RAK439 移植手册

深圳市瑞科慧联技术有限公司 www.rakwireless.com

邮箱：info@rakwireless.com

目录

1. 硬件平台资源

MCU 需要的外设资源如下：

- 一个 SPI 接口
- 一个外部中断引脚
- 一个复位模块引脚
- 一个控制模块电源开关引脚（可选）
- 一个向上的计数器，用来处理模块内部驱动的超时

RAK439 驱动库不使用 OS 占用的 memory 如下：

- Flash 占用 35K 字节左右
- RAM
  全局变量、静态变量：740 字节
  堆：至少分配 4K 字节（1 个 RX buffer，1 个 socket buffer，4 个 scan buffer）

2. 驱动库参数

2.1 驱动库参数介绍

```
typedef struct
{
    bool  spi_int_enable;          // customer can choose enable or disenable spi int event
                                   // driver
    uint8_t rx_queue_num;          // rx buffer queue num >= 1
    uint8_t socket_max_num;        // module support socket numbers max 8
    uint8_t scan_max_num;          // scan result buffer numbers normal : 10 if you need more
                                   // can raise it
    uint8_t tcp_retry_num;         // tcp backoff retry numbers
    char*    host_name;            // module host name ,you can see it in router clients when
                                   // DHCP success
    char*    country_code;         // set module country code ,CN (1-13),JS(1-14),UP(1-11)
    struct   driver_cb_ driver_cb; // platform related driver used
    struct   app_cb_ app_cb;       // application related callback info
}rw_DriverParams_t;
```

参数说明：

- spi_int_enable：不使用 OS 时，设置 spi_int_enble 为 true，表示模块使能了外部中断检测，MCU 需要连接模块的中断引脚；设置 spi_int_enble 为 false，表示模块不使用外部中断检测，MCU 不需要连接模块的中断引脚。使用 OS 时，模块强制使用中断检测。

- rx_queue_num：RX buffer 的个数>=1，一个 RX buffer 的大小为 1664 字节，RAK439 加载驱动（执行 rw_sysDriverInit）的时候从堆里分配，卸载驱动（执行 rw_sysDriverDeinit）的时候释放。
- socket_max_num：socket 个数，最大 8 个，一个 socket buffer 的大小为 48 个字节，RAK439 加载驱动（执行 rw_sysDriverInit）的时候从堆里分配，卸载驱动（执行 rw_sysDriverDeinit）的时候释放。
- scan_max_num：扫描 buffer 的个数，最少 4 个，一个 scan buffer 的大小为 44 个字节，RAK439 在 STA 模式下，连接路由器（执行 rw_wlanConnect）的时候驱动里会从堆里分配内存，连接成功或失败后释放；用户程序扫描（执行 rw_wlanNetworkScan）的时候驱动会从堆里分配内存，但不会释放，所以用户需要在扫描回调函数里手动释放内存。
- tcp_retry_num：tcp 重传最大时间间隔为 $2^{tcp_retry_num+1}$s。
- host_name：DHCP 成功以后，在路由器客户端列表显示的主机名
- country_code：模块国家代码，CN (1-13)，JP(1-14)，US(1-11)
- driver_cb：平台相关接口
- app_cb：应用回调接口

2.2 平台相关接口

用户在应用程序里实现，供 RAK439 驱动调用

```
struct driver_cb_
{
    rw_HwInit_          hw_init;
    rw_HwInit_          hw_deinit;
    rw_PowerUpDown_     hw_power;
    rw_Malloc_          driver_malloc;
    rw_Free_           driver_free;
    rw_TimeDelay_      time_delay;
    rw_Stamp_           Stamp_get;
    rw_ToggleIrq_      toggle_irq;
    rw_SpiIoBuffer_     spi_io_buffer;
    rw_AssertFunc_     driver_assert;
};
```

2.2.1. hw_init

hw_init 类型 typedef uint32_t(*rw_HwInit_)(void)

返回

SPI 时钟

说明

硬件接口初始化——

初始化控制 RAK439 电源开关的引脚（可选）；
初始化复位 RAK439 的引脚；
初始化 SPI 接口；
初始化外部中断引脚。
此函数在 RAK439 加载驱动（执行 rw_sysDriverInit）的时候调用。

2.2.2. hw_deinit

```
hw_deinit   类型 typedef uint32_t(*rw_HwInit_)( void )
```

返回

无

说明

硬件接口反初始化——
关闭 SPI 时钟。
此函数在 RAK439 卸载驱动（执行 rw_sysDriverDeinit）或者复位驱动（执行 rw_sysDriverReset）的时候调用。

2.2.3. hw_power

```
hw_power   类型 typedef void(*rw_PowerUpDown_)( uint8_t status )
```

参数

[in] **status** 0：掉电；1：上电

返回

无

说明

RAK439 模块上电并复位，掉电并复位
上电并复位操作—RAK439 电源开关引脚置低（可选）；RAK439 复位引脚置高。
掉电并复位操作—RAK439 电源开关引脚置高（可选）；RAK439 复位引脚置低。

2.2.4. driver_malloc

```
driver_malloc   类型 typedef void*(*rw_Malloc_)( uint32_t size )
```

参数

[in] **size** 待分配内存大小

返回

指向分配内存的指针

说明

分配内存

2.2.5. driver_free

```
driver_free  类型 typedef void(*rw_Free_)( void* data )
```

参数

[in] **data** 指向待释放的内存

返回

无

说明

释放内存

2.2.6. time_delay

```
time_delay  类型 typedef void(*rw_TimeDelay_)( int ms )
```

参数

[in] **ms** 延时 ms 数

返回

无

说明

延时函数

2.2.7. Stamp_get

```
Stamp_get  类型 typedef rw_stamp_t(*rw_Stamp_)( void )
```

返回

rw_stamp_t --uint32_t

说明

获取当前 tick 数，单位 ms，RAK439 driver 会使用这个函数来判断超时。

2.2.8. toggle_irq

```
toggle_irq  类型 typedef void(*rw_ToggleIrq_)( uint8_t enable )
```

参数

[in] **enable** 1:使能外部中断 0:禁止外部中断

返回

无

说明

使能、禁能外部中断

2.2.9. spi_io_buffer

```
spi_io_buffer   类型 typedef void(*rw_SpiIoBuffer_)( uint8_t* write,
                                                     uint8_t* read,
                                                     uint16_t len )
```

参数

[in] **write** 写数据到 RAK439 模块
[out] **read** 从 RAK439 模块读数据
[in] **len** 读写数据的长度

返回

无

说明

MCU 与 RAK439 模块通过 SPI 接口传输数据

当 read = NULL，write != NULL 时，MCU 写数据到 RAK439；
当 read != NULL，write = NULL 时，MCU 从 RAK439 读数据；
当 read != NULL，write != NULL，MCU 读写数据。

2.2.10. driver_assert

```
driver_assert   类型 typedef void(*rw_AssertFunc_)( const char* file,
                                                    int line );
```

参数

[out] **file** 出现错误的文件
[out] **line** 错误所在的行

返回

无

说明

打印来自 wifi 驱动的错误信息

2.3 应用回调接口

RAK439 驱动返回给用户应用程序的回调事件

```
struct app_cb_
{
```

```
    rw_WlanConnEvent_   conn_cb;
    rw_WlanScan_        scan_cb;
    rw_WlanEasyWps_     easy_wps_cb;
    rw_IpDhcp_          dhcp_cb;
    rw_DnsResult_       dns_cb;
};
```

2.3.1. conn_cb

```
conn_cb   类型 typedef void(*rw_WlanConnEvent_)( uint8_t event,
                                                 rw_WlanConnect_t* wlan_info,
                                                 RW_DISCONNECT_REASON dis_reasoncode )
```

参数

[out]	**event**	回调事件类型	
		CONN_STATUS_STA_CONNECTED	成功连上路由
		CONN_STATUS_STA_DISCONNECT	与路由断开连接
		CONN_STATUS_AP_ESTABLISH	AP 建立成功
		CONN_STATUS_AP_CLT_CONNECTED	有客户端连到 AP
		CONN_STATUS_AP_CLT_DISCONNECT	客户端与 AP 断开
[out]	**wlan_info**	路由器信息，包括 bssid，信道，ssid，密码，加密方式等。	
[out]	**dis_reasoncode**	网络断开原因	

返回

无

说明

Wlan 连接事件回调

2.3.2. scan_cb

```
scan_cb   类型 typedef void(*rw_WlanScan_)( rw_WlanNetworkInfoList_t* scan_info )
```

参数

[out]	**scan_info**	扫描到的路由信息

返回

无

说明

扫描事件回调，在回调函数里处理完扫描信息以后要释放掉对应的内存。

2.3.3. easy_wps_cb

```
easy_wps_cb   类型 typedef void(*rw_WlanEasyWps_)( rw_WlanEasyConfigWpsResponse_t
                                                    *pResponse, int status)
```

参数

[out] **pResponse** 保存获取到的路由信息

[out] **status** easyconfig 或 WPS 是否成功

返回

无

说明

easyconfig 或 WPS 事件回调

2.3.4. dhcp_cb

```
dhcp_cb   类型 typedef void(*rw_IpDhcp_)( rw_IpConfig_t* addr,
                                          int status )
```

参数

[out] **addr** 保存获取到的 ip 信息

[out] **status** ip 是否成功获取

返回

无

说明

dhcp 连接事件回调

2.3.5. dns_cb

```
dns_cb   类型 typedef void(*rw_DnsResult_) ( int dnsIp )
```

参数

[out] **dnsIp** 域名解析得到的 ip 地址

返回

无

说明

dns 域名解析事件回调

3. 驱动库 OS 接口介绍

3.1 任务接口

3.1.1. rw_creat_task

```
void* rw_creat_task(RW_OS_TASK_PTR p_task)
```

参数

[in] **p_task** 函数指针，指向待创建的任务

返回

指针，指向任务的句柄

说明

任务创建函数

3.1.2. rw_del_task

```
int rw_del_task(void* p_tcb)
```

参数

[in] **p_tcb** 任务句柄

返回

RW_OS_OK

RW_OS_ERROR

说明

任务删除函数

3.2 互斥信号量接口

3.2.1. rw_creat_mutex

```
void* rw_creat_mutex(void)
```

参数

无

返回

指向所创建的互斥信号量的指针

说明

创建互斥信号量函数

3.2.2. rw_del_mutex

```
int rw_del_mutex(void* p_mutex)
```

参数

[in] **p_mutex** 指向互斥信号量的指针

返回

RW_OS_OK

RW_OS_ERROR

说明

删除互斥信号量函数

3.2.3. rw_lock_mutex

```
int rw_lock_mutex (  void* p_mutex,
                     uint32_t timeout
                  )
```

参数

[in] **p_mutex** 指向互斥信号量的指针

[in] **timeout** block 超时时间：0 为一直等待

返回

RW_OS_OK

RW_OS_ERROR

说明

互斥信号量上锁函数

3.2.4. rw_unlock_mutex

```
int rw_unlock_mutex(void* p_mutex)
```

参数

[in] **p_mutex** 指向互斥信号量的指针

返回

RW_OS_OK

RW_OS_ERROR

说明

互斥信号量解锁函数

3.3 信号量接口

3.3.1. rw_creat_sem

```
void* rw_creat_sem(void)
```

参数

无

返回

指向所创建信号量的指针

说明

创建信号量函数

3.3.2. rw_del_sem

```
int rw_del_sem(void* p_sem)
```

参数

[in] **p_sem** 指向信号量的指针

返回

RW_OS_OK

RW_OS_ERROR

说明

删除信号量函数

3.3.3. rw_post_sem

```
int rw_post_sem(void* p_sem)
```

参数

[in] **p_sem** 指向信号量的指针

返回

RW_OS_OK

RW_OS_ERROR

说明

释放信号量函数

3.3.4. rw_pend_sem

```
int rw_pend_sem ( void* p_sem,
                  uint32_t timeout
                )
```

参数

[in] **p_sem** 指向信号量的指针

[in] **timeout** block 超时时间：0 为一直等待

返回

RW_OS_OK

RW_OS_TIME_OUT

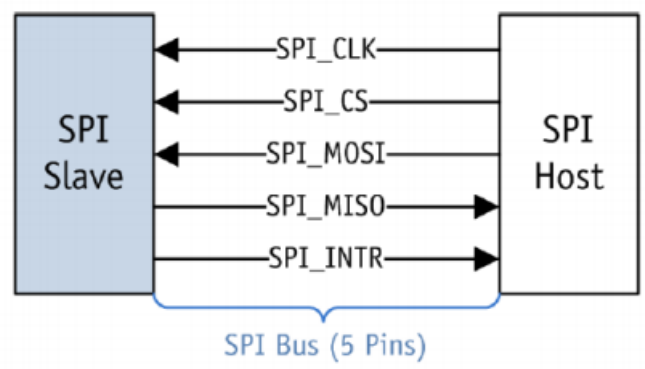
RW_OS_ERROR

说明

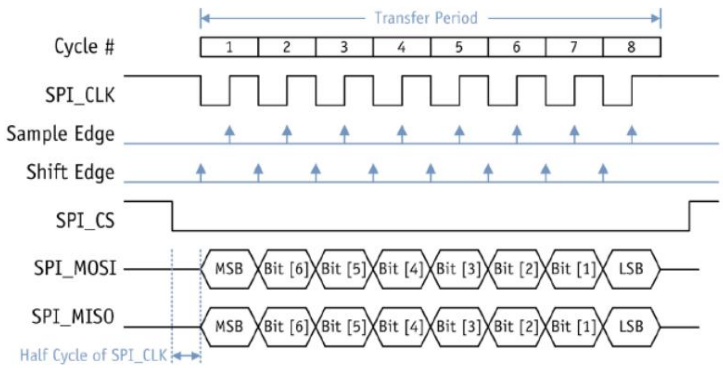
阻塞在信号量函数

4. RAK439 SPI 接口

SPI Host 与 RAK439 的 SPI 接口如下图所示：



SPI 时序如下图所示：



SPI 描述：

- SPI clock 最大 24Mhz。
- 模块为 SPI 从模式。
- SPI 传输的字节序为高位在先(MSB)。
- SPI clock 空闲电平为高，极性为 1（CPOL=1）；在 SPI clock 的上升沿采样数据(SPI_MOSI)，下降沿发送数据(SPI_MISO)，相位为 1（CHPA =1）。
- SPI 传输数据长度为 8 bit

执行 rw_sysDriverInit 时的 SPI 起始数据：

```
send=44 recv=b4
send=0 recv=b4
send=0 recv=b4
send=80 recv=b4
send=c2 recv=b4
send=0 recv=b4
send=0 recv=c
send=0 recv=5b
```

接收到 5b 表示 SPI 接口工作正常，SPI 接口接触不良或者供电不足则收不到 5b。

5. RAK439 外部中断

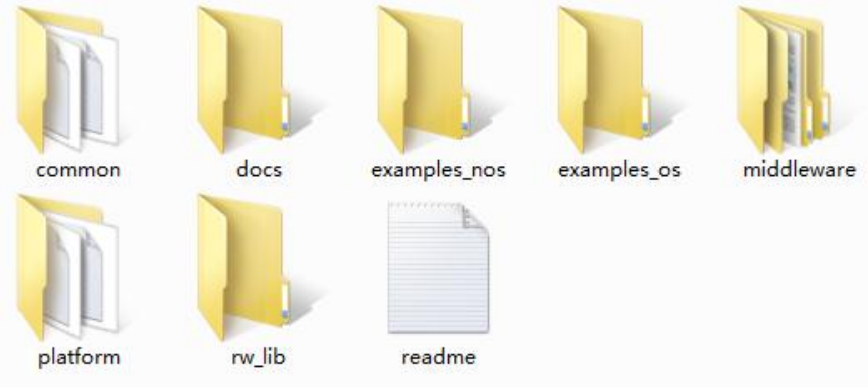
外部中断引脚设置为下降沿触发

6. STM32F4 平台移植示例

6.1 软件包介绍

本次移植使用的 MCU 为 stm32f411，使用的 stm32 库为 STM32F4xx_StdPeriph_Driver，版本 V1.5.0。

移植好的软件包为 RAK439_STM32F4xx_SDK_1_0_0，目录如下图所示：



/common　　该目录下的文件与平台无关。
/common/include　　该目录下是 error code 头文件、wifi 驱动库头文件、socket 头文件。
/common/mem　　该目录下是内存操作程序。
/common/rw_os　　该目录下是 wifi 驱动库 OS 接口程序。

/docs　　该目录下是相关软件文档。

/examples_nos　　该目录下是不带 OS 的示例程序，工程支持 KEIL 和 IAR。
/examples_os　　该目录下是带 OS 的示例程序，工程支持 KEIL 和 IAR。

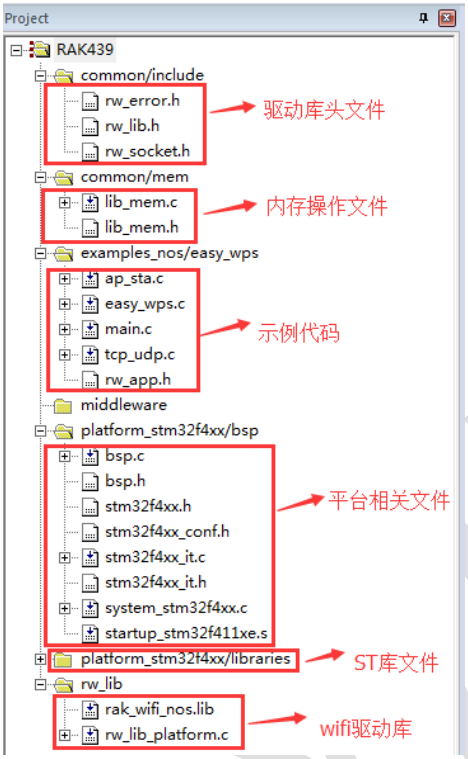
/middleware　　该目录下是一些独立的中间件代码，有 FreeRTOS，PolarSSL 等。

/platform/bsp　　该目录下是不带 OS、平台相关的文件。
/platform/bsp_os　　该目录下是带 OS、平台相关的文件。
/platform/ST　　该目录下是 ST 的标准库。
/platform/rw_lib_platform.c
　　该文件是不含 OS 时，wifi 驱动库和硬件平台之间的接口。
/platform/rw_lib_platform_os.c
　　该文件是含 OS 时，wifi 驱动库和硬件平台之间的接口。

/rw_lib　　该目录下是 wifi 驱动库文件。

6.2　不含 OS 移植步骤

下图是一个不带 OS 的工程示例：



rw_lib_platform.c 文件是 RAK439 驱动库和硬件平台之间的接口，所以是移植的关键文件。

void wifi_init_params(rw_DriverParams_t* params)——此函数负责初始化 RAK439 驱动参数，包括硬件接口函数、时间管理函数、内存管理函数、RAK439 回调函数和系统运行参数。

```
void wifi_init_params(rw_DriverParams_t* params)
{
    //硬件接口函数
    params->driver_cb.hw_init = _init_interface;          //初始化 GPIO，SPI，外部中断
    params->driver_cb.hw_deinit = _deinit_interface;      //SPI 反初始化—关闭 SPI 时钟
    params->driver_cb.hw_power = _power_up_down;          //上电、掉电
    params->driver_cb.spi_io_buffer = _spi_io_buffer;     //SPI 主机与 WiFi 模块传输数据
    params->driver_cb.toggle_irq = _ext_interrupt;        //外部中断开启、关闭

    //时间管理函数
    params->driver_cb.time_delay = delay_ms;              //延时函数
    params->driver_cb.Stamp_get = get_stamp;              //获取系统时间

    //内存管理函数
    params->driver_cb.driver_free = vPortFree;            //释放内存
    params->driver_cb.driver_malloc = pvPortMalloc;       //分配内存

    //wifi 回调函数
    params->driver_cb.driver_assert = customer_assert;    //错误打印回调
    params->app_cb.conn_cb = connect_callback;            //连接到路由回调事件
```

```
    params->app_cb.scan_cb = scan_callback;                 //扫描回调事件
    params->app_cb.dhcp_cb = ip_callback;                   //ip 获取回调事件
    params->app_cb.dns_cb = dns_ipcallback;                 //dns 回调事件
    params->app_cb.easy_wps_cb = wps_easy_callback;         //WPS、easyconfig 回调事件

    //系统运行参数设定
    params->spi_int_enable = false;       //外部中断使能/禁能设置，只对 NOS 有效
    params->rx_queue_num = 1;             //接收数据 buffer 个数
    params->scan_max_num = 10;            //scan buffer 个数
    params->tcp_retry_num = 5;            //tcp 重传设置
    params->socket_max_num = 8;           //socket buffer 个数
    params->country_code = "CN";          //模块国家代码，CN (1-13)，JP(1-14)，US(1-11)
    params->host_name = "rakmodule";      //DHCP 成功后，模块在路由器客户端列表显示的主机名
}
```

- 内存管理函数、wifi 回调函数、系统运行参数设定与硬件平台无关，用户可以使用例程默认的设置。
- 时间管理函数要根据不同的硬件平台来实现，本次移植使用的单片机 stm32f411 自带一个 24 位 systick，ST 的库里也有操作这个 systick 的 api，用户可以参考例程来移植。
- 硬件接口函数与硬件平台相关，也是本次移植最重要的一部分。下面将详细介绍各个硬件接口函数。

6.2.1. hw_init 实现

GPIO、SPI、外部中断初始化

```
static uint32_t _init_interface(void)
{
  WIFI_GPIO_Init();           //初始化 wifi 模块电源开关引脚(可选)、复位引脚、SPI 片选引脚
  WIFI_SPI_Init ();           //初始化 SPI 主机
  WIFI_INT_Init ();           //初始化与 wifi 模块相连的外部中断引脚
}
```

1. GPIO 管脚初始化——初始化 WiFi 模块电源开关引脚（可选），初始化复位引脚，初始化 SPI 片选引脚。

```
void WIFI_GPIO_Init(void)
{
  GPIO_InitTypeDef  GPIO_InitStructure;
  /* GPIOA GPIOB Peripheral clock enable */
  RCC_AHB1PeriphClockCmd(WIFI_PWD_GPIO_CLK|WIFI_CS_GPIO_CLK, ENABLE);   //使能 GPIO 时钟
  /*初始化 WiFi 模块电源开关引脚，可选*/
#if defined (USE_WIFI_POWER_FET)
```

```
  RCC_AHB1PeriphClockCmd(WIFI_FET_GPIO_CLK, ENABLE);
  GPIO_InitStructure.GPIO_Pin = WIFI_FET_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_Init(WIFI_FET_GPIO_PORT, &GPIO_InitStructure);
  GPIO_WriteBit(WIFI_FET_GPIO_PORT, WIFI_FET_PIN, Bit_SET);
#endif
  /*初始化复位引脚*/
  GPIO_InitStructure.GPIO_Pin = WIFI_PWD_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_Init(WIFI_PWD_GPIO_PORT, &GPIO_InitStructure);
  GPIO_WriteBit(WIFI_PWD_GPIO_PORT, WIFI_PWD_PIN, Bit_RESET);
  /*初始化 SPI 片选引脚*/
  GPIO_InitStructure.GPIO_Pin = WIFI_CS_PIN;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(WIFI_CS_GPIO_PORT, &GPIO_InitStructure);
  GPIO_WriteBit(WIFI_CS_GPIO_PORT, WIFI_CS_PIN, Bit_SET);
}
```

2. SPI 主机初始化

SPI 初始化：配置为 SPI 主机，全双工模式，CHPA=1 CPOL=1，8 位数据传输，高位在先。

```
void SPI1_Config(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  SPI_InitTypeDef  SPI_InitStructure;
  /*!< Enable the SPI clock */
  WIFI_SPI_CLK_INIT(WIFI_SPI_CLK, ENABLE);
  /*!< Enable GPIO clocks */
  RCC_AHB1PeriphClockCmd(WIFI_SPI_SCK_GPIO_CLK | WIFI_SPI_MISO_GPIO_CLK |
                         WIFI_SPI_MOSI_GPIO_CLK , ENABLE);
  /*!< SPI pins configuration *************************************************/
  /*!< Connect SPI pins to AF */
  GPIO_PinAFConfig(WIFI_SPI_SCK_GPIO_PORT, WIFI_SPI_SCK_SOURCE, WIFI_SPI_SCK_AF);
  GPIO_PinAFConfig(WIFI_SPI_MISO_GPIO_PORT, WIFI_SPI_MISO_SOURCE, WIFI_SPI_MISO_AF);
```

```
GPIO_PinAFConfig(WIFI_SPI_MOSI_GPIO_PORT, WIFI_SPI_MOSI_SOURCE, WIFI_SPI_MOSI_AF);
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
/*!< SPI SCK pin configuration */
GPIO_InitStructure.GPIO_Pin = WIFI_SPI_SCK_PIN;
GPIO_Init(WIFI_SPI_SCK_GPIO_PORT, &GPIO_InitStructure);
/*!< SPI MOSI pin configuration */
GPIO_InitStructure.GPIO_Pin =  WIFI_SPI_MOSI_PIN;
GPIO_Init(WIFI_SPI_MOSI_GPIO_PORT, &GPIO_InitStructure);
/*!< SPI MISO pin configuration */
GPIO_InitStructure.GPIO_Pin =  WIFI_SPI_MISO_PIN;
GPIO_Init(WIFI_SPI_MISO_GPIO_PORT, &GPIO_InitStructure);
/*!< SPI configuration */
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;     //全双工模式
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;                          //主机模式
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;                      //8 位数据传输
SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;                            //CPOL=1
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;                           //CPHA=1
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;    //96M/4
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;                     //高位在先
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_Init(WIFI_SPI, &SPI_InitStructure);
/*!< Enable the WIFI_SPI  */
SPI_Cmd(WIFI_SPI, ENABLE);}                                           //使能 SPI1
```

3. 外部中断引脚初始化

```
void WIFI_INT_Init (void)
{
  EXTI_InitTypeDef   EXTI_InitStructure;
  GPIO_InitTypeDef   GPIO_InitStructure;
  NVIC_InitTypeDef   NVIC_InitStructure;
  /* Enable GPIOB clock */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
  /* Configure PB0 pin as input floating */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Pin = WIFI_INT_PIN;
  GPIO_Init(WIFI_INT_GPIO_PORT, &GPIO_InitStructure);
  /* Enable SYSCFG clock */
```

```
  /* Connect EXTI Line0 to PB0 pin */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
  SYSCFG_EXTILineConfig(WIFI_INT_GPIO_EXTI_PORT, WIFI_INT_EXTI_PIN_SOURCE);
  /* Configure EXTI Line0 */
  EXTI_InitStructure.EXTI_Line = WIFI_INT_EXTI_LINE;
  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
  EXTI_InitStructure.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStructure);
  /* Enable and set EXTI Line0 Interrupt to the lowest priority */
  NVIC_InitStructure.NVIC_IRQChannel = WIFI_INT_EXTI_IRQN;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x00;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}
```

4. 外部中断服务程序

```
void EXTI0_IRQHandler(void)
{
  if(EXTI_GetITStatus(WIFI_INT_EXTI_LINE) != RESET)
  {
    /* Clear the EXTI line 0 pending bit */
    EXTI_ClearITPendingBit(WIFI_INT_EXTI_LINE);
    DRIVER_INT_HANDLE();              //此函数在 rak_wifi_nos.lib 里实现，用户不必关心
  }
}
```

6.2.2. spi_io_buffer 实现

SPI 主机与 RAK439 通过 SPI 传输数据函数实现

```
static void _spi_io_buffer(uint8_t* write, uint8_t* read, uint16_t len)
{
  uint32_t i;
  uint8_t dummy;
  uint8_t recv;
  GPIO_WriteBit(WIFI_CS_GPIO_PORT, WIFI_CS_PIN, Bit_RESET);//拉低 SPI 片选,选中 RAK439 模块
  if(read == NULL) {
    for(i=0;i<len;i++) {
      while((WIFI_SPI->SR&SPI_FLAG_TXE)==RESET);
```

```
      if(write == NULL) {
        WIFI_SPI->DR = dummy;
      }else {
        WIFI_SPI->DR = write[i];                    //write 不为空，向 RAK439 写数据
      }
      while((WIFI_SPI->SR&SPI_FLAG_RXNE)==RESET);
      recv = WIFI_SPI->DR;
    }
  }
  else {                                            //read 不为空，从 RAK439 读数据
    for(i=0;i<len;i++) {
      while((WIFI_SPI->SR&SPI_FLAG_TXE)==RESET);
      if(write == NULL) {
        WIFI_SPI->DR = dummy;
      }else {
        WIFI_SPI->DR = write[i];
      }
      while((WIFI_SPI->SR&SPI_FLAG_RXNE)==RESET);
      read[i] = WIFI_SPI->DR;
    }
  }
  GPIO_WriteBit(WIFI_CS_GPIO_PORT, WIFI_CS_PIN, Bit_SET); //拉高 SPI 片选线
}
```

6.2.3. toggle_irq 实现

使能、禁能外部中断函数实现

```
static void _ext_interrupt(uint8_t enable)
{
    if (enable){
        NVIC_EnableIRQ(WIFI_INT_EXTI_IRQN);
    }
    else{
        NVIC_DisableIRQ(WIFI_INT_EXTI_IRQN);
    }
}
```

6.2.4. hw_power 实现

掉电、上电函数实现

```
static void _power_up_down(uint8_t status)
{
   if (status) {
#if defined (USE_WIFI_POWER_FET)
     GPIO_WriteBit(WIFI_FET_GPIO_PORT, WIFI_FET_PIN, Bit_RESET);
#endif
     delay_ms(10);
     GPIO_WriteBit(WIFI_PWD_GPIO_PORT, WIFI_PWD_PIN, Bit_SET);
  } else {
#if defined (USE_WIFI_POWER_FET)
     GPIO_WriteBit(WIFI_FET_GPIO_PORT, WIFI_FET_PIN, Bit_SET);
#endi
     GPIO_WriteBit(WIFI_PWD_GPIO_PORT, WIFI_PWD_PIN, Bit_RESET);
  }
}
```

6.2.5. hw_deinit 实现

SPI 接口反初始化实现

```
static void _deinit_interface(void)
{
  WIFI_SPI_Deinit();
  return 0;
}
void WIFI_SPI_Deinit(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  /*!< Disable the WIFI_SPI  ********************************************/
  SPI_Cmd(WIFI_SPI, DISABLE);
  /*!< DeInitializes the WIFI_SPI ***************************************/
  SPI_I2S_DeInit(WIFI_SPI);
  /*!< WIFI_SPI Periph clock disable ************************************/
  WIFI_SPI_CLK_INIT(WIFI_SPI_CLK, DISABLE);
  /*!< Configure all pins used by the SPI as input floating *************/
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

  GPIO_InitStructure.GPIO_Pin = WIFI_SPI_SCK_PIN;
  GPIO_Init(WIFI_SPI_SCK_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = WIFI_SPI_MISO_PIN;
```

```
  GPIO_Init(WIFI_SPI_MISO_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = WIFI_SPI_MOSI_PIN;
  GPIO_Init(WIFI_SPI_MOSI_GPIO_PORT, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = WIFI_CS_PIN;
  GPIO_Init(WIFI_CS_GPIO_PORT, &GPIO_InitStructure);
}
```

6.3 不含 OS 例程介绍

/examples_nos 文件夹下的示例程序包括：

- AP & STA 联网
- 一键联网——easyconfig、WPS 联网
- Socket 通信——tcp server、tcp client、udp server、udp client

6.3.1. AP & STA 联网示例介绍：

1. STA 模式——运行此程序只需修改要连到的 AP 名称和密码

```
int rw_network_startSTA(void)
{
   ...
   conn.ssid = "Nescafe";              //修改为要连到的 AP 名称
   conn.psk = "1234567890";            //修改 AP 密码
   ...
}
```

串口打印信息：

```
main.c:32  Host platform init...success
226  main.c:43  rak wifi LibVersion:1.0.4-2.1.39                    //wifi 库版本
230  main.c:45  rak wifi module-MAC:60:C5:A8:60:03:79               //模块 mac 地址
2646  rw_lib_platform.c:23  connect_callback event = 0x0            //连接路由成功事件
2652  rw_lib_platform.c:28  ---------connected AP info list---------
2658  rw_lib_platform.c:29  bssid = 8C:21:0A:D8:1C:0C
2664  rw_lib_platform.c:35  channel =6
2668  rw_lib_platform.c:36  ssid =Nescafe
2672  rw_lib_platform.c:37  psk =1234567890
```

```
2677  rw_lib_platform.c:38  sec_mode =1
2681  rw_lib_platform.c:39  auth_mode =4
2685  rw_lib_platform.c:43  ---------CONN_STATUS_STA_CONNECTED--------
3641  rw_lib_platform.c:70  ipquery success addr = 0xc0a80a6a       //dhcp 成功事件
```

2. Soft AP 模式——程序默认会建立一个 RAK_AP 的热点，密码 1234567890

```
int rw_network_startAP(void)
{
   ...
   conn.ssid = "RAK_AP";                                    //AP 名称
   conn.psk = "1234567890";                                 //AP 密码
   ...
}
```

串口打印信息：

```
 main.c:32  Host platform init...success
226  main.c:43  rak wifi LibVersion:1.0.4-2.1.39                  //wifi 库版本
230  main.c:45  rak wifi module-MAC:60:C5:A8:60:03:79             //wifi 模块 mac 地址
3363  rw_lib_platform.c:23  connect_callback event = 0x2          //创建 AP 成功事件
3369  rw_lib_platform.c:51  ---------CONN_STATUS_AP_ESTABLISH--------
242418  rw_lib_platform.c:23  connect_callback event = 0x3        //客户端连接成功
242424  rw_lib_platform.c:54  ---------CONN_STATUS_AP_CLT_CONNECTED--------
```

电脑连接模块 soft ap 成功如下图所示：

6.3.2. 一键配置示例介绍：

模块可以通过手机实现 easyconfig 一键配置，也可以按下路由的 WPS 键进行配置。

1. easyconfig 一键配置

```
app_demo_ctx.easywps_mode = CONFIG_EASY;                    //easyconfig 模式
rw_network_startConfig(app_demo_ctx.easywps_mode);          //开启一键配置
```

串口打印信息：

```
 main.c:32  Host platform init...success
226  main.c:43  rak wifi LibVersion:1.0.4-2.1.39            //wifi 驱动库版本
230  main.c:45  rak wifi module-MAC:60:C5:A8:60:03:79       //wifi 模块 mac 地址
236  main.c:57  rw_network_startConfig ...                  //开始 easyconfig
41163  rw_lib_platform.c:110  bssid = 8C:21:0A:D8:1C:0C
41169  rw_lib_platform.c:116  channel =6
41173  rw_lib_platform.c:117  ssid =Nescafe
41178  rw_lib_platform.c:118  psk =1234567890
42921  rw_lib_platform.c:23  connect_callback event = 0x0     //连接路由成功事件
42926  rw_lib_platform.c:28  ---------connected AP info list---------
42933  rw_lib_platform.c:29  bssid = 8C:21:0A:D8:1C:0C
42939  rw_lib_platform.c:35  channel =6
42943  rw_lib_platform.c:36  ssid =Nescafe
42947  rw_lib_platform.c:37  psk =1234567890
42952  rw_lib_platform.c:38  sec_mode =1
42956  rw_lib_platform.c:39  auth_mode =4
42960  rw_lib_platform.c:42  ---------CONN_STATUS_STA_CONNECTED--------
44580  rw_lib_platform.c:69  ipquery success addr = 0xc0a80a75   //dhcp 成功事件
44588  easy_wps.c:29  RAK_UdpServer sockfd = 0 creat          //创建 udp server 端口 25000
44998  easy_wps.c:49  recvfrom 0xc0a80a6e:25000 on sockfd=0 data_len=16 :@LT_EAS
Y_DEVICE@
45007  easy_wps.c:72  local Discovery Response               //向手机发送 mac 和 ip 地址
```

easyconfig 成功的话手机 app 会显示模块的 mac 和 ip 地址，如下图所示：

2. WPS 配置

```
app_demo_ctx.easywps_mode = CONFIG_WPS;                    //WPS 模式
rw_network_startConfig(app_demo_ctx.easywps_mode);         //开启一键配置
```

串口打印信息：

```
main.c:32  Host platform init...success
226  main.c:43  rak wifi LibVersion:1.0.4-2.1.39
230  main.c:45  rak wifi module-MAC:60:C5:A8:60:03:79
236  main.c:57  rw_network_startConfig ...                 //开始进入 WPS 配置状态
14337  rw_lib_platform.c:111  bssid = 8C:21:0A:D8:1C:0C
14342  rw_lib_platform.c:117  channel =0
14346  rw_lib_platform.c:118  ssid =Nescafe
14351  rw_lib_platform.c:119  psk =1234567890
17014  rw_lib_platform.c:23  connect_callback event = 0x0   //连接路由成功事件
17020  rw_lib_platform.c:28  ---------connected AP info list---------
17027  rw_lib_platform.c:29  bssid = 8C:21:0A:D8:1C:0C
17033  rw_lib_platform.c:35  channel =6
17037  rw_lib_platform.c:36  ssid =Nescafe
17041  rw_lib_platform.c:37  psk =1234567890
17046  rw_lib_platform.c:38  sec_mode =1
17050  rw_lib_platform.c:39  auth_mode =4
17054  rw_lib_platform.c:43  ---------CONN_STATUS_STA_CONNECTED--------
18178  rw_lib_platform.c:70  ipquery success addr = 0xc0a80a6a  //dhcp 成功事件
```

```
18186  easy_wps.c:29  RAK_UdpServer sockfd = 0 creat            //创建 udp server
```

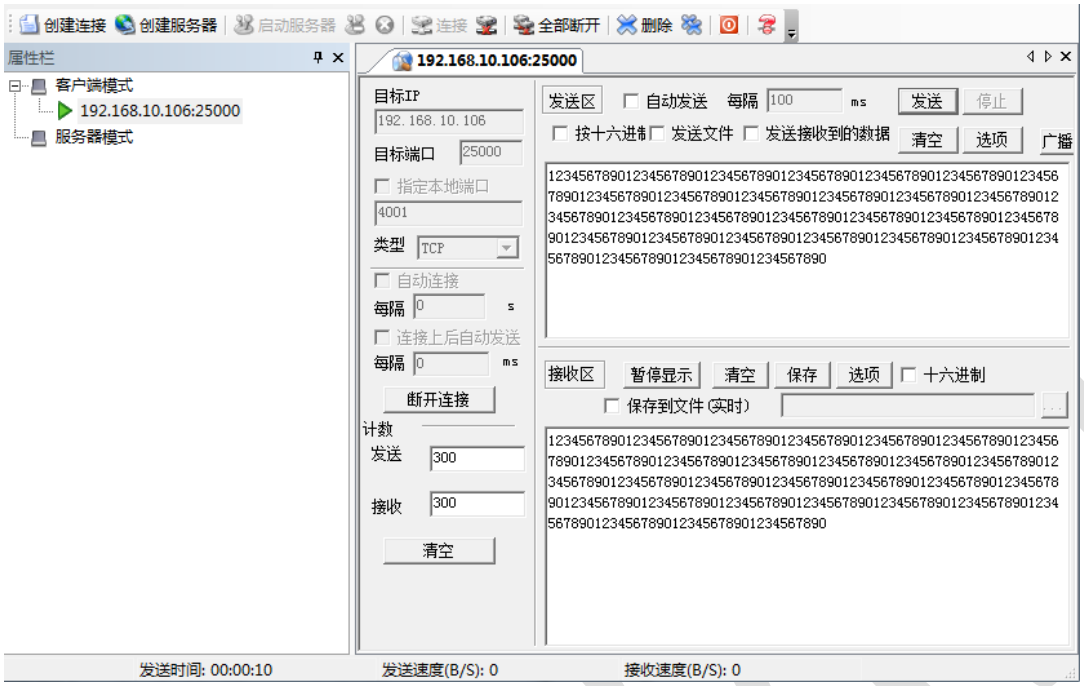
6.3.3. Socket 通信示例介绍：

模块连上路由并成功获取 IP 后，会建立一个 tcp server；
电脑上建立一个 tcp client 连接到模块，并定时向模块发数据；
模块收到数据后回发给电脑。

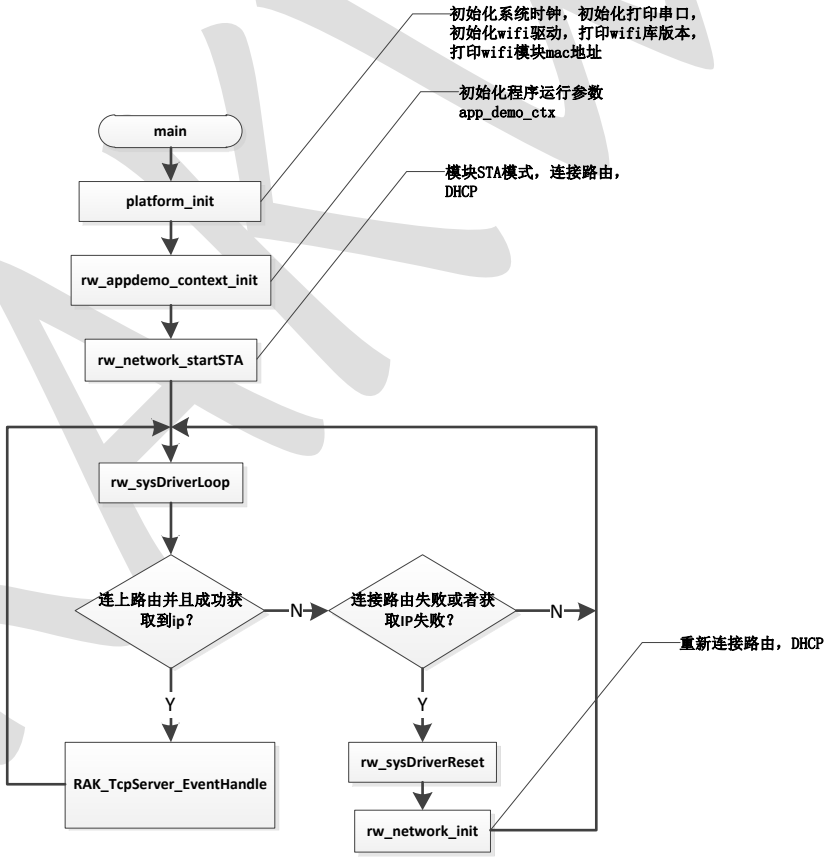
串口打印信息：

```
 main.c:32  Host platform init...success
226  main.c:43  rak wifi LibVersion:1.0.4-2.1.39
230  main.c:45  rak wifi module-MAC:60:C5:A8:60:03:79
2662  rw_lib_platform.c:23  connect_callback event = 0x0            //连接路由成功事件
2668  rw_lib_platform.c:28  ---------connected AP info list---------
2675  rw_lib_platform.c:29  bssid = 8C:21:0A:D8:1C:0C
2680  rw_lib_platform.c:35  channel =6
2684  rw_lib_platform.c:36  ssid =Nescafe
2689  rw_lib_platform.c:37  psk =1234567890
2694  rw_lib_platform.c:38  sec_mode =1
2698  rw_lib_platform.c:39  auth_mode =4
2702  rw_lib_platform.c:43  ---------CONN_STATUS_STA_CONNECTED--------
3832  rw_lib_platform.c:70  ipquery success addr = 0xc0a80a6a       //dhcp 成功事件
3841  tcp_udp.c:207  RAK_TcpServer sockfd = 0 creat                 //创建 tcp server
58804  tcp_udp.c:173  recv new sockfd=1 from ip=0xc0a824ad ,port=45466  //tcp client 连接
                                                                      //成功
```

Socket 收发数据如下图所示：



Socket 通信示例流程图：



6.4 含 OS 移植步骤

操作系统使用 FreeRTOS。与 NOS 下的移植类似，在 FreeRTOS 下的移植也需要从 rw_lib_platform_os.c 开始，注意以下几点：

- 外部中断的优先级不能高于 configMAX_SYSCALL_INTERRUPT_PRIORITY（值越小优先级越高），使用 ST 库优先级分组函数 NVIC_PriorityGroupConfig 的话，设置优先级分组为 4，优先级数值>=5；使用 CMSIS 库 NVIC_SetPriorityGrouping 的话，优先级分组随意设置，优先级数值>=5。详细请参考 http://www.freertos.org/RTOS-Cortex-M3-M4.html
- time_delay、Stamp_get 用 os 里的系统函数实现
- driver_free、driver_malloc 用 os 里的系统函数实现
- 只有 conn_cb，easy_wps_cb 回调

以下是 Wifi 驱动库的 OS 接口实现，使用了 stm32 的 cmsis_os 接口，这个接口对操作系统做了封装，使用户的应用代码可以轻松的移植到不同的操作系统。

6.4.1. rw_creat_task 实现

```
void* rw_creat_task(RW_OS_TASK_PTR p_task)
{
   osThreadId p_tcb;                                          //指向任务的句柄
   //定义任务的启动信息——任务名，任务代码地址，优先级，堆栈大小
   osThreadDef(task_wifi, (os_pthread)p_task, osPriorityHigh, 0,
               configMINIMAL_STACK_SIZE * 7);
   //创建任务
   p_tcb = osThreadCreate (osThread(task_wifi), NULL);
   return p_tcb;
}
```

6.4.2. rw_del_task 实现

```
int rw_del_task(void* p_tcb)
{
   osThreadTerminate(p_tcb);                                  //删除任务
   return RW_OS_OK;
}
```

6.4.3. rw_creat_mutex 实现

```
void* rw_creat_mutex(void)
{
   osMutexId p_mutex;
   p_mutex = osMutexCreate(osMutex(mutex));                    //创建互斥信号量
   return (void *)p_mutex;
}
```

6.4.4. rw_del_mutex

```
int rw_del_mutex(void* p_mutex)
{
   osMutexDelete(p_mutex);                                     //删除互斥信号量
   return RW_OS_OK;
}
```

6.4.5. rw_lock_mutex 实现

```
int rw_lock_mutex(void* p_mutex, uint32_t timeout)
{
  if(timeout == 0) {
    timeout = osWaitForever;
  }
  osMutexWait(p_mutex,timeout);                                //等待互斥信号量
  return RW_OS_OK;
}
```

6.4.6. rw_unlock_mutex 实现

```
int rw_unlock_mutex(void* p_mutex)
{
   osMutexRelease(p_mutex);                                    //释放互斥信号量
   return RW_OS_OK;
}
```

6.4.7. rw_creat_sem 实现

```
void* rw_creat_sem(void)
{
   osSemaphoreId p_sem;
   p_sem = osSemaphoreCreate(osSemaphore(sem),1);          //创建二值信号量
   return p_sem;
}
```

6.4.8. rw_del_sem 实现

```
int rw_del_sem(void* p_sem)
{
   osSemaphoreDelete(p_sem);                               //删除二值信号量
   return RW_OS_OK;
}
```

6.4.9. rw_post_sem 实现

```
int rw_post_sem(void* p_sem)
{
   osSemaphoreRelease(p_sem);                              //释放二值信号量
   return RW_OS_OK;
}
```

6.4.10. rw_pend_sem 实现

```
int rw_pend_sem(void* p_sem, uint32_t timeout)
{
  int oserr;
  if(timeout == 0) {
    timeout = osWaitForever;
  }
  oserr = osSemaphoreWait(p_sem,timeout);                 //等待二值信号量
  if(oserr == osOK) {
```

```
    return RW_OS_OK;
  } else if(oserr == osErrorOS) {
    return RW_OS_TIME_OUT;
  }
  return RW_OS_ERROR;
}
```

6.5 含 OS 例程介绍

/examples_os 文件夹下的示例程序包括：

- AP & STA 联网
- 一键联网——easyconfig、WPS 联网
- Socket 通信——tcp server、tcp client、udp server、udp client

6.5.1. AP & STA 联网示例介绍：

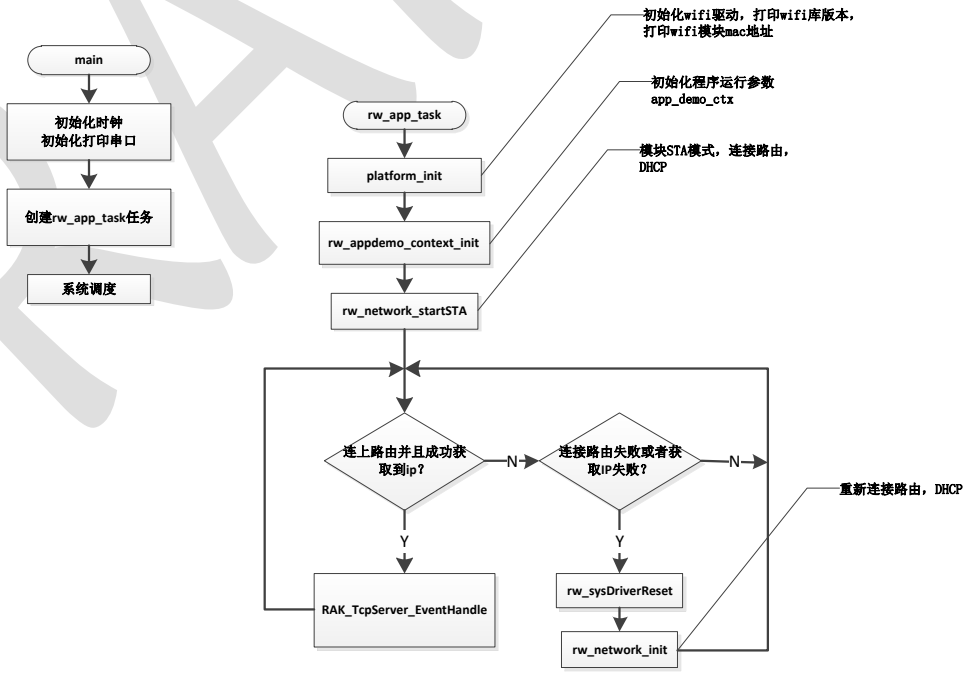
同 6.3.1

6.5.2. 一键配置示例介绍：

同 6.3.2

6.5.3. Socket 通信示例介绍

Socket 通信示例流程图：



销售与服务

北京

FAE 邮箱：allan.jin@rakwireless.com 金彦哲

电话：010-62716015

传真：010-62716015

地址：北京市海淀区德胜门外西三旗金燕龙大厦 1108 室

上海

FAE 邮箱：steven.tang@rakwireless.com 汤孝义

电话：021-54721182

传真：021-54721038

地址：上海市闵行区万源路 2161 弄 150 号冉东商务中心 1 幢 306 室

深圳

FAE 邮箱：vincent.wu@rakwireless.com 吴先顺

电话：0755- 26506594

传真：0755- 86152201

地址：深圳市南山区科技园北区清华信息港综合楼 406 室

7. 历史版本

版本号	修改内容	修改日期
V1.0	建立文档	2015-05-23
V1.1	1. 驱动库参数增加 spi_int_enable 和 tcp_retry_num 2. 应用回调接口去掉了 tcpc_cb 3. OS 接口修改了 rw_lock_mutex 和 rw_pend_sem 4. SPI 接口部分，增加了 wifi 驱动初始化时 SPI 起始传输数据介绍	2015-06-29