M88WI6800-K SDK
User Manual

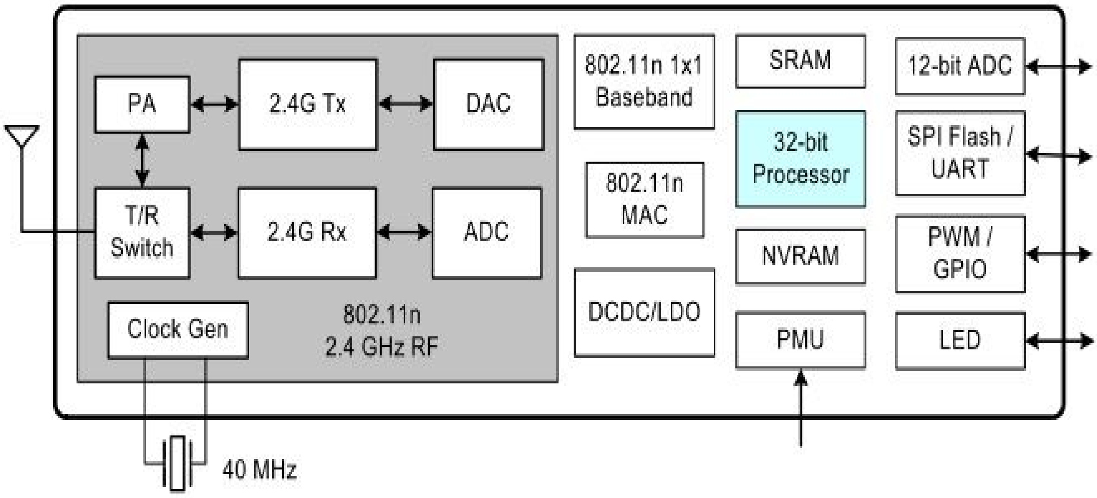
<table>
<tr><th>Reversion</th><th>Description</th><th>Status</th><th>Date</th></tr>
<tr><td>0.1</td><td>Initial version</td><td></td><td>20160321</td></tr>
<tr><td>0.2</td><td>Add firmware upgrade utility and SDK API</td><td></td><td>20160523</td></tr>
<tr><td>0.3</td><td>Update firmware utitliy and wireless API</td><td></td><td>20161212</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="4"><b><i>This document contains confidential and proprietary information that belongs to Montage. Using any of the information contained herein or imaging all or part of this document by any means is strictly forbidden without express written consent of Montage</i></b></td></tr>
</table>

1. M88WI6800 Introduction

The purpose of this document is to describe the usage of SDK and demonstrate how to build your code in M88WI6800 SDK.

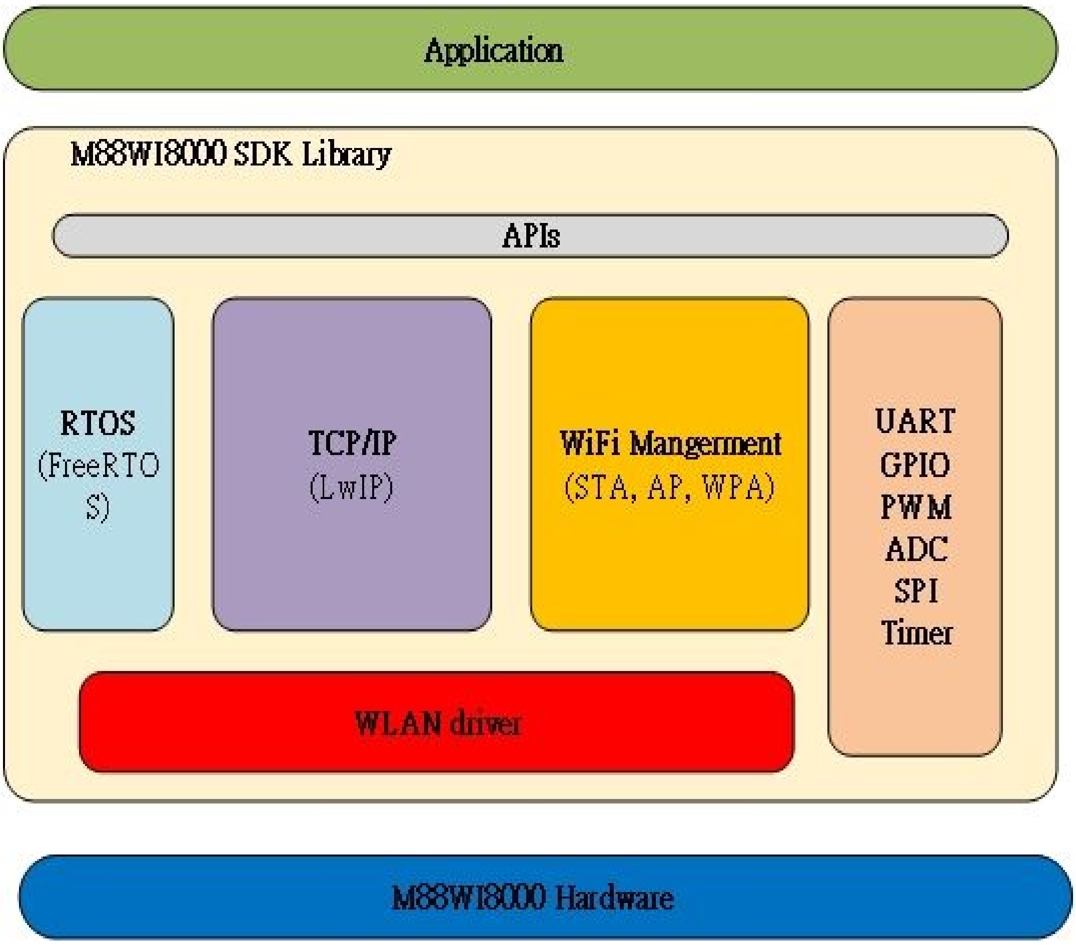
1.1 Overview

M88WI6800 is a System-on-Chip with Wi-Fi micro processor with ADC/PWM/GPIOs/UART interfaces. The chip is designed to operate in 2.4 GHz frequency and full complies with IEEE 802.11b/g/n standard based on 1T1R technology. It integrates a 32-bits high performance 32 bits micro-processor with over 200 MIPS and 320KB embedded RAM on which all application programs excuted. It also integrated with a 32 KHz low-speed clock and power manage unit to operate on low power state. It is an ideal solution for network enabled applications, such as internet of things. A block diagram illustrating the components of M88WI6800 is shown in Figure1.



1.2 Architecture

To simplify configuring connectivity of M88WI6800 chip, the SDK provides WLAN static library of station mode or access point mode. Real-time OS and TCPIP(LwIP) are also included in SDK library to easily achieve multi-tasking and networking application. To control the sensor or device M88WI6800 is incorporated into, PWM, GPIO, ADC and UART APIs are provided to easily use. M88WI6800 SDK software architechture is shown as follows.



1.3 Memory Map

Memory teyp	End address	Start address	Size
ROM Library	0x0007_FFFF	0x0006_8000	96KB
Reserved	0x0006_7FFF	0x0006_0000	32KB
DMA SRAM	0x0005_FFFF	0x0005_0000	64KB
SRAM	0x0004_FFFF	0x0000_0000	320KB

- Program can not run directly from serial flash.
- All program runs from SRAM or ROM.
- DMA SRAM is reserved for Hardware.

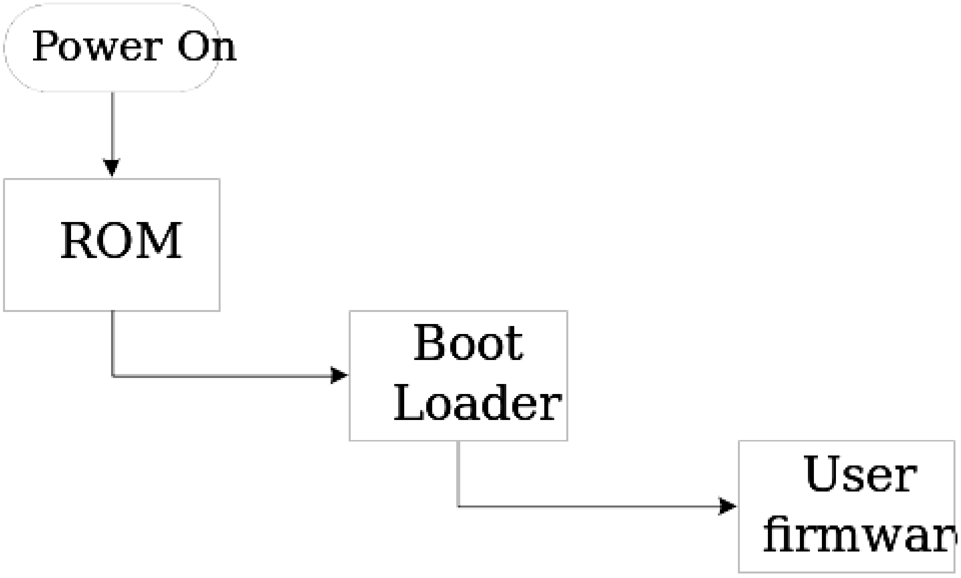
1.4 Serial Flash Layout

Offset	Section	Size	Description
0x0	Bootloader	64 KBytes	Do RF calibration and load Primary firmware
0x10000	Config	64 KBytes	Store configuration
0x20000	User Firmware	256 KBytes	Primary firmware location
0x80000	OTA Firmware	256 KBytes	OTA firmware location
0xE0000	TBD		User define

- Recommended size 1MB or more.
- Support maximum size to 16MB.

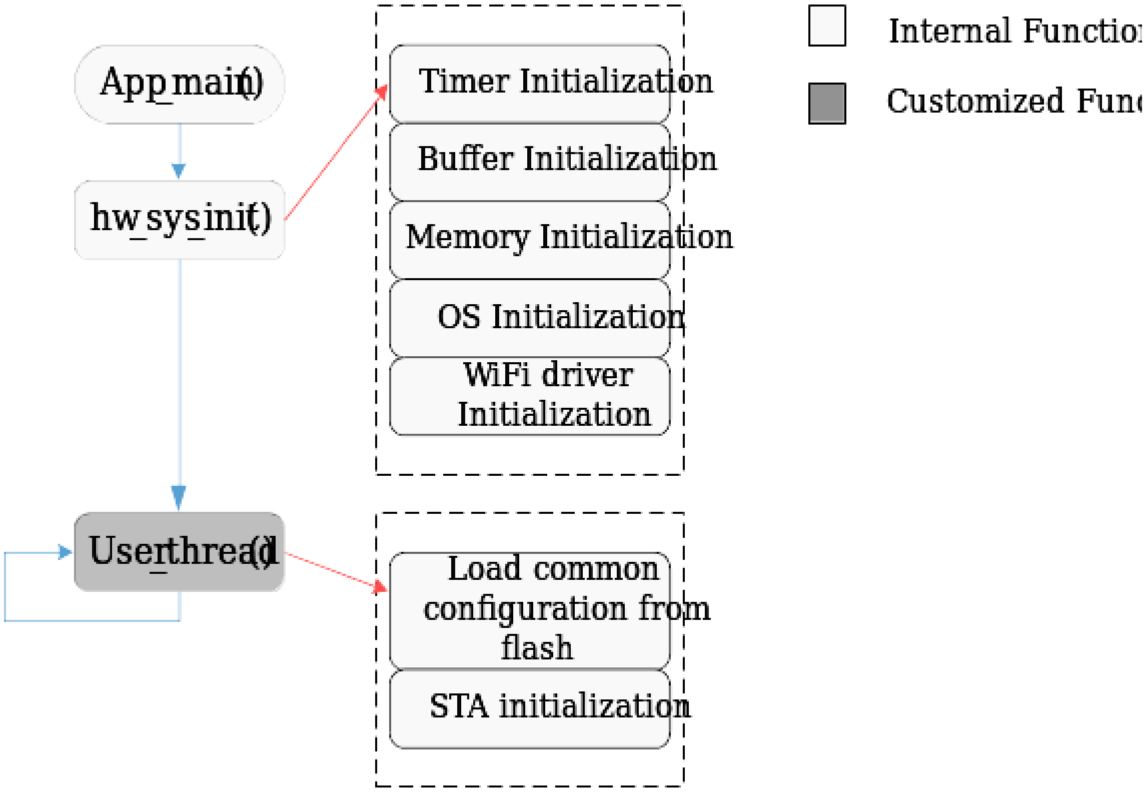
1.5 Firmware boot up sequence

M88WI6800 powers on and load bootloader image from serial flash to embedded RAM and execute. “bootloader” image checks integrity of user firmware on flash. If firmware is not exist or broken, it enters firmware upgrading state and wait for upgrade command from UART. Once firmware on flash passed verification of bootloader, bootloader loads firmware to embedded RAM and execute.



1.6 User Firmware flowchart

app_main() is called once application start-up. Users can place structure or global parameter initialization here. It's recommended that users' software initialization is done before entering while loop in **user_thread()**. **sys_msleep()** is invoked to make user thread to sleep for some milliseconds. The all functions in while loop will be executed again once user thread is wake up. It is not recommend add any customized code into app_main function. User_thread is entry of the user task. We expect user code should be located on this function.



Key features of M88WI6800 SDK:

- BA22 Toolchain
- Static libraries for APIs
- Firmware upgrade tool
- Sample source codes
- Wi-Fi station or access point(AP) working mode
- Support open, share, WPA-PSK, WPA2-PSK, WAPI authentications
- Support WEP64/128, TKIP, CCMP, SMS4
- AP mode supports up to 8 stations
- Support concurrent station/AP mode on one device
- Support PS-nopoll, PS-Poll, and UAPSD power saving mechanisms
- Hidden SSID
- Embedded TCP/IP protocal stack supports IPv4, UDP, TCP, ICMP, ARP
- Support DHCP client and server
- Support DNS client
- Support HTTP server
- Access profile from flash
- Multiple task management
- Hardware PWM APIs
- GPIO APIs
- Software I2C master function
- Chip Power management
- Sample codes

2. Developement Environment

The section provides a guide to generate firmware image from SDK.

2.1 Preparing the build environment

2.1.1 Installing M88WI6800 SDK for Linux

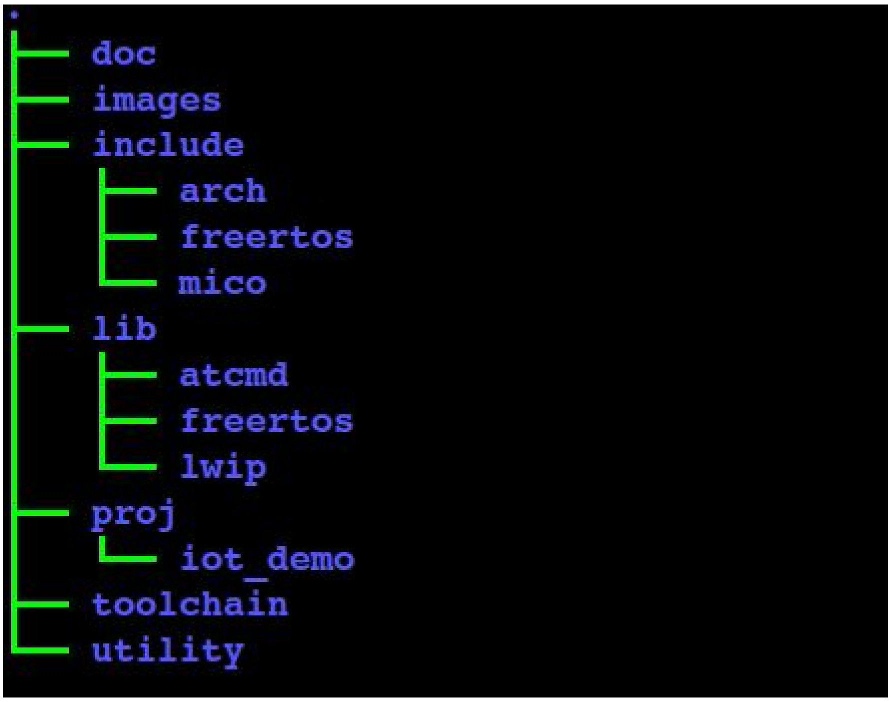
M88WI6800 SDK for Linux requires Ubuntu Linux. Any version can be used, however the 14.04 LTS 32-bit is recommended. If your Linux kernel is 64-bit, you need to install ia32-libs to support SDK toolchain.

“sudo apt-get install build-essential ia32-libs”

Step to install development environment:

1. Extract M88WI6800 SDK for Linux to your desired directory.
   “tar xzfv WI6800_sdk.tgz”
2. Locate the tool chain file ba-elf_4.7.3.tgz, and extract its content to SDK_path/toolchain folder.
   “tar xzfv ba-elf_4.7.3.tgz”

2.1.2 Directory Structure



```
.
├── doc
├── images
├── include
│   ├── arch
│   ├── freertos
│   └── mico
├── lib
│   ├── atcmd
│   ├── freertos
│   └── lwip
├── proj
│   └── iot_demo
├── toolchain
└── utility
```

- “doc” directory : the SDK related documents
- “images” directory : boot_loader, firmware binaries
- “include” directory : SDK header files
- “lib” directory : the library files for SDK
- “proj” directory : the example codes. User can create new project name under this folder. “iot_demo” is default example code of SDK.
- “toolchain” directory : BA2 tool chain should be extracted and placed at here.
- “utility” directory : checksum utility.

2.2 Building project

The default example is iot_demo project which under “proj” folder. Enter the root directory of SDK.

“make clean-iot_demo” → clean object code of proj/iot_demo
“make iot_demo” → build iot_demo.img

After building, output files will be generated on images diectory.

If you create new project folder under “proj”, for example, naming “user_test”. Just type “make user_test” to build “user_test.img” firmware image.

2.3 Burning image into flash

2.3.1 Set up the environment in Windows

1. **Unzip package files**

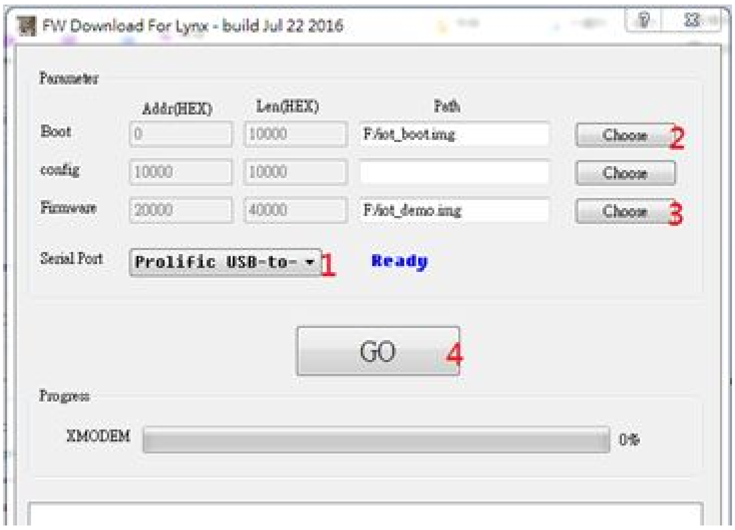
   unzip firmware_utility_tan_windows.zip ForWindows.zip

2. **Run the application**

   serial_tan.exe

2.3.2 Operations

After generating the bootocde or firmware image, users can choose the images for burning them into the flash. Please follow the following steps.



FW Download For Lynx - build Jul 22 2016

Parameter

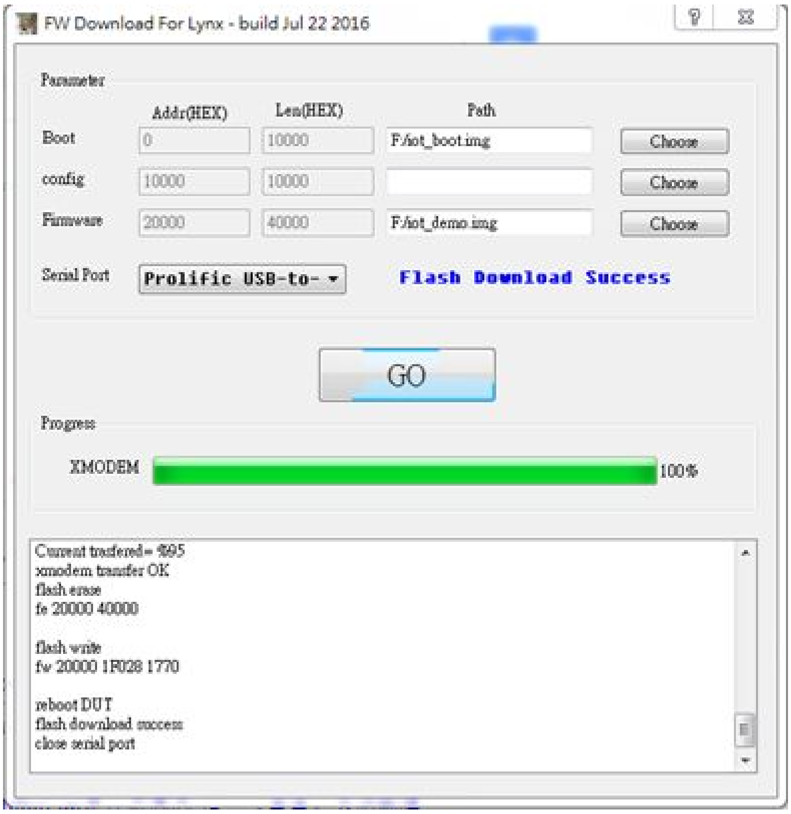
	Addr(HEX)	Len(HEX)	Path
Boot	0	10000	F:/iot_boot.img
config	10000	10000	
Firmware	20000	40000	F:/iot_demo.img

Serial Port: Prolific USB-to- Ready

Progress

XMODEM 0%

1. Select the correct serial port.
2. Choose boot.img if need.
3. Choose Firmware iot_demo.img
4. Press GO button
5. Flash download success will show as below



	Addr(HEX)	Len(HEX)	Path
Boot	0	10000	F:/iot_boot.img
config	10000	10000	
Firmware	20000	40000	F:/iot_demo.img

Serial Port: Prolific USB-to-

Flash Download Success

XMODEM 100%

```
Current trasfered= 5995
xmodem transfer OK
flash erase
fe 20000 40000

flash write
fw 20000 1F028 1770

reboot DUT
flash download success
close serial port
```

3. Programming Guide

3.1 Firmware bootup sequence

The firmware of M88WI6800 should locate on serial flash. After chip power on, M88WI6800 will load bootloader and firmware from serial flash to execute. The detail sequence is shown in as follows:

1. M88WI6800 powers on and load bootloader image from serial flash to embedded RAM and execute.
2. “bootloader” image checks integrity of user firmware on flash. If firmware is not exist or broken, it enters firmware upgrading state and wait for upgrade command from UART.
3. Once firmware on flash passed verification of bootloader, bootloader loads firmware to embedded RAM and execute.
4. User fimware should execute first function, app_main which located on app_init.c file. The app_main calls hw_sys_init function to initialize wlan hardware of M88WI6800 and spawns an user task finally. To make sure system bring up

smoothly, it is not recommend add any customized code into app_main function.

5. User_thread is entry of the user task. We expect user code should be located on this function.

3.2 Debug

M88WI6800 provides two hardware UARTs and system uses UART1 to output debug messages by default. User can utilize serial_printf() to debug program. Default baudrate is 115200bps.

3.3 Sample code

User_thread

void user_thread(void *arg) is the default method which provides users to add functions like network initialization, Wi-Fi parameters setting, and other initializations in the interface.

3.3.1 Connecting M88WI6800 to a station

Example usage:

```
// Users can load, modify, and write back Wi-Fi configurations before invoking
// wlan_start() to start Wi-Fi. Users can invoke Wi-Fi related APIs to make
// Wi-Fi interfaces to receive and transmit data. Without user configurations,
// M88WI6800 uses default configurations if Wi-Fi is brought up.

// Setup STA configurations
memset(&wNetConfig, 0x0, sizeof(wNetConfig));
// Wlan ssid string
memcpy(wNetConfig.wifi_ssid, "Demo_AP", sizeof(wNetConfig.wifi_ssid));
//  WEP key length:         ASCII=5(64 bits) or 13(128 bits)
//                          HEX=10(64 bits) or 26(128 bits)
//  WPA/WPA2 key length:    ASCII=8-63, HEX=64

// Wlan key string or hex data
strcpy((char *)wNetConfig.wifi_key, "12345678");
// Station mode
wNetConfig.wifi_mode = STATION;
// Fetch Ip address from DHCP server
wNetConfig.dhcp_mode = DHCP_CLIENT;

strcpy((char *)wNetConfig.local_ip_addr, "192.168.0.105");
strcpy((char *)wNetConfig.net_mask, "255.255.255.0");
strcpy((char *)wNetConfig.gateway_ip_addr, "192.168.0.1");
strcpy((char *)wNetConfig.dnssvr_ip_addr, "8.8.8.8");
// Retry interval after a failure connection
wNetConfig.wifi_retry_interval = 100;
wlan_set_reconnect(1);
wlan_set_myaddr(STATION, my_bssid);
```

```
// Connect Now!
wlan_start(&wNetConfig);
```

3.3.2 Configuring M88WI6800 as an AP

Example usage:

```
// Users can load, modify, and write back Wi-Fi configurations before invoking
// wla_set_opmode() to start Wi-Fi. Users can invoke Wi-Fi related APIs to make
// Wi-Fi interfaces to receive and transmit data. Without user configurations,
// M88WI6800 uses default configurations if Wi-Fi is brought up.

#if 0
{
        // Setup channel number.
       wlan_set_channel(11);
       // Setup AP configurations
       memset(&wNetConfig, 0x0, sizeof(wNetConfig));
       // Wlan ssid string
       memcpy(wNetConfig.wifi_ssid, "Demo_AP1", sizeof(wNetConfig.wifi_ssid));
       //  WEP key length:         ASCII=5(64 bits) or 13(128 bits)
       //                          HEX=10(64 bits) or 26(128 bits)
       //  WPA/WPA2 key length:    ASCII=8-63, HEX=64

       // Wlan key string or hex data
       strcpy((char *)wNetConfig.wifi_key, "12345678");
       // AP mode
       wNetConfig.wifi_mode = SOFT_AP;
       // Start DHCP server
       wNetConfig.dhcp_mode = DHCP_SERVER;

       strcpy((char *)wNetConfig.local_ip_addr, "192.168.169.1");
       strcpy((char *)wNetConfig.net_mask, "255.255.255.0");
       strcpy((char *)wNetConfig.gateway_ip_addr, "192.168.169.1");
       strcpy((char *)wNetConfig.dnssvr_ip_addr, "8.8.8.8");
       // Retry interval after a failure connection
       wNetConfig.wifi_retry_interval = 100;
       wlan_set_myaddr(SOFT_AP, my_bssid);
       // Connect Now!
       wlan_start(&wNetConfig);
}
#endif
```

3.3.3 Create a new thread

```
// Create a new thread.
void demo_thread(void *param)
{
    :
}

void user_thread(void *arg)
{
    :
    // "demo"           ->name of thread.
    // demo_thread      ->Pointer to function to run.
    // NULL             ->Argument passed into function.
    // 2048             ->Required stack amount in bytes.
    // 5                ->Thread priority.
    sys_thread_new("demo", demo_thread, NULL, 2048, 5);
    :
}
```

4. SDK API

Detailed Description

Timer API functions

void* add_timeout (void(*)(void *) *timer_func*, void * *func_parm*, unsigned int *msec*)

The function register a callback function on software timer list. Once the specific time is up, the callback function will be invoked.

Parameters:

timer_func	Pointer to callback function.
func_parm	Parameter of callback function.
msec	Milliseconds to count down.

Returns:

None.

int arc4random (void)

Random seed generator.

Parameters:

None.	

Returns:

A random seed.

void del_timeout (void(*)(void *) *timer_func*, void * *func_parm*)

The function is used to delete a previously timer, registered on software timer. Note that the timer_func and func_parm must have same value with **add_timeout()**.

Parameters:

timer_func	Pointer to callback function.
func_parm	Parameter of callback function.

Returns:

None.

void get_random_bytes (void * *buf*, unsigned int *len*)

Generate random byte in specific array.

Parameters:

buf	Pointer to input array.

len	Array size.

Returns:

None.

void hw_timer_start (unsigned int *us*, void(*)(void) *func*, int *autoload*)

Start hardware timer1 to countdown. When time is up, callback function is invoked and the timer is reloaded according autoload flag.

Parameters:

us	Microsecond to timeout.
func	Pointer to callback function.
autoload	Autoload flag.

Returns:

None.

void hw_timer_stop (void)

Stop hardware timer

Parameters:

None.	

Returns:

None.

int micros (void)

Get system time in microsecond.

Parameters:

None.	

Returns:

Microsecond.

int millis (void)

Get system time in millisecond.

Parameters:

None.	

Returns:

Millisecond.

void udelay (unsigned int *us*)

Delay specific microseconds.

Parameters:

us	Microseconds to wait.

Returns:

None.

Detailed Description

Memory allocation API functions

void free (void * *mem*)

The free function causes the space pointed to by mem to be deallocated, that is, made available for further allocation.

Parameters:

mem	Pointer to a previously allocated region of memory to be freed.

Returns:

None.

void* malloc (size_t *size*)

The malloc function allocates space for an object whose size is specified by size and whose value is indeterminate.

Parameters:

size	Size, in bytes, of the region to allocate.

Returns:

NULL is returned if the space could not be allocated. Otherwise, a pointer to a region of the requested size is returned.

Detailed Description

wireless API functions

int wlan_add_notification (notify_types *type*, void * *functionAddress*)

The function register notification and it's callback function.

Parameters:

type	system defined notifications.

functionAddress	callback function.

Returns:

NO_ERR.

int wlan_del_notification (notify_types *type*)

The function unregister notification and it's callback function.

Parameters:

type	system defined notifications.

Returns:

NO_ERR.

int wlan_del_notification_all (notify_types *type*)

The function unregister all notification and callback functions.

Parameters:

type	system defined notifications.

Returns:

NO_ERR.

int wlan_disable_powersave (void)

Disable IEEE power save mode.

Parameters:

None.	

Returns:

NO_ERR: succeed
WLAN_ERR_GENERAL: failed

void wlan_drv_init (void)

The function initials Wi-Fi driver with LWIP.

Parameters:

None.	

Returns:

None.

int wlan_enable_powersave (void)

Enable IEEE power save mode.

When this function is enabled, Wlan enter IEEE power save mode if Wlan is in station mode and has connected to an AP, and do not need any other control from application. To save more power, use mcu_powersave_config.

Parameters:

None.	

Returns:

NO_ERR: succeed
WLAN_ERR_GENERAL: failed

int wlan_get_hidden_ssid (void)

Get Wi-Fi network hidden ssid status (ap mode only).

Parameters:

type	Specifies wlan interface.

Returns:

hidden ssid status.

int wlan_get_ifs_sm (wlan_if_types *type*)

The function gets Wi-Fi connection status.

Parameters:

type	Specifies wlan interface.

Returns:

-1: failed
1 0: STATE_IDLE
2 1: STATE_SCAN
3 2: STATE_SCAN_DONE
4 3: STATE_LINK_UP
5 4: STATE_LINK_DOWN

int wlan_get_link_sts (link_sts * *sts*, wlan_if_types *type*)

Read current wireless link status.

Parameters:

sts	Point to the buffer to store the link status.

Returns:

NO_ERR.

char* wlan_get_myaddr (wlan_if_types *type*)

Get Wi-Fi network MAC address.

Parameters:

type	Specifies wlan interface.

Returns:

Interface MAC address.

int wlan_get_reconnect (void)

The function gets reconnect policy in STA mode.

Parameters:

None.	

Returns:

1: enabled
0: disabled

int wlan_get_sta_info (int(*)(const char *) *get_mac_callback*)

The function gets Wi-Fi station information (ap mode only).

Parameters:

get_mac_callback	The callback function.

Returns:

0: succeed
-1: failed

int wlan_get_sta_num (wlan_if_types *type*)

The function gets Wi-Fi station number.

Parameters:

type	Specifies wlan interface.

Returns:

-1: failed
otherwise: station number

void wlan_init (void)

The function initialize Wi-Fi basic settings.

Parameters:

None.	

Returns:

None.

int wlan_init_notification (void)

The function initializes notification center.

Parameters:

None.	

Returns:

NO_ERR.

void wlan_led_install (void)

The function starts to service Wi-Fi LED callback function.

Parameters:

None.	

Returns:

None.

void wlan_led_uninstall (void)

The function stop to service Wi-Fi LED callback function.

Parameters:

None.	

Returns:

None.

int wlan_monitor_rx_type (filter_rx_types *type*)

Set which wifi packet will be captured. RX all wifi packet if didn't call this function. This function can be called more than once to set RX different type packet.

Parameters:

type	Capture packet type.

Returns:

NO_ERR.

int wlan_power_off (void)

Close the RF chip's power supply, all network connection is lost.

Parameters:

None.	

Returns:

NO_ERR: succeed
WLAN_ERR_GENERAL: failed

int wlan_power_on (void)

Open the RF's power supply and do some necessary initialization.

Note:

The default RF state is powered on after **wlan_init**, so this function is not needed after wlan_init.

Parameters:

None.	

Returns:

NO_ERR: succeed
WLAN_ERR_GENERAL: failed

void wlan_register_monitor_cb (monitor_cb_t *fn*)

Set the callback function to RX the captured wifi packet.

Parameters:

fn	Callback function.

Returns:

None.

void wlan_scan_result_to_buffer (void)

The function dumps information of all APs into apps buffer without output to serial port.

Parameters:

None.	

Returns:

None.

int wlan_set_ch_bandwith (int *channel*, int *ht40*)

Set the monitor channel and bandwidth. Valid channel is 1~13. Set ht40 as 0 for 20M-Hz ,ht40 as 1 for 40M-Hz.

Parameters:

channel	Monitor channel.
ht40	Bandwidth is 40M-Hz.

Returns:

NO_ERR.

int wlan_set_channel (int *channel*)

Set the monitor channel. Valid channel is 1~13.

Parameters:

channel	Monitor channel.

Returns:

NO_ERR.

void wlan_set_hidden_ssid (int *en*)

Set Wi-Fi network hidden ssid status (ap mode only).

Parameters:

en	Hidden ssid or not.

Returns:

None.

void wlan_set_myaddr (wlan_if_types *type*, char * *myaddr*)

Set Wi-Fi network MAC address.

Parameters:

type	Specifies wlan interface.
myaddr	Interface MAC address.

Returns:

None.

int wlan_set_phy (int *phy*)

The function sets Wi-Fi physical mode. It checks range from 0 to 7 (bit[2:0]=ngb) and restarts Wi-Fi if it is running.

Parameters:

phy	The physical mode.

Returns:

0: succeed
-1: failed
-2: busy

void wlan_set_reconnect (int *en*)

The function sets reconnect policy in STA mode. If the policy is enabled, the STA will reconnect to AP once it's disconnected.

Parameters:

en	1: enable 0: disable

Returns:

None.

int wlan_set_txpwr (int *level*)

The function sets TX power level. It checks range from 0 to 12 and restarts Wi-Fi if it is running.

Parameters:

level	The TX power level.

Returns:

0: succeed
-1: failed

int wlan_start (network_info * *net*)

Connect or establish a Wi-Fi network in normal mode (station or soft ap mode).

This function can establish a Wi-Fi connection as a station or create a soft AP that other stations can connect (4 stations Max). In station mode, Wlan first scan all of the supported Wi-Fi channels to find a wlan that matches the input SSID, and read the security mode. Then try to connect to the target wlan. If any error occurs in the connection procedure or disconnected after a successful connection, Wlan start the reconnection procedure in background after a time interval defined in inNetworkInitPara. Call this function twice when setup coexistence mode (station + soft ap). This function returns immediately in station mode, and the connection will be executed in background.

Parameters:

net	Specifies wlan parameters.

Returns:

In station mode, always return WLAN_NO_ERR. In soft ap mode, return WLANXXXERR

int wlan_start_adv (network_info_adv * *net*)

Connect to a Wi-Fi network with advantage settings (station mode only).

This function can connect to an access point with precise settings, that greatly speed up the connection if the input settings are correct and fixed. If this fast connection is failed for some reason, Wlan change back to normal: scan + connect mode refer to **wlan_start**. This function returns after the fast connection try.

Note:

This function cannot establish a soft ap, use **wlan_start()** for this purpose. If input SSID length is 0, Wlan use BSSID to connect the target wlan. If both SSID and BSSID are all wrong, the connection will be failed.

Parameters:

net	Specifies the precise wlan parameters.

Returns:

Always return WLAN_NO_ERR although error occurs in first fast try. Return WLAN_ERR_TIMEOUT if DHCP client timeout.

int wlan_start_monitor (void)

Start wifi monitor.

Parameters:

None.	

Returns:

NO_ERR.

void wlan_start_scan (void)

Start a wlan scanning in 2.4GHz in background.

Once the scan is completed, Wlan sends a notify: NOTIFY_WIFI_SCAN_COMPLETED, with callback function: void (*function)(scan_result *pApList, Context_t * const inContext). Register callback function using add_notification() before scan.

Parameters:

None.	

Returns:

None.

int wlan_stop_monitor (void)

Stop wifi monitor.

Parameters:

None.	

Returns:

NO_ERR.

int wlan_suspend (void)

Close all the Wi-Fi connections, station mode and soft ap mode.

Note:

This function also stop the background retry mechanism started by **wlan_start()** and **wlan_start_adv()**.

Parameters:

None.	

Returns:

NO_ERR.

int wlan_suspend_station (void)

Close the connection in station mode.

Note:

This function also stop the background retry mechanism started by **wlan_start()** and **wlan_start_adv()**.

Parameters:

None.	

Returns:

NO_ERR.

Detailed Description

net API functions.

int net_add_notification (int *type*, void * *functionAddress*)

The function register notification and it's callback function.

Parameters:

type	system defined notifications.
functionAddress	callback function.

Returns:

NO_ERR.

int net_del_notification (int *type*)

The function unregister notification and it's callback function.

Parameters:

type	system defined notifications.

Returns:

NO_ERR.

int net_del_notification_all (int *type*)

The function unregister all notification and callback functions.

Parameters:

type	system defined notifications.

Returns:

NO_ERR.

void net_drv_init (void)

The function initials LWIP device and all of netifs name.

Parameters:

None.	

Returns:

None.

int net_get_client_info (const char * *mac*)

The function gets client information.

Parameters:

mac	The client's address.

Returns:

err_t error code

char* net_get_dns (int *idx*, unsigned int * *ipaddr*)

The function gets DNS server's IP address.

Parameters:

idx	The DNS server index.
ipaddr	IP address.

Returns:

IP string of DNS server.

int net_get_hostbyname (const char * *name*, char * *ipaddr*)

The function queries host IP address by hostname. It also dumps host's IP address.

Parameters:

name	The hostname.
ipaddr	The host IP address.

Returns:

1: succeed
0: failed

char* net_get_name (int *idx*)

The function gets netif name.

Parameters:

idx	The netif index.

Returns:

Name string of netif.

void net_if_down (int *idx*)

The function sets interface down.

Parameters:

idx	The netif index.

Returns:

None.

void net_if_ip_sts (void * *data*, int *type*)

The function reads current IP status on a network interface.

Parameters:

data	Point to the buffer to store the IP address.
type	Specifies wlan interface.

Returns:

None.

void net_if_up (int *idx*, unsigned char *dhcp*, char * *mac*, unsigned char * *_ip*, unsigned char * *_mask*, unsigned char * *_gw*, unsigned char * *_dns*)

The function sets interface up and network configurations.

Parameters:

idx	The netif index.
dhcp	DHCP mode.
mac	MAC address.
_ip	IP address.
_mask	Net mask.
_gw	Gateway address.
_dns	DNS server address.

Returns:

None.

int net_init_notification (void)

The function initializes notification center.

Parameters:

None.	

Returns:

NO_ERR.

void net_ping (unsigned int *dip*, unsigned int * *size*, unsigned int * *iter*, unsigned int * *to*, unsigned int * *interval*)

The function ping destination address.

Parameters:

dip	Destination address.
size	Packet size.
iter	Iteration.
to	Timeout(seconds).
interval	Packet interval(milliseconds).

Returns:

None.

void net_set_dns (int *idx*, unsigned int * *ipaddr*)

The function sets DNS server by server index.

Parameters:

idx	The DNS server index.
ipaddr	IP address.

Returns:

None.

Detailed Description

GPIO API functions

void gpio_enable (int *pin*, int *mode*)

Enable GPIO function.

Parameters:

pin	GPIO number
mode	0: disable 1: enable

Returns:

None

void pin_mode (int *pin*, int *mode*)

Set GPIO pin mode, include gpio_enable.

Parameters:

pin	GPIO number
mode	0: input 1: output

Returns:

None

int digital_read (int *pin*)

Read GPIO pin input data, call it after pin_mode.

Parameters:

pin	GPIO number

Returns:

0: low
1: high

void digital_write (int *pin*, int *val*)

Set GPIO pin output data, call it after pin_mode.

Parameters:

pin	GPIO number
val	0: low 1: high

Returns:

None

void digital_write_two (int *pin*, int *val*, int *pin2*, int *val2*)

Set two GPIO pin output data at the same time, call it after pin_mode.

Parameters:

pin	GPIO number
val	0: low 1: high
pin2	GPIO number
val2	0: low 1: high

Returns:

None

void pin_dis_intr (int *pin*, int *mode*)

Disable GPIO pin interrupt.

Parameters:

pin	GPIO number
mode	0: rising 1: falling 2: high level 3: low level

Returns:

None

void pin_en_intr (int *pin*, int *mode*)

Enable GPIO pin interrupt.

Parameters:

pin	GPIO number
mode	0: rising 1: falling 2: high level 3: low level

Returns:

None

Detailed Description

I2C API functions. More details in appendix 5.4.

int i2c_read_byte (unsigned char *slave_addr*)

I2C master read data(1 byte).

Parameters:

slave_addr	I2C slave address

Returns:

data

void i2c_read_data (unsigned char *slave_addr*, char * *str*, int *len*)

I2C master read data.

Parameters:

slave_addr	I2C slave address
len	data length, master must know the correct length of data

Returns:

None

int i2c_send_byte (unsigned char *slave_addr*, unsigned char *byte*)

I2C master send data(1 byte).

Parameters:

slave_addr	I2C slave address
byte	data

Returns:

0: ack

int i2c_send_data (unsigned char *slave_addr*, char * *data,* int *len*)

I2C master send data.

Parameters:

slave_addr	I2C slave address
**data*	data pointer
len	data length

Returns:

0: ack

int i2c_send_str (unsigned char *slave_addr*, char * *str*)

I2C master send string.

Parameters:

slave_addr	I2C slave address
**str*	data pointer

Returns:

0: ack

Detailed Description

MADC API functions

int analog_read (int *pin*)

MADC read digital data.

Parameters:

pin	select madc chan 0: CH I 1: CH Q

Returns:

digital data(0 ~ 4095)

Detailed Description

PWM API functions

void pwm_set_enable (int *pwm_ch*, int *value*)

Set PWM enable

channel 1 use the same pin as PWM0,1 (GPIO6,7)

channel 2 use the same pin as PWM2,3 (GPIO8,9)

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)
value	0: disable 1: enable

Returns:

None

void pwm_set_freq (int *pwm_ch*, int *id*)

Set PWM frequency

PWM0 and PWM1 share the same frequency setting. PWM1's frequency will be changed, if user changes PWM0's frequency. (PWM2 and PWM 3 as well). More details of PWM register setting in appendix 5.3.

id:

0: 0.3 Hz	1: 0.5	2: 10	3: 25	4: 45	5: 90
6: 160	7: 250.4	8: 500.8	9: 600.96	10: 849.18	11: 1k
12: 1.502k	13: 2.056k	14: 3k	15: 3.906k	16: 5k	17: 8k
18: 10k	19: 12.5k	20: 15k	21: 20.16k	22: 25k	23: 31.25k
24: 50k	25: 62.5k	26: 125k	27: 250k	28: 312.5k	29: 625k
30: 1250k					

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)
id	0 ~ 30

Returns:

None

int pwm_get_freq (int *pwm_ch*)

Get PWM frequency

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)

Returns:

frequency id, -1: can't find correct id

void pwm_set_duty (int *pwm_ch*, int *duty*)

Set PWM duty

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)
duty	0 ~ 31; 16 = 50%, 31 = 100%

Returns:

None

int pwm_get_duty (int *pwm_ch*)

Get PWM duty

Parameters:

pwm_ch	0 ~ 3(GPIO 6 ~ 9)

Returns:

duty 0 ~ 31

void pwm_set_polarity (int *pwm_ch*, int *value*)

Set PWM polarity

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)
value	0: active low

	1: active high

Returns:

None

int pwm_get_polarity (int *pwm_ch*)

Get PWM polarity

Parameters:

pwm_ch	0 ~ 3 (GPIO 6 ~ 9)

Returns:

value
0: active low
1: active high

Detailed Description

Configuration API functions

int config_submit (void)

Config data all burn into flash memory

Returns:

1: success
0: error

int config_get (sdk_param **param*)

Config get data

Parameters:

**param*	get data pointer

Returns:

1: success
0: error

int config_load (void)

Config load data to memory, initialize config read/write data process. Call config_load before read/write config data.

Returns:

1: success
0: error

int config_set (sdk_param *param)

Config set data

Parameters:

**param*	set data pointer

Returns:

1: success

0: error

Detailed Description

Serial API functions

void serial_conf (int *br_id*, int *parity*, int *stopbits*, int *chan*)

Uart configuration

Parameters:

br_id	baudrate table index 0 ~ 12 0: 1200 1: 2400 2: 4800 3: 9600 4: 19200 5: 38400 6: 57600 7: 115200 8: 230400 9: 460800 10: 500000 11: 576000 12: 921600 13: 1000000 14: 1152000 15: 1500000
parity	0: none 1: odd 2:even
stopbits	1, 2 bit
chan	uart chan 0 ~ 2

Returns:

None

int serial_init (int *chan*)

Serial initial

chan 0(UART1), chan 1-2(UART2), initial tx buffer for transparent mode.

Parameters:

chan	uart channel 0 ~ 2

Returns:

1: success
0: error

int serial_read_byte (int *mode*, int *chan*, char * *buf*, int *len*, char *end_c*)

Read serial data

Parameters:

mode	0: read one byte 1: read bytes 2: read bytes until terminator character
chan	uart channel 0 ~ 2
**buf*	read buffer pointer
len	data length
end_c	terminator character

Returns:

mode 0: return the first byte of incoming serial data (-1 means no data available)

mode 1, 2 : return data length (0 means no valid data was found)

int serial_write (int *chan*, char * *pdata*, int *datalen*)

Copy data to tx buffer and insert to fifo

Need to initial txbuf, call serial_init(chan) first

Parameters:

chan	uart channel 0 ~ 2
pdata	rx buffer
datalen	data length

Returns:

stat

0: done

1: busy, tx buffer is full

-1: fail, tx buffer is null

int uart_no_wait_putc (int *chan*, int *c*)

Uart tx put character no wait

Parameters:

chan	uart channal 0 ~ 2
c	character data

Returns:

0: tx fifo not full
-1: tx fifo full

void uart_set_timeout (unsigned int *set_timeout*)

Set the maximum milliseconds to wait for serial data

Parameters:

set_timeout	time(ms)

Returns:

None

int uart_timeout_getc (int *chan*)

Uart get character, wait until time out

Parameters:

chan	uart channel 0 ~ 2

Returns:

character, -1 means no data available

5. Appendix

5.1 PWM Frequency Formula

	Pre-scaler	T_b	T_a
bit	29 - 22	19 - 17	16 - 14
0xc0010	PWM0 & PWM1	PWM1	PWM0
0xc0014	PWM2 & PWM3	PWM3	PWM2

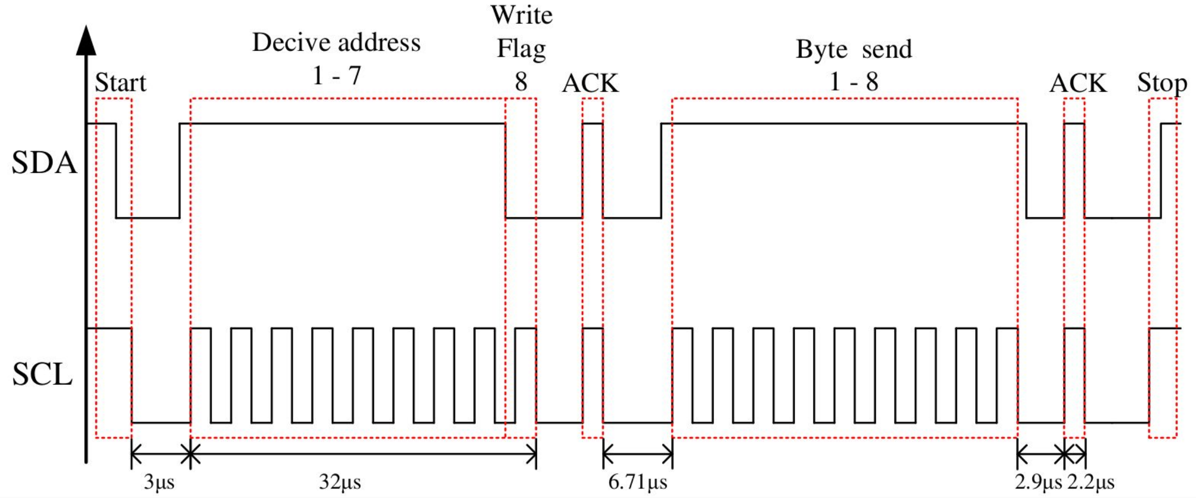
$$Period_n(\text{ms}) = \frac{1}{pwm\_clock} \times \frac{256}{1000} \times (Pre\text{-}scaler + 1) \times tick\_max$$
$$= \frac{1}{pwm\_clock} \times \frac{256}{1000} \times (Pre\text{-}scaler + 1) \times (T_n \times 1000 + 12000)$$

$$\text{if } T_n = 0 \Rightarrow Period_n(\text{ms}) = \frac{1}{pwm\_clock} \times \frac{256}{1000} \times (Pre\text{-}scaler + 1)$$
$$\text{if } T_n = 7 \Rightarrow Period_n(\text{ms}) = \frac{1}{pwm\_clock} \times \frac{256}{1000} \times (Pre\text{-}scaler + 1) \times 32$$
$$n = a, b$$

5.3 I2C Master R/W Transfer Timing Diagram

Lynx I^{2}C bus is emulated using two GPIO pins (GPIO 17 and GPIO 18). One pin is for clock signals (SCL), and one pin is for data signals (SDA).

a. Write Transfer Sequence



b. Read Transfer Sequence