069444	11	227272.7273

Table 3 Baud rate calculation examples

It may still be necessary to experiment with baud rate to find the precise value that works for any particular computer or device.

The UART settings can be changed in a number of ways:

- By storing a new configuration to Flash at the memory locations shown in Table 4.

Flash position	UART configuration	Len	Offset
0x18018	UART Baud rate	4	24
0x1801C	UART Data bits	1	28
0x1801D	UART Parity bits	1	29
0x1801E	UART Stop bits	1	30

Table 4 Memory locations for UART settings

- Modifying the values in the source file (`iot_custom.c`) as shown below, upgrading the firmware then resetting to default values by using the `AT#Default` command.

```
#define DEFAULT_UART_BAUDRATE          UART_BAUD_115200
#define DEFAULT_UART_DATA_BITS         len_8
#define DEFAULT_UART_PARITY            pa_none
#define DEFAULT_UART_STOP_BITS         sb_1
```

MT7681 registers UART ISR handler callbacks for UART Tx and Rx:

- `uart_rx_cb()` is called when the UART Rx interrupt assert is triggered indicating the Rx FIFO is close to full.
- `uart_tx_cb() is called when` the UART Tx Done handler is triggered after the UART Tx interrupt assert, which occurs when `iot_uart_output()` will fill the Tx buffer and in turn trigger a UART Tx event.

3.2.1. iot_uart_output

This function writes a given length of data to the UART port.

Syntax	`iot_uart_output(msg, count)`
Parameters	[OUT] `msg (uint8*)` — Pointer to a UART Tx buffer [IN] `count (int32)` — Length of data to write
Return Value	`0 (int32)`

`msg` is copied to the UART Tx buffer declared in `UARTTxBuf[UARTTX_RING_LEN]` and initialized in `uart_customize_init()`, which is called in `bsp_init()`.

The following example shows how the AT command process function `iot_atcmd_exec_ver()` in`iot_at_cmd.c`, parses `AT#Ver` and outputs the current software version, using `iot_uart_output`.

```
int16 iot_atcmd_exec_ver(puchar pCmdBuf)
{
    size_t  len=0 , len2 =0;

    /* the response header format is:  "AT#CmdType=" */
    iot_atcmd_resp_header((int8 *)pCmdBuf, &len, AT_CMD_PREFIX, 
AT_CMD_VER);

    /*AT command Version*/
    len2 = strlen(FW_VERISON_CUST);
    memcpy(pCmdBuf + len, FW_VERISON_CUST, len2);
    len += len2;

    len2 = strlen("\n");
    if( (len + len2 + 3) >= AT_CMD_MAX_LEN) {
        return -1;
    }
    memcpy(pCmdBuf + len, "\n", len2);
    len += len2;
    iot_uart_output((puchar)pCmdBuf, (int16)len);
    return 0;
}
```

3.3. GPIO

MT7681 has 5 GPIO pins: GPIO0 to GPIO4. These GPIO pins have two modes of operation:

- input mode. In this mode the external signal input to the GPIO pin can be queried. For example: when High Signal is fed to GPIO2 in an input mode, it is possible to get the GPIO2 input value by calling iot_gpio_read().
- output mode. In this mode GPIO pin can output High or Low signals by calling iot_gpio_output(). It is also possible to get the output status of a GPIO pin by calling iot_gpio_read().

The mode of a GPIO pin is set in iot_gpio_input() and iot_gpio_output().

The mode and value of multiple GPIO pins can be set using iot_gpios_mode_chg() or iot_gpios_output().

MT7681 supports GPIO interrupts when a GPIO pin is in the input mode. There are 4 trigger options: no trigger, rising trigger, falling trigger, both rising and falling trigger. See, iot_cust_set_gpiint_mode(), iot_cust_get_gpiint_mode() and iot_cust_gpiint_hdlr().

3.3.1. iot_gpio_read

This function retrieves the current mode (input or output) and value (high or low) from a single GPIO pin.

Syntax	iot_gpio_read(gpio_num, pVal, pPolarity)
Parameters	[IN] gpio_num (int32) —Specify the GPIO number, which can be 0 to 4. [OUT] pPolarity (uint8*) — Read the GPIO polarity, 0=output, 1=input [OUT] pVal (uint8*) —Read the GPIO status, 0=low, 1=high
Return Value	(void) returns no values

3.3.2. iot_gpio_input

This function sets a GPIO pin to input mode and reads its input value.

Syntax	iot_gpio_input(gpio_num, input)
Parameters	[IN] gpio_num (int32) — The GPIO pin, which can be 0 to 4. [OUT] input (uint32*) — The input status of the given GPIO pin. 0=low, 1=high.
Return Value	0 (int32) — pin set and read 1 (int32) — gpio_num is invalid. 2 (int32) — input is invalid.

3.3.3. iot_gpio_output

This function configures the output status of a GPIO pin to be high or low.

Syntax	`iot_gpio_output(gpio_num, output)`
Parameters	[IN] `gpio_num (int32)` -- The GPIO pin number, which can be 0 to 4. [IN] `output (int32)` — The output status of the given GPIO pin. `0`=low, `1`=high.
Return Value	`(int32)` with values: • `0`, GPIO pin output set. • `1, gpio_num` is invalid. • 2, if output is invalid.

3.3.3. iot_gpios_mode_chg

This function configures multiple GPIO pins to output mode. It uses a bitmap to indicate which pins should be set or not.

Syntax	`iot_gpios_mode_chg(output_bitmap)`
Parameters	[IN] `output_bitmap (uint32)` —Specifies the bitmap defining which GPIO pins will be set to output mode. Bit(i) stands for gpio(i) and the value ranges from 00000B to 11111B. For example: • 01001B, will set `gpio0` and `gpio3` to output mode, the others to input mode. • 00110B, will set `gpio1` and `gpio2` to output mode, the others to input mode.
Return Value	`(int32)` taking the values: • `0`, which has no specific meaning

3.3.4. iot_gpios_output

This function configures a batch of GPIO pins to specific output states.

Syntax	`iot_gpios_output(output_bitmap, value_bitmap)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`output_bitmap`	`uint32`	Specifies the GPIO output mode bitmap, where Bit(i) stands for gpio(i) from the range 00000B to 11111B.
	IN	`value_bitmap`	`uint32`	Specifies the GPIO output status bitmap where Bit(i) stands for gpio(i) from the range 00000B to 11111B.
Return Value	`(int32)` taking the values: • `0`, which has no specific meaning			

This function does not change GPIO pins to output mode. It modifies the output value only.

Table 5 shows examples of the two bitmaps and their effect on the GPIO pins.

output_bitmap	value_bitmap	result
10001	10000	gpio0=low, gpio4=high
00110	00010	gpio1=high, gpio2=low

Table 5 Examples of iot_gpios_output bit maps

3.3.5. iot_cust_set_gpiint_mode

This function sets the interrupt mode for one GPIO pin

<table>
<tr><td>Syntax</td><td colspan="4"><code>iot_cust_set_gpiint_mode(GPIO_Num, Val)</code></td></tr>
<tr><td rowspan="3">Parameters</td><td>Mode</td><td>Name</td><td>Type</td><td>Description</td></tr>
<tr><td>IN</td><td><code>GPIO_Num</code></td><td>uint8</td><td>Takes values 0 to 4, mapping to GPIO0 to GPIO4</td></tr>
<tr><td>IN</td><td><code>Val</code></td><td><code>uint8</code></td><td>Takes values 0 to 4,
• 0: no trigger,
• 1: falling edge trigger
• 2: rising edge trigger
• 3: both falling and rising edge trigger</td></tr>
<tr><td>Return Value</td><td colspan="4">(<code>uint8</code>) returning values:
• 0, success
• 1, invalid input</td></tr>
</table>

3.3.6. iot_cust_get_gpiint_mode

This function gets the interrupt mode setting for all GPIO pins at once, returning settings in a binary array.

Syntax	`iot_cust_get_gpiint_mode(pGPI_INT_MODE)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	OUT	pGPI_INT_MODE	uint16*	Returns the binary array: • [b1:b0] GPIO0 interrupt mode • [b3:b2] GPIO1 interrupt mode • [b5:b4] GPIO2 interrupt mode • [b7:b6] GPIO3 interrupt mode • [b9:b8] GPIO4 interrupt mode Where each binary pair denotes: • 00: no trigger • 01: falling edge trigger • 10: rising edge trigger • 11: both falling and rising edge trigger
Return Value	(void) returning no value			

1.1.1 iot_cust_gpiint_hdlr

This handler is called if any GPIO Interrupt is triggered.

Syntax	`iot_cust_gpiint_hdlr(GPI_STS)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	GPI_STS	uint8	A bitmask indicating which GPIOs were triggered. The values of bit[0] to bit[4] map to interrupt status of GPIO0 to GPIO4. For example. bit[0] = 1 means this interrupt event was triggered by GPIO0
Return Value	(void) returning no values			

To illustrate the functionality of this API, take an example where GPIO1 is set to:

- an input mode, defined by `iot_gpio_input(1, &input)`
- an interrupt mode as 'falling and rising edge trigger', defined by `iot_cust_set_gpiint_mode(1, 3)`

When the input signal to `GPIO1` changes from 0 to 1 or from 1 to 0, `iot_cust_gpiint_hdlr(GPI_STS)` will be called with the `bit[1]` of `GPI_STS` set to 1, as shown below.

```
void iot_cust_init(void)
{
    /* run customer initial function */
    uint32 input = 0;
    iot_gpio_input(1, &input);
    iot_cust_set_gpiint_mode(1, 3)};
}

void iot_cust_gpiint_hdlr(IN uint8 GPI_STS)
{
    if ((GPI_STS >> 0) & 0x01) {
        printf_high("GPIO_0 interrupted\n");
    } else if ((GPI_STS >> 1) & 0x01) {
        printf_high("GPIO_1 interrupted\n");
    } else if ((GPI_STS >> 2) & 0x01) {
        printf_high("GPIO_2 interrupted\n");
    } else if ((GPI_STS >> 3) & 0x01) {
        printf_high("GPIO_3 interrupted\n");
    } else if ((GPI_STS >> 4) & 0x01) {
        printf_high("GPIO_4 interrupted\n");
    } else {
        printf_high("Ignored\n");
    }
}
```

3.4. PWM

MT7681 does not support hardware PWM. It, however, provides software emulated PWM on GPIO0 to GPIO4. The available options for PWM frequency and duty cycle are limited by the 1ms precision of the hardware time on the MT7681, so that:

Frequency * duty cycle = 1000

For example:

- If the duty cycle is 20, frequency is 50Hz.
- If the duty cycle is 10, frequency is 100Hz)

The default PWM resolution (20) is defined by the macro `PWM_HIGHEST_LEVEL`.

3.4.1. IoT_sw_pwm_add

This function configures the PWM frequency and duty cycle of a GPIO pin.

Syntax	`IoT_sw_pwm_add(gpio_num, dutycycle, resolution)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`gpio_num`	`uint8`	GPIO number, takes values 0 to 4.
	IN	`dutycycle`	`uint16`	PWM duty cycle, takes values 0 to `resolution` (value of 3rd parameter) in milli seconds.
	IN	`resolution`	`uint16`	PWM resolution in milli seconds
Return Value	(`int32`) returning values: • 0, duty cycle and frequency set on the pin • 1, `gpio_num` is invalid.			

Note the following:

`frequency = 1000 / resolution`

`dutycycle` (as a percentage) = `100 * dutycycletime / resolution`

Figure 2 shows the relationship between the PWM waveform and the duty cycle and resolution.

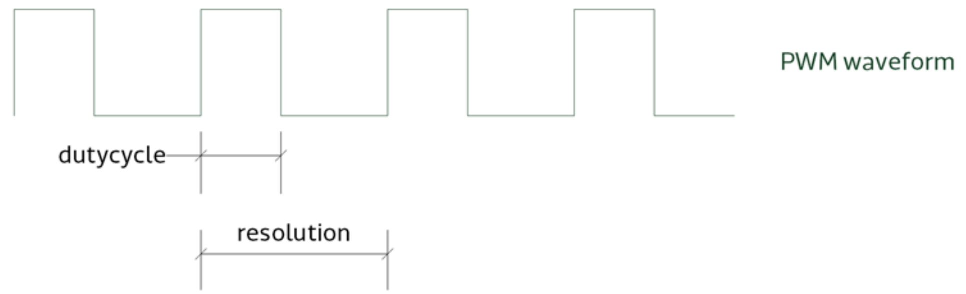


Figure 2 Relationship between PWM duty cycle and resolution

To illustrate the use of this function: the following example runs:

- GPIO3 at PWM frequency 50Hz with 5% duty cycle
- GPIO4 at PWM frequency 100Hz with 60% duty cycle

```
void iot_cust_init( void)
{
    iot_sw_pwm_add (3, 1, 20); // GPIO3 at 5% duty cycle  [Freq=1000/20=50Hz]
    iot_sw_pwm_add (4, 6, 10); // GPIO4 at 60% duty cycle
[Freq=1000/10=100Hz]
}
```


3.4.2. iot_sw_pwm_del

This function disables PWM and returns a GPIO pin to the default output mode and value. Default output value is defined by macro `DEFAULT_GPIO(nn)_OUTVAL` where `(nn)` corresponds to the GPIO pin number, 0 to 4.

Syntax	`iot_sw_pwm_del(gpio_num)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	OUT	`gpio_num`	`uint8`	Specifies the GPIO pin number, takes values 0 to 4.
Return Value	(`int32`) returning values: • 0, pin reset • 1, `gpio_num` is invalid.			

4 Wi-Fi API

This section describes the APIs available for Wi-Fi setting and communication functions, and includes hardware and software Rx Filters.

3.5. Rx Filter

This API provides an interface to get and set Rx hardware filters —such as data packet, management packet, control frame packet and alike — to limit the packets received by the MT7681. It also provides for setting software Rx Filters. The hardware filtering helps improve efficiency and increase the performance of communication functions. The filter options available are detailed in Table 6.

Bits	Type	Name	Description	Initial value
15	R/W	DROP_BAR	Drop BAR	0
14	R/W	DROP_BA	Drop BA	1
13	R/W	DROP_PSPOLL	Drop PS-Poll	0
12	R/W	DROP_RTS	Drop RTS	1
11	R/W	DROP_CTS	Drop CTS	1
10	R/W	DROP_ACK	Drop ACK	1
9	R/W	DROP_CFEND	Drop CF-END	1
8	R/W	DROP_CFACK	Drop CF-END + CF-ACK	1
7	R/W	DROP_DUPL	Drop duplicated frame	1
6	R/W	DROP_BC	Drop broadcast frame	0
5	R/W	DROP_MC	Drop multicast frame	0
4	R/W	DROP_VER_ERR	Drop 802.11 version error frame	1
3	R/W	DROP_NOT_MYBSS	Drop frame that is not my BSSID	1
2	R/W	DROP_UC_NOME	Drop not to me unicast frame	1
1	R/W	DROP_PHY_ERR	Drop physical error frame	1
0	R/W	DROP_CRC_ERR	Drop CRC error frame	1

Table 6 The Rx Filter options available

When `bit5=1`, both BC/MC packets will be dropped even if `bit6=0`.

3.5.1. iot_get_rxfilter

This function queries the hardware Rx filter settings, returning a binary array.

Syntax	`iot_get_rxfilter()`
Parameters	None
Return Value	(`uint16`) value of Rx filter. See Table 6 for filter definitions.

3.5.2. iot_set_rxfilter

This function applies a new set of hardware Rx filters.

Syntax	`iot_set_rxfilter(Value)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`Value`	`uint16`	Binary array of Rx Filter values, see Table 6 for filter definitions.
Return Value	(`uint16`) returning the value written to RxFilter, which should be the same as input value.			

The default Rx filter setting is defined and cannot be modified at compile time. If you want to apply your own filter, please set a new Rx filter in the `wifi_state_machine()` where state changes apply. See section 4.4, "Wi-Fi State Machine" in the MediaTek LinkIt Connect 7681 Developers Guide for more details.

To illustrate the use of this function: When MT7681 is powered on, `iot_cust_init` (see below) is invoked and outputs the current Rx filter settings (`0x7F97`, the default value, a Hex representation of 0111 1111 1001 0111B). This default value applies for filters: BA, PS-Poll, RTS, CTS, ACK, CF-End, CF-End+CF-ACK, Duplicated, 80211VersionError, Not to Me Unicast, Physical Error and CRC Error.

When `IoT_Set_RxFilter(0x7f93)` is invoked, it disables the filtering of 'Not to Me Unicast' packets and MT7681 will receive more information.

```
void iot_cust_init( void)
{
    /* run customer initial function */
    uint16 uRxFilter = 0;
    uRxFilter= IoT_Get_RxFilter();
    printf_high("The Current Rx Filter setting is: 0x%x", uRxFilter);
    IoT_Set_RxFilter(0x7f93);
}
```

The Rx Filter is reset to its default value when the Wi-Fi state is changed by calling `wifi_state_chg()`.

3.5.3. WifiRxFsIntFilterOut

This function sets the software Rx filters and its source code is available for modifications to provide the required software Rx filtering. This function is called by the wireless Rx hardware interrupt. That is, if MT7681 hardware receives a wireless packet, it will trigger an Rx interrupt and invoke `WifiRxFsIntFilterOut()` to check the received packet. If `WifiRxFsIntFilterOut()` returns FALSE, the packet will be dropped, and if it returns TRUE the packet will be kept and copied to a RX QUEUE and processed in the `WifiRxDoneInterruptHandle()`.

Syntax	`WifiRxFsIntFilterOut(RxpBufDesc)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`RxpBufDesc`	`pBD_t`	Pointer to the Rx buffer (passed from the interrupt)
Return Value	(bool) returning values: • FALSE, packet dropped • TRUE, packet retained			

To illustrate the use of this function: in the following version of the source code (`rtmp_data_pub.c`) with MT7681 in AP mode, `WifiRxFsIntFilterOut()` is used to filter out Management packets.

```
bool WifiRxFsIntFilterOut(pBD_t RxpBufDesc)
{
    puchar          pBuff;
    PRXINFO_STRUC   pRxINFO;
    PRXWI_STRUC     pRxWI;
    PHEADER_802_11  pHeader;
    uint8           type;
    uint8           subtype;
    pBuff    = (puchar)RxpBufDesc->pBuf;
    pRxINFO  = (PRXINFO_STRUC)(pBuff);
    pRxWI    = (PRXWI_STRUC)(pBuff + RXINFO_SIZE);
    pHeader  = (PHEADER_802_11)(pBuff + RXINFO_SIZE + RXWI_SIZE);
    type     = pHeader->FC.Type;
    subtype  = pHeader->FC.SubType;

    if(RTMPCheckRxError(pHeader, pRxWI, pRxINFO) == NDIS_STATUS_FAILURE) {
        return TRUE;  /* free packet */
    }

#ifdef CONFIG_SOFTAP
    if (type == BTYPE_MGMT)  {
        /* There are too many Beacons from other AP routers, or management
           frame not send to MyBSS */
        /* Drop notMyBss management frame, except ProbeReq for response
           Active scan from STA*/
       if ((pRxINFO->MyBss == 0) && (subtype != SUBTYPE_PROBE_REQ))
            return TRUE;
    }
#endif
    return FALSE;
}
```

Refer to `rtmp_general_pub.h` for details of the static variables for 802.11 operations.

3.6. Wi-Fi Transmission and Reception

This group of functions provides features for Wi-Fi Tx and Rx manipulation.

3.6.1. sta_legacy_frame_tx

This function transmits an encrypted or non-encrypted 802.11 packet out to the air.

Syntax	`sta_legacy_frame_tx(PktBuff, PacketLen, bClearFrame)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`PktBuff`	`pBD_t`	Pointer to the send packet (mac802_3 + payload)
	IN	`PacketLen`	`uint16`	Len of packet (include mac802_3 and payload length)
	IN	`bClearFrame`	`bool`	Takes values: • FALSE: encrypt when sending • TRUE: do not encrypt when sending If `bClearFrame` is FALSE, and MT7681 is connected to an AP-router using WEP, WPA or WPA2 Auth Modes the frame will be encrypted with the pairwise key or group key that was negotiated with the AP router
Return Value	(`int`) returning values: • `NDIS_STATUS_SUCCESS`, packet successfully queued in `TxSwQueue`. • `NDIS_STATUS_FAILURE`, packet was not successfully queued.			

Refer to `rtmp_general_pub.h` for details of the static variables for 802.11 operations.

To illustrate the use of this API: the example below calls `sta_legacy_frame_tx()` to wirelessly transmit an unencrypted packet. The packet is filled with ones and has a size of `ATE_TX_PAYLOAD_LEN` (as defined in `ate.h`).

```
void SendATETxDataFrame(void)
{
    uint8                 *mpool;
    pBD_t                 pBufDesc;

    uint8 playload[ATE_TX_PAYLOAD_LEN] = {1};
    memset(playload, 1, sizeof(playload));

    //handle_FCE_TxTS_interrupt();
    pBufDesc = apiQU_Dequeue(&gFreeQueue2);
    if(pBufDesc ==NULL)
    {
        printf_high("=>%s DeQ fail\n",__FUNCTION__);
        return;
    }
    mpool = pBufDesc->pBuf;
    memcpy(mpool, playload, gATEInfo.PayLoadLen);
    // packet is not encrypted.
    sta_legacy_frame_tx(pBufDesc, gATEInfo.PayLoadLen, TRUE);
}
```

3.7. Channel and BW

The 802.11 family of specifications uses ISM (industrial, scientific and medical) bands as follows:

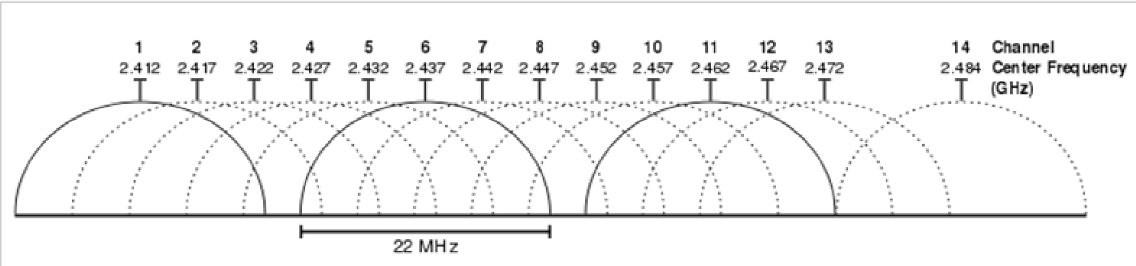
- 802.11b, 802.11g and 802.11n-2.4 utilize the 2.400 to 2.500 GHz spectrum
- 802.11a and 802.11n use the more heavily regulated 4.915 to 5.825 GHz band.

These are commonly referred to as the "2.4 GHz and 5 GHz bands".

Each spectrum is sub-divided into channels with a center frequency and bandwidth, analogous to the way radio and TV broadcast bands are sub-divided.

MT7681 supports the 2.4GHz ISM band only and provides for setting up channel and bandwidth parameters with `asic_set_channel()`.

The 2.4 GHz band is divided into 14 channels spaced 5 MHz apart, beginning with channel 1 which is centered at 2.412 GHz, see Figure 3. The remaining channels have additional restrictions or are unavailable for use in some regulatory domains.



Source: http://en.wikipedia.org/wiki/List_of_WLAN_channels

Figure 3 The Wi-Fi channels in the 2.4 GHz band

For each channel in 802.11b and 802.11g, 20MHz bandwidth (as per the 802.11 spec) is used.

The 2.4 GHz ISM band is fairly congested. With 802.11n there is an option to double the bandwidth per channel to 40 MHz, which results in slightly more than double the data rate.

The specification calls for one primary 20 MHz channel as well as a secondary adjacent channel spaced ±20 MHz away. The primary channel is used to communicate with clients incapable of 40 MHz mode. When in 40 MHz mode, the center frequency is actually the mean of the primary and secondary channels. For details, see Table 7.

Primary channel	20 MHz	40 MHz above			40 MHz below		
	Blocks	2nd ch.	Center	Blocks	2nd ch.	Center	Blocks
1	1–3	5	3	1–7	Not Available		
2	1–4	6	4	1–8	Not Available		
3	1–5	7	5	1–9	Not Available		
4	2–6	8	6	2–10	Not Available		
5	3–7	9	7	3–11	1	3	1–7
6	4–8	10	8	4–12	2	4	1–8
7	5–9	11	9	5–13	3	5	1–9
8	6–10	12	10	6–13	4	6	2–10
9	7–11	13	11	7–13	5	7	3–11
10	8–12	Not Available			6	8	4–12
11	9–13	Not Available			7	9	5–13
12	10–13	Not Available			8	10	6–13
13	11–13	Not Available			9	11	7–13

Table 7 802.11 channel specifications

For the channel and bandwidth descriptions, please refer to the 802.11 specification. You may also find the following information on Wikipedia helpful:

- IEEE 802.11
- IEEE 802.11n-2009
- List of WLAN channels

3.7.1. asic_set_channel

This function switches the channel, provides bandwidth and channel settings.

Syntax	`asic_set_channel(ch, bw, ext_ch)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`ch`	`uint8`	Channel number, takes values 1~14. If bw is `BW_40`, `ch` is the center channel.
	IN	`bw`	`uint8`	Bandwidth, take static variable `BW_20` or `BW40`
	IN	`ext_ch`	`uint8`	11n bandwidth setting, takes static variables `EXTCHA_NONE`, `EXTCHA_ABOVE` or `EXTCHA_BELOW`.
Return Value	(`int32`) returning values: • `0`, default, no specific meaning			

When `bw=BW_20`, `ext_ch` can be `EXTCHA_NONE` or `EXTCHA_ABOVE` only.
The static variables used in this function are defined in `iot_api.h`.

To illustrate the use of this function:

- `asic_set_channel(8, BW_40, EXTCHA_ABOVE)` switches to center channel 8 of bandwidth 40 in 40MHz above mode. The primary channel will be 6 (refer to Table 7).
- If you want to scan all primary channels from 1 to 13, you will use the following code:

```
// Scan primary channel from 1~9
for(i=3;i<=11;i++)
{
    asic_set_channel(i, BW_40, EXTCHA_ABOVE)
    // channel processing
}

// Scan primary channel from 10~13
for(i=8;i<=11;i++)
{
    asic_set_channel(i, BW_40, EXTCHA_BELOW)
    // channel processing
}
```

4. Control API

This API provides various functions to use control features of the MT7681, including:

- software timers
- hardware timers

4.1. Software Timer APIs

MT7681 has two hardware timers: `timer0` is used for system control and implementing the software timer and `timer1` is used for customization. The precision of both hardware timers is 1ms.

4.1.1. cnmTimerInitTimer

This function initializes a timer.

Syntax	`cnmTimerInitTimer(prTimer, pfFunc, u4Data, u4Data2)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	prTimer	P_TIMER_T	Pointer to a timer structure
	IN	pfFunc	PFN_MGMT_TIMEOUT_FUNC	Pointer to the call back function
	IN	u4Data	uint32	parameter1 for pfFunc
	IN	u4Data2	uint32	parameter2 for pfFunc
Return Value	(void) return no values			

4.1.2. cnmTimerStartTimer

This function starts a single shot (non-repeating) timer.

Syntax	`cnmTimerStartTimer(prTimer, u4TimeoutMs)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	prTimer	P_TIMER_T	Pointer to a timer structure
	IN	u4TimeoutMs	uint32	Timeout to issue the timer and callback function (unit: ms)
Return Value	(void) returns no values			

4.1.3. cnmTimerStopTimer

This function is used to stop a running timer.

Syntax	`cnmTimerStopTimer(prTimer)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	prTimer	P_TIMER_T	Pointer to a timer structure
Return Value	(void) returns no values			

To illustrate the use of this function: the following code starts a timer and outputs text to UART.

```
typedef struct GNU_PACKED _IOT_CUST_CFG_{
    TIMER_T    custTimer0;
```

```
    TIMER_T   custTimer1;      /* ◊ add a timer1 for this example */
} IOT_CUST_TIMER;

void CustTimer1TimeoutAction(uint32 param, uint32 param2)    /* ◊ add a
timeout action */
{
    static uint16 T1=0;
    printf_high("CustTimer1TimeoutAction: [%d] processing...\n", T1++);
    if (T1 < 3) {
        printf_high("CustTimer1 Start, will timeout after 3 second...\n");
        cnmTimerStartTimer (&IoTCustTimer.custTimer1, 3000);
    } else {
        printf_high("CustTimer1 Stop\n");
        cnmTimerStopTimer (&IoTCustTimer.custTimer1);
    }
}

void iot_cust_init( void)
{
    /* run customer initial function */
    /*for customer timer initialization*/
    printf_high("CustTimer1 Initialization...\n");
    cnmTimerInitTimer(&IoTCustTimer.custTimer1,  CustTimer1TimeoutAction, 0,
0);

    printf_high("CustTimer1 Start, will timeout after 2 second...\n");
    cnmTimerStartTimer (&IoTCustTimer.custTimer1, 2000);
}
```

Resulting in the following UART output:

```
==> Recovery Mode
<== Recovery Mode
(-)
SM=0, Sub=0
CustTimer1 Initialization...
CustTimer1 Start, will timeout after 2 second...
SM=1, Sub=0
[WTask]5480271
CustTimer1TimeoutAction: [0] processing...
CustTimer1 Start, will timeout after 3 second...
[WTask]5485272
CustTimer1TimeoutAction: [1] processing...
CustTimer1 Start, will timeout after 3 second...
CustTimer1TimeoutAction: [2] processing...
CustTimer1 Stop
[WTask]5490273
[WTask]5495274
[WTask]5500275
```


4.1.4. iot_get_ms_time

This function provides the time (in milliseconds (ms)) elapsed since the system started, to the precision of 1ms.

Syntax	iot_get_ms_time(void)
Parameters	None
Return Value	(uint32) returns the time in ms

To illustrate the use of this function: the following code outputs a message including the time in ms — "Hello, CurTime=5000", "Hello, CurTime=6000" and so on —every second.

```
void iot_cust_subtask1( void)
{
    static  uint32 PreTime = 0;
    uint32 CurTime = 0;

    CurTime = iot_get_ms_time();
    if (((CurTime - PreTime) >= 1000) || (CurTime < PreTime)) {
        PreTime = CurTime;
        /* one-second periodic execution */
        printf_high("Hello, CurTime = %d \n\n", CurTime);
    }
}
```

4.2. Hardware timer1 interrupt function

An interrupt is available for hardware `timer1`. The recommended frequency of the interrupt ticker is 1 to 10 ticks per second, and is defined as follows:

```
#define TICK_HZ_HWTIMER1 10   /*T = 1/TICK_HZ_HWTIEMR1*/
```

The interrupt ticker can run up to the resolution of the hardware clock, i.e. 1ms. However, ticks higher than 10 (100ms) may cause performance issues.

To illustrate the use of this interrupt: the following makes GPIO4 blink every 100ms using the tick value returned from `iot_cust_hwtimer1_tick()`.

```
#define TICK_HZ_HWTIMER1 10   /*T = 1/TICK_HZ_HWTIEMR1*/
void iot_cust_hwtimer1_hdlr(void)
{
    /*Sample code for HW timer1 interrupt handle*/
    /*Notice:  Do not implement too much process here, as it is running in 
interrupt level*/
    uint8 input, Polarity;
    /*Make GPIO 4 blinking by Timer1 HW EINT */
    iot_gpio_read(4, &input, &Polarity);
    iot_gpio_output(4, (input==0)?1:0 );
}
uint32 iot_cust_hwtimer1_tick(void)
{
    return TICK_HZ_HWTIMER1;
}
```

Where:

- `iot_cust_hwtimer1_tick()` is called by the hardware initialization layer to initialize hardware timer1.
- `iot_cust_hwtimer1_hdlr` is triggered every 100ms (that is `T=1/10`).

5. Security API

This section describes the functions available to implement aspects of Wi-Fi security, using the Cipher algorithm and Integrity check algorithm support provided in MT7681. You can use these algorithms to encrypt or decrypt communication data and perform integrity checks on this data.

Example CRC16 and CRC32 implementations are also provided, in `crypt_crc.c`.

5.1. AES functions

The following functions provide for various Advanced Encryption Standard features.

5.1.1. RT_AES_Decrypt

This function is used to decrypt data using the AES algorithm.

Syntax	`RT_AES_Decrypt(CipherBlock[], CipherBlockSize, Key[], KeyLen, PlainBlock[], PlainBlockSize)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`CipherBlock[]`	`uint8`	The block of cipher text, 16 Bytes (128 bit) each block
	IN	`CipherBlockSize`	`uint32`	The length of block of cipher text in bytes
	IN	`Key[]`	`uint8`	Cipher key , it maybe 16, 24 or 32bytes
	IN	`KeyLen`	`uint32`	The length cipher key in bytes
	IN	`PlainBlockSize`	`uint32`	The length of allocated plain block in bytes
	OUT	`PlainBlock[]`	`uint8`	Plain block to store plain text
	OUT	`PlainBlockSize`	`uint32*`	The length of the used plain block in bytes
Return Value	(void) returns no values			

`RT_AES_Decrypt()` only provides for 16 Bytes input and 16 Bytes output of `PlainBlock`.

5.1.2. RT_AES_Encrypt

This function is used to decrypt data using the AES algorithm.

Syntax	`RT_AES_Encrypt(PlainBlock[], PlainBlockSize, Key[], KeyLen, CipherBlock[], CipherBlockSize)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`PlainBlock[]`	`uint8`	The block of Plain text, 16 bytes (128 bit) each block
	IN	`PlainBlockSize`	`uint32`	The length of block of plain text in bytes
	IN	`Key[]`	`uint8`	Cipher key , it maybe 16, 24 or 32bytes
	IN	`KeyLen`	`uint32`	The length cipher key in bytes
	IN	`CipherBlockSize`	`uint32`	The length of allocated cipher block in bytes
	OUT	`CipherBlock[]`	`uint8`	Cipher text
	OUT	`CipherBlockSize`	`uint32*`	The length of the used cipher block in bytes
Return Value	(void) returns no values			

`RT_AES_Encrypt()` only provides for 16 Bytes input and 16 Bytes output of `CipherBlock`.

To illustrate the use of this function: the following functions are AES ECB decryption and encryption implementations base on `RT_AES_Decrypt()` and `RT_AES_Encrypt()`.

```
void aes_ecb_decry_test(
        IN  puchar pCipter,
        IN  puint32 pCipterLen,
        OUT puchar pPlain,
        INOUT puint32 pPlainLen)
{
    uint32 index = 0;
    uint8  Key[AES_BLOCK_SIZES] =
{0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0
x88};
    uint32 iPlainBlkLen = AES_BLOCK_SIZES;

   printf_high("AES_DecryTest\n");

    /*
     * 1. Check the input parameters
     *    - CipherTextLength must be divided with no remainder by block
     */
    if ((*pCipterLen % AES_BLOCK_SIZES) != 0) {
        printf_high("aes_ecb_decry_test: cipher text length is %d bytes, it
can't be divided with no remainder by block size(%d).\n",
            *pCipterLen, AES_BLOCK_SIZES);
        return;
    }

    if (*pPlainLen != *pCipterLen)  {
        printf_high("aes_ecb_decry_test: cipher text length is %d bytes,
```

```
should smae as .\n",
            *pCipterLen, AES_BLOCK_SIZES);
        return;
    }

    /*
     * 2. Main algorithm
     *     - Cypher text divide into serveral blocks (16 bytes/block)
     *     - Execute RT_AES_Decrypt procedure.
     */
    for (index=0; index<(*pCipterLen/AES_BLOCK_SIZES); index++) {
        RT_AES_Decrypt(
            pCipter + (index*AES_BLOCK_SIZES),
            AES_BLOCK_SIZES,
            Key,
            AES_BLOCK_SIZES,
            pPlain + (index*AES_BLOCK_SIZES),
            &iPlainBlkLen);
    }
}



void aes_ecb_encry_test(
        IN  puchar pPlain,
        IN  puint32 pPlainLen,
        OUT puchar pCipter,
        INOUT puint32 pCipterLen)
{
    uint32 index = 0;
    uint32 iBlockCount = 0;
    uint32 PaddingSize=0, PlainBlockEnd=0;

    uint8  Key[AES_BLOCK_SIZES] =
{0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0
x88};
    uint32 iCipterBlkLen = AES_BLOCK_SIZES;

    uint8  Block[AES_BLOCK_SIZES];
    uint32 BlockSize = 0;

    printf_high("aes_ecb_encry_test\n");

    if ((pPlain == NULL) || (pCipter == NULL)) {
        printf_high("aes_ecb_encry_test invalid data.\n");
        return;
    }

    if (*pPlainLen % ((uint32)AES_BLOCK_SIZES) > 0) {
        PaddingSize = ((uint32) AES_BLOCK_SIZES) - ( *pPlainLen %
((uint32)AES_BLOCK_SIZES));
        if (*pCipterLen < (*pPlainLen + PaddingSize)) {
            printf_high("aes_ecb_encry_test: cipher text length is %d bytes <
(plain text length %d bytes + padding size %d bytes).\n",
                  *pCipterLen, *pPlainLen, PaddingSize);
            return;
        } /* End of if */
    }
```

```
    iBlockCount   = (*pPlainLen + AES_BLOCK_SIZES - 1)/AES_BLOCK_SIZES;
    printf_high("iBlockCount = %d \n",iBlockCount);

    for (index=0; index<iBlockCount; index++) {
        PlainBlockEnd += AES_BLOCK_SIZES;
        BlockSize = AES_BLOCK_SIZES;

        if (PlainBlockEnd > *pPlainLen) {
            /*Set Padding value*/
            memset(Block, 0 , sizeof(Block));
            BlockSize = *pPlainLen%AES_BLOCK_SIZES;
        }
        memcpy(Block, pPlain+(index*AES_BLOCK_SIZES), BlockSize);

        RT_AES_Encrypt(
            Block,
            AES_BLOCK_SIZES,
            Key,
            AES_BLOCK_SIZES,
            pCipter + (index*AES_BLOCK_SIZES),
            &iCipterBlkLen);
    }

    return;
}
```

5.2. MD5

The following functions provide for various MD5 message-digest algorithm features.

5.2.1. RT_MD5

This function generates the MD5 message digest for a specified message.

Syntax	RT_MD5(Message[], MessageLen, DigestMessage[])			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	Message	const uint8	Message context
	IN	MessageLen	uint32	The length of message in bytes
	OUT	DigestMessage	uint8*	Digest message
Return Value	(void) returns no values			

To illustrate the use of this function: in the following code RT_MD5() calls RT_MD5_Init(), RT_MD5_Append() and RT_MD5_End() to calculate an MD5 digest.

```
#define MD5_BLOCK_SIZE    64 /* 512 bits = 64 bytes */
#define MD5_DIGEST_SIZE   16 /* 128 bits = 16 bytes */
typedef struct {
    uint32 HashValue[4];
    uint64 MessageLen;
    uint8  Block[MD5_BLOCK_SIZE];
    uint32   BlockLen;
} MD5_CTX_STRUC, *PMD5_CTX_STRUC;

void __romtext RT_MD5 (
    const uint8 Message[],
    uint32 MessageLen,
    uint8 DigestMessage[])
{
    MD5_CTX_STRUC md5_ctx;

    NdisZeroMemory(&md5_ctx, sizeof(MD5_CTX_STRUC));
    RT_MD5_Init(&md5_ctx);
    RT_MD5_Append(&md5_ctx, Message, MessageLen);
    RT_MD5_End(&md5_ctx, DigestMessage);
} /* End of RT_MD5 */
```

5.2.2. RT_MD5_Init

This function initializes an Md5_CTX_STRUC.

Syntax	RT_MD5_Init(pMD5_CTX)			
Parameters	Mode	Name	Type	Description
	OUT	pMD5_CTX	MD5_CTX_STRUC*	Pointer to Md5_CTX_STRUC
Return Value	(void) returns no values			

5.2.3. RT_MD5_Append

This function appends a message to a block. If block size > 64 Bytes, MD5_Hash is called.

Syntax	RT_MD5_Append(pMD5_CTX, Message[], MessageLen)			
Parameters	Mode	Name	Type	Description
	IN	pMD5_CTX	MD5_CTX_STRUC	Pointer to Md5_CTX_STRUC
	IN	Message	const uint8	Message context
	IN	MessageLen	uint32	The length of message in bytes
Return Value	(void) returns no values			

5.2.4. RT_MD5_End

This function performs the following actions:

1) appends bit 1 to the end of the message.
2) appends the length of message in rightmost 64 bits.
3) transforms the Hash Value to digest message.

Syntax	RT_MD5_End(pMD5_CTX, DigestMessage[])			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	pMD5_CTX	MD5_CTX_STRUC	Pointer to MD5_CTX_STRUC
	OUT	digestMessage	uint8*	Digest message
Return Value	(void) returns no values			

To illustrate the use of this function: below is a HMAC MD5 implementation that uses the MD5 APIs.

```
/* HMAC using MD5 hash function */
void RT_HMAC_MD5(
    const uint8 Key[],    // secret key
    uint32 KeyLen,          // the length of the key in bytes
    const uint8 Message[],// message context
    uint32 MessageLen,      // the length of message in bytes
    uint8 MAC[],          // message authentication code
    uint32 MACLen           // length of message authentication code
)
{
    MD5_CTX_STRUC md5_ctx1;
    MD5_CTX_STRUC md5_ctx2;
    uint8 K0[MD5_BLOCK_SIZE];
    uint8 Digest[MD5_DIGEST_SIZE];
    uint32 index;

    NdisZeroMemory(&md5_ctx1, sizeof(MD5_CTX_STRUC));
    NdisZeroMemory(&md5_ctx2, sizeof(MD5_CTX_STRUC));
    /*
     * If the length of K = B(Block size): K0 = K.
     * If the length of K > B: hash K to obtain an L byte string,
     * then append (B-L) zeros to create a B-byte string K0 (i.e., K0 = H(K)
|| 00...00).
     * If the length of K < B: append zeros to the end of K to create a B-
byte string K0
     */
    NdisZeroMemory(K0, MD5_BLOCK_SIZE);
    if (KeyLen <= MD5_BLOCK_SIZE) {
        NdisMoveMemory(K0, Key, KeyLen);
    } else {
        RT_MD5(Key, KeyLen, K0);
    }

    /* Exclusive-Or K0 with ipad */
    /* ipad: Inner pad; the byte x¡¦36¡¦ repeated B times. */
    for (index = 0; index < MD5_BLOCK_SIZE; index++)
        K0[index] ^= 0x36;
        /* End of for */

    RT_MD5_Init(&md5_ctx1);
```



```
    /* H(K0^ipad) */
    RT_MD5_Append(&md5_ctx1, K0, sizeof(K0));
    /* H((K0^ipad)||text) */
    RT_MD5_Append(&md5_ctx1, Message, MessageLen);
    RT_MD5_End(&md5_ctx1, Digest);

    /* Exclusive-Or K0 with opad and remove ipad */
    /* opad: Outer pad; the byte x¡¦5c¡¦ repeated B times. */
    for (index = 0; index < MD5_BLOCK_SIZE; index++)
        K0[index] ^= 0x36^0x5c;
        /* End of for */

    RT_MD5_Init(&md5_ctx2);
    /* H(K0^opad) */
    RT_MD5_Append(&md5_ctx2, K0, sizeof(K0));
    /* H( (K0^opad) || H((K0^ipad)||text) ) */
    RT_MD5_Append(&md5_ctx2, Digest, MD5_DIGEST_SIZE);
    RT_MD5_End(&md5_ctx2, Digest);

    if (MACLen > MD5_DIGEST_SIZE)
        NdisMoveMemory(MAC, Digest, MD5_DIGEST_SIZE);
    else
        NdisMoveMemory(MAC, Digest, MACLen);
} /* End of RT_HMAC_SHA256 */
```

5.3. PMK

MT7681 provides a function to determine the password hash for Pairwise Master Key, the shared secret key used in the IEEE 802.11i-2004 protocol.

5.3.1. RtmpPasswordHash

This function is used to calculate the PMK (pre-master key). The PMK will be used in the 4 way handshake state, between the station and an AP-router, to generate PTK (pairwise key) and GTK (group key) for encryption and decryption of transmitted and received packets

Syntax	RtmpPasswordHash(password, ssid, ssid_len, output)			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	password	pchar	ASCII string up to 63 characters in length
	IN	ssid	puchar	octect string up to 32 octects
	IN	ssid_len	int32	length of SSID in octects
	OUT	output	puchar	must be 40 octects in length and 0~32 octects (256 bits) is the key
Return Value	(int32) returning values: • none			

This function runs the hash algorithm several times; it takes about 6 seconds to run. So, if SSID and password are fixed in your application, you can calculate PMK offline to improve performance.

To illustrate the use of this function: in the following example a PMK is generated using an SSID and password.

```
/* This function will be called in iot_ap_startup()*/
void iot_ap_pmk_set(void)
{
    switch(pIoTApCfg->MBSSID.AuthMode)  {
        /*OPEN Mode*/
        case Ndis802_11AuthModeOpen:
            pIoTApCfg->MBSSID.WepStatus = Ndis802_11EncryptionDisabled;
            break;
        /*WPA mode, GTK is TKIP (KeyID=1), PTK is TKIP */
        case Ndis802_11AuthModeWPAPSK:
            pIoTApCfg->MBSSID.WepStatus = Ndis802_11Encryption2Enabled;
            break;
        /*WPA2 mode,  GTK is AES (KeyID=1), PTK is AES */
        case Ndis802_11AuthModeWPA2PSK:
            pIoTApCfg->MBSSID.WepStatus = Ndis802_11Encryption3Enabled;
            break;
        /*Mixed mode, GTK is TKIP (KeyID=1), PTK is AES */
        case Ndis802_11AuthModeWPA1PSKWPA2PSK:
            pIoTApCfg->MBSSID.WepStatus = Ndis802_11Encryption4Enabled;
            break;
        default:
            pIoTApCfg->MBSSID.AuthMode  = Ndis802_11AuthModeOpen;
            pIoTApCfg->MBSSID.WepStatus = Ndis802_11EncryptionDisabled;
            break;
    }
    pIoTApCfg->MBSSID.GroupKeyWepStatus  =  pIoTApCfg->MBSSID.WepStatus;

    if (pIoTApCfg->MBSSID.AuthMode >= Ndis802_11AuthModeWPA)  {
        /*Deriver PMK by AP 's SSID and Password*/
        uint8 keyMaterial[40] = {0};

        printf_high("PMK Updating ...\n");
        RtmpPasswordHash((pchar)pIoTApCfg->MBSSID.Passphase,
                         pIoTApCfg->MBSSID.Ssid,
                         ( int32)pIoTApCfg->MBSSID.SsidLen, keyMaterial);
        memcpy(pIoTApCfg->MBSSID.PMK, keyMaterial, LEN_PMK);
    }
}
```

5.4. CRC 16/32

MT7681 provides a cyclic redundancy check (CRC), a type of hash function used to produce a checksum - which is a small, fixed number of bits - against a block of data, such as a packet of network traffic or a block of a computer file. .Implementations of CRC16 and CRC32 are provided in a source code - `crypt_crc.c`.

There are 3 algorithms for CRC16 (ported from CCITT) and 1 algorithm for CRC32 (ported from libatc.

5.4.1. crc_cal_by_bit

This function provides CRC16 algorithm -1: calculate CRC by a bit. This algorithm can be used, but `crc_cal_by_byte` is the recommended option.

Syntax	`int16 crc_cal_by_bit(ptr, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`ptr`	`const unsigned char`	Pointer to the data on which the CRC calculation is performed
	IN	`len`	`unsigned char`	The length of data in bytes
Return Value	`(unsigned int16)` returning the calculated CRC value (16bit)			

The recommended polynomial for CRC-CCITT is 0x1021, for more information see the Wikipedia article Cyclic Redundancy Check.

5.4.2. crc_cal_by_byte

This function provides CRC16 algorithm-2: calculate CRC by Byte. This algorithm is the recommended option for the CRC calculations on MT7681 as it's the fastest of the three algorithms.

Syntax	`int16 crc_cal_by_byte(ptr, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`ptr`	`unsigned char`	Pointer to the data on which the CRC calculation is performed
	IN	`len`	`unsigned char`	The length of data in bytes
Return Value	`(unsigned int16)` returns the calculated CRC value (16bit).			

The recommended polynomial for CRC-CCITT is 0x1021, for more information see the Wikipedia article Cyclic Redundancy Check.

5.4.3. crc_cal_by_halfbyte

This function provides CRC16 algorithm-3: calculate CRC by half a byte. This algorithm can be used, but `crc_cal_by_byte` is the recommended option.

Syntax	`crc_cal_by_halfbyte(ptr, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`ptr`	`unsigned char`	Pointer to the data on which the CRC calculation is performed
	IN	`len`	`unsigned char`	The length of data in bytes
Return Value	(`unsigned int16`) returns the calculated CRC value (16bit).			

The recommended polynomial for CRC-CCITT is 0x1021, for more information see the Wikipedia article Cyclic Redundancy Check.

5.4.4. crc32

This function provides the CRC32 algorithm: calculates CRC to 32-bits.

Syntax	`crc32(ptr, len)`			
Parameters	**Mode**	**Name**	**Type**	**Description**
	IN	`ptr`	`unsigned char`	Pointer to the data on which the CRC calculation is performed
	IN	`len`	`unsigned short`	The length of data in bytes
Return Value	(`unsigned int`) returns the calculated CRC value (32 bit).			

The recommended polynomial for CRC32 is 0x04C11DB7, for more information see the Wikipedia article Cyclic Redundancy Check.

To illustrate the use of the CRC functions: the example below calculates the CRC values of the same input using each of the CRC functions.

```
void iot_cust_init(void)
{
    /* run customer initial function */
    uint8 INPUT[32] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
                       17,18,19,20,21,22,23,24,25,26,27,28,29,30,31};

    /*CRC-CCITT*/
    printf_high("[ByBit]----The CRC16 value=[0x%x] \n",
crc_cal_by_bit(INPUT, sizeof(INPUT)));
    printf_high("[ByByte]---The CRC16 value=[0x%x] \n",
crc_cal_by_byte(INPUT, sizeof(INPUT)));
    printf_high("[ByHalfByte]The CRC16 value=[0x%x] \n",
crc_cal_by_halfbyte(INPUT, sizeof(INPUT)));

    /*CRC-32*/
    printf_high("[ByBit]The CRC32 value=[0x%x] \n",
crc32(INPUT, sizeof(INPUT)));
}
```

the resulting output on the UART display will be the following:

```
==> RecoveryMode
<== RecoveryMode
(-)
SM=0, Sub=0
[ByBit]----The CRC16 value=[0xd2ff]
[ByByte]---The CRC16 value=[0xd2ff]
[ByHalfByte]The CRC16 value=[0xd2ff]
[ByBit]The CRC32 value= [0x91267e8a]
SM=1, Sub=0
[WTask]682487
```