SIM800系列_Embedded AT _开发指导

GPRS 模组

芯讯通无线科技(上海)有限公司
上海市长宁区金钟路633号晨讯科技大楼B座6楼
电话：86-21-31575100
技术支持邮箱：support@simcom.com
官网：www.simcom.com

文档名称：	SIM800 系列_Embedded AT_开发指导
版本：	1.04
日期：	2020.6.15
状态：	已归档

版权声明

芯讯通无线科技(上海)有限公司
上海市长宁区金钟路 633 号晨讯科技大楼 B 座 6 楼
电话：86-21-31575100
邮箱：simcom@simcom.com
官网：www.simcom.com

了解更多资料，请点击以下链接：
http://cn.simcom.com/download/list-230-cn.html

技术支持，请点击以下链接：
http://cn.simcom.com/ask/index-cn.html 或发送邮件至 support@simcom.com

关于文档

版本历史

版本	日期	作者	备注
1.00	2012-10-20	毛斌	第一版
1.01	2013-11-10	毛斌	适用范围更新 添加GPIO的读写和控制接口使用注意事项 修改GPIO的读写和控制接口中枚举定义
1.02	2014-05-30	张平 朱定芬	增加socket功能说明 增加短信功能说明
1.03	2015-07-30	毛斌	增加SIM800C项目
1.04	2020-06-15	方凡	修改文档结构和风格

适用范围

本文档适用于 SIM800 系列 Embedded AT 模块，包括 SIM800W，SIM840W，SIM800V，SIM800H，SIM800，SIM800M64， SIM808, SIM800C 的 Embedded AT 模块。

本文档描述了 Embedded AT 的开发指导及相关注意事项。

目录

1 EMBEDDED AT 介绍

1.1 简介

EMBEDDED AT 主要用于客户对 SIM800 系列模块进行二次开发，SIMCom 提供相关的 API 函数，资源及运行环境，客户 app 程序运行在 SIM800 系列模块内部。这样可以不再需要外部 MCU，节省成本。

1.2 代码风格

EMBEDDED AT 基本可以实现客户外围 MCU 的编程框架。并且支持多线程，客户使用不同的线程运行不同的子任务。

1.3 SIM800 系列模块的 Embedded AT 提供的资源

API 函数主要包括以下几种：

API 种类	作用
音频 API	播放 AMR,MIDI 格式的音频，及生成指定频率的声音
AT 指令 API	主要用于客户端程序与模块程序 AT 交互使用
Flash API	对 Flash 操作的一些 API 函数，如擦除，烧写，升级应用程序等
系统 API	包括发送和接收消息、关机和重启功能、动态内存分配功能、事件获取功能、信号量操作、获取版本号功能
外围设备 API	包括 SPI 功能、GPIO 控制功能、外部中断使用、PWM 功能和 ADC 功能
Timer API	Timer 的控制及相关时间管理，包括定时器操作功能、RTC 设置功能、计时功能
文件系统 API	对文件操作的接口，读、写、创建、删除文件，删除及创建文件夹，获取文件信息功能
串口 API	读写串口功能
Debug API	打开关闭 Debug 口的功能，往 debug 口输出数据，用于往 Debug 打印客户的调试信息

升级功能 API	用于客户程序的升级，可以通过网络将新的程序下载到模块中，然后通过 API 函数进行升级
SOCKET API	提供 GPRS 激活，TCP/UDP 使用等功能
SMS API	提供短信读取，删除，发送等功能

2 Embedded AT 基本概念

以下是 EMBEDDEDAT 的软件框架图：

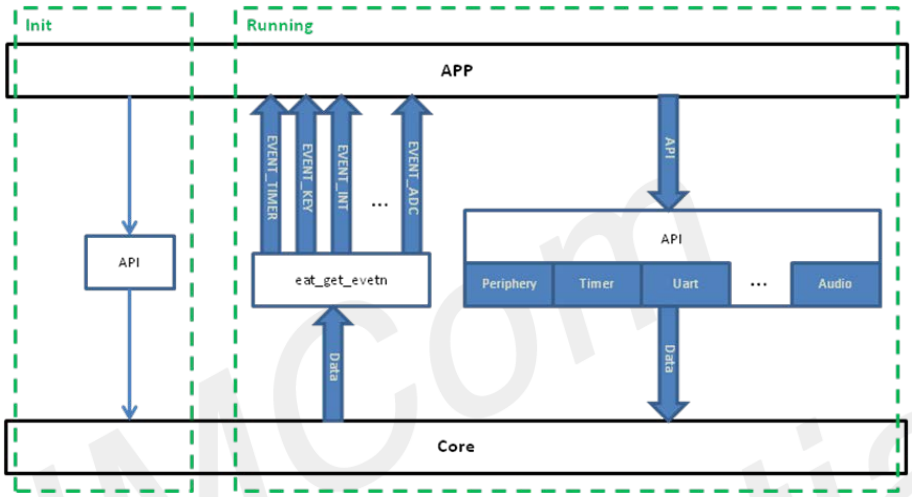


图 1：软件架构图

上图说明：

EMBEDDED AT 主要由两部分组成，一个是主程序即 core system，另一个是客户端程序即 Embedded Application。主程序提供模块的主要功能，以及 API 供客户端程序使用。主要功能包括标准的 AT 命令，如 TCPIP 连接功能，打电话功能以及发短信等相关功能。所提供的 API 包括事件获取 API，文件系统操作，timer 的控制，周围设备的 API，及一些常用的系统 API 等。

在后续章节中，一般使用 EAT 进行表述，EAT 是 EMBEDDED AT 的缩写形式。

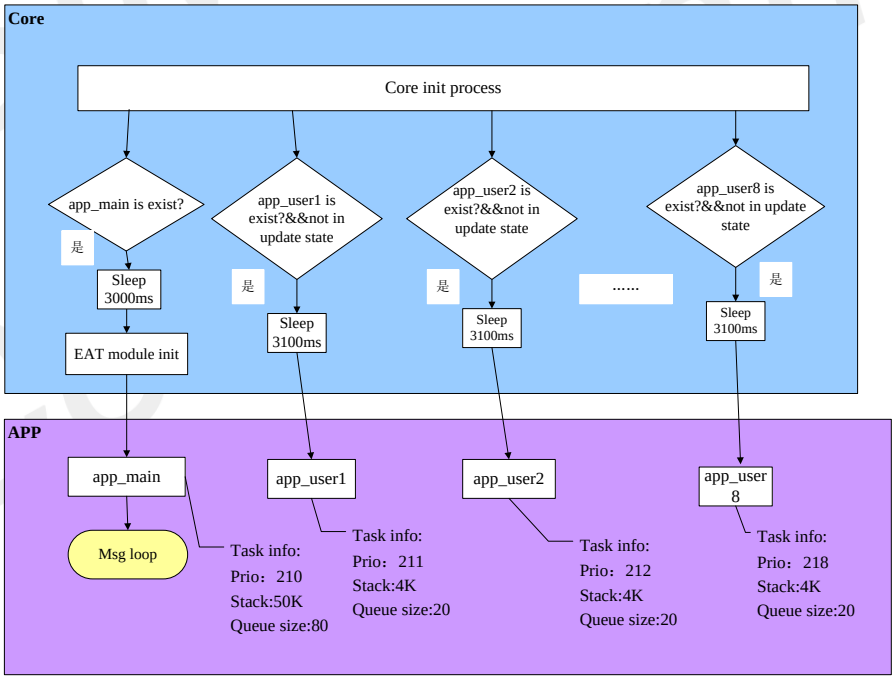
3 多线程

3.1 多线程功能描述

平台提供多线程功能，目前支持 1 个主线程和最多 8 个子线程，主线程用于和系统通信，如接收系统事件。

高优先级的 suspend 的线程，在满足运行条件时，会优于正在运行的低优先级的线程得到调度。

3.2 各线程介绍



图一线程初始化信息

- 图一说明如下：

➢ **main.c 中对应的结构体**

/*SIM800H 平台 EAT app 代码中需要加入 APP_CFG 段 begin*/

```
#pragma arm section rodata = "APP_CFG"
APP_ENTRY_FLAG
#pragma arm section rodata
/*SIM800H 平台 EAT app 代码中需要加入 APP_CFG 段 end*/

#pragma arm section rodata="APPENTRY"

const EatEntry_st AppEntry =
{
    app_main,
    app_func_ext1,
    (app_user_func)EAT_NULL,//app_user1,
    (app_user_func)EAT_NULL,//app_user2,
    (app_user_func)EAT_NULL,//app_user3,
    (app_user_func)EAT_NULL,//app_user4,
    (app_user_func)EAT_NULL,//app_user5,
    (app_user_func)EAT_NULL,//app_user6,
    (app_user_func)EAT_NULL,//app_user7,
    (app_user_func)EAT_NULL,//app_user8,
};
#pragma arm section rodata
```

- 具体 task 说明

EAT_USER_0，EAT_USER_1，EAT_USER_2 …EAT_USER_8 是用户可使用的 9 个线程，如果在结构体 EatEntry_st 数据中的成员被赋值，同时系统没有在升级过程中，则该入口会被调用，同时系统分配 task 相关的信息。

如下示例，app_main，app_func_ext1，app_user1，app_user3 会被调用：

```
const EatEntry_st AppEntry =
{
    app_main,
    app_func_ext1,
    (app_user_func)app_user1,//app_user1,
    (app_user_func)app_user2,//app_user2,
    (app_user_func)app_user3,//app_user3,
    (app_user_func) EAT_NULL,//app_user4,
    (app_user_func) EAT_NULL,//app_user5,
    (app_user_func) EAT_NULL,//app_user6,
    (app_user_func) EAT_NULL,//app_user7,
    (app_user_func) EAT_NULL,//app_user8,
……
};
```

SIM800W 和 SIM800V 中 EAT_USER_0 线程的栈大小为 50K，消息队列大小为 80，其他线程栈大小为 4K，

消息队列大小为 20。

SIM800H，SIM800，SIM800M64，SIM808 和 SIM800C 中，EAT_USER_0（EatEntry_st.entry）栈空间为 10K，队列大小为 80，EAT_USER_1 到 EAT_USER_4 栈空间为 10K，队列大小为 50，其他线程栈空间 2k byte，队列大小为 20。

线程优先级依次降低，即 EAT_USER_0 > EAT_USER_1>...> EAT_USER_8。启动次序为 EAT_USER_0，EAT_USER_1，...，EAT_USER_8。

注意：

所以使用时请注意，不要在线程中使用大的数组，以免栈溢出。

3.3 消息

3.3.1 消息定义

函数接口	功能
EAT_EVENT_TIMER	定时器消息
EAT_EVENT_KEY	按键消息
EAT_EVENT_INT	外部 GPIO 中断触发消息
EAT_EVENT_MDM_READY_RD	EAT 接收到 Modem 发过来的数据
EAT_EVENT_MDM_READY_WR	Modem 接收缓冲区由满转为非满状态时上报消息
EAT_EVENT_MDM_RI	保留，暂时不用
EAT_EVENT_UART_READY_RD	串口接收到数据
EAT_EVENT_UART_READY_WR	串口接收缓冲区由满转非满状态时上报消息
EAT_EVENT_ADC	ADC 消息
EAT_EVENT_UART_SEND_COMPLETE	串口底层硬件数据发送完成消息
EAT_EVENT_USER_MSG	线程收到其他线程消息时上报
EAT_EVENT_IME_KEY	输入法消息（只在 SIM800W 中有效）

3.3.2 接口定义

以下两个函数，仅能在 app_main 中使用，获取 core 发送过来的消息，或者 app_user1 app_user2 …app_user8 使用 eat_send_msg_to_user 发送的 ID 为 EAT_EVENT_USER_MSG 的消息。

函数接口：

函数接口	功能
eat_get_event	获取队列消息
eat_get_event_num	获取队列消息个数

示例：

```
EatEvent_st event;
u32 num;
void app_main(void *data)
{
	eat_get_event(&event);
	num=eat_get_event_num();
	if(event.event == EAT_EVENT_UART_SEND_COMPLETE)
	{
	}
}
```

在 app_user1 app_user2 …app_user8 这 8 个客户可使用的 task 中，以上功能对应的函数如下：

函数接口：

函数接口	功能
eat_get_event_for_user	获取队列消息
eat_get_event_num_for_user	获取队列消息个数

示例：

```
void app_user1(void *data)
{
	u32 num;
	EatEvent_st event;
	while(1)
	{
		num= eat_get_event_num_for_user(EAT_USER_1);
		eat_get_event_for_user(EAT_USER_1, &event);

		if(event.event == EAT_EVENT_USER_MSG)
		{
		}
	}
}
```

3.3.3 使用注意事项

- 对于消息发送，函数 eat_send_msg_to_user 可以在 app_main 及 app_user1 app_user2 …app_user8 之间使用。

- SIM800H 及 SIM800 系统消息可以发送给子线程，遵循的原则是从哪个线程调用的 API，则对应的消息就发送给哪个线程。

例如在 user1 中调用 eat_uart_open(EAT_UART_1) 打开串口 1，则当串口 1 收到数据时，EAT_EVENT_UART_READY_RD(event.uart.uart=EAT_UART_1)消息会上报到 user1 中。

例如在 app_main 中调用 eat_timer_start(EAT_TIMER_1)，在 user1 中调用 eat_timer_start(EAT_TIMER_2)，则当 EAT_TIMER_1 时间到后，EAT_EVENT_TIMER（event.timer.timer_id = EAT_TIMER_1）消息会发送到 app_main，当 EAT_TIMER_2 时间到后，EAT_EVENT_TIMER（event.timer.timer_id= EAT_TIMER_2）消息会发送到 user1 中。

例如在 user1 中调用 eat_modem_write("AT\r\n",4)发送 AT 指令到 Core，则 AT 指令执行结果通过消息 EAT_EVENT_MDM_READY_RD 上报到 user1。此后 AT 的 URC 上报也会发送到 user1，直到其他线程，例如 user2 发送 AT 指令，此后 AT 执行结果及 URC 消息会发送到 user2。

开机过程中的 AT URC 消息（EAT_EVENT_MDM_READY_RD）默认上报到 main 线程，可以通过 eat_modem_set_poweron_urc_dir( )设置开机 URC 发送到指定线程。

- eat_get_event 或者 eat_get_event_for_user 是同步接口，当调用该接口时，如果该线程有 EVENT，则立即返回，如果没有，则线程会挂起。

如果不需要线程挂起，则可以先用 eat_get_event_num()或者 eat_get_event_num_for_user(EAT_USER_x)获取当前线程事件队列中事件个数，如果为 0，则不调用 eat_get_event(_for_user)接口，大于 0 再调用该接口。

4 定时器使用

4.1 功能介绍

提供了两种定时器，分别是 16 个毫秒级别定时器和 1 个微秒级别定时器。

设置线程睡眠。

设置及获取系统当前日期及时间。

获取两个时刻的时间差。

4.2 相关功能接口及示例

EVENT：

EAT_EVENT_TIMER

结构体：

```
typedef struct {
    unsigned char sec; /* [0, 59] */
    unsigned char min; /* [0,59]    */
    unsigned char hour; /* [0,23]    */
    unsigned char day; /* [1,31]    */
    unsigned char mon; /* [1,12] */
    unsigned char wday; /* [1,7] */
    unsigned char year; /* [0,127] */
} EatRtc_st;
```

回调函数：

```
typedef void (*eat_gpt_callback_func)(void);
```

函数接口：

函数名称	描述
eat_timer_start/eat_timer_stop	启动及停止毫秒级定时器
eat_gpt_start /eat_gpt_stop	启动及停止微秒级定时器
eat_sleep	线程睡眠
eat_get_current_time	获取当前时间
eat_get_duration_us eat_get_duration_ms	获取时间差
eat_get_rtc/ eat_set_rtc	获取及设置系统时间

示例：

毫秒级别定时器使用

```
//Start the timer
eat_timer_start(EAT_TIMER_1, 100);
//Get timer EVENT
eat_get_event(&event);
if( EAT_EVENT_TIMER == event.event)
{
    //do something
}
```

微秒级别定时器使用

```
void gpt_time_handle(void)
{
    //do something...
}
eat_gpt_start(, EAT_FALSE, gpt_time_handle);
```

获取两个时刻时间差

```
unsigned int time =   eat_get_current_time();
//do something
unsigned int time_ms = eat_get_duration_ms(time);
```

设置及获取系统时间

```
EatRtc_st rtc;
eat_set_rtc(&rtc);
rtc.year = 12;
eat_get_rtc(&rtc);
```

4.3 使用注意事项

- eat_gpt_start 为硬件定时器，在中断中执行定时器回调函数。所以定时器回调函数不能占用太长时间，不能使用有阻塞函数，比如睡眠，内存分配，信号量等相关函数操作。

- 定时器的使用会影响睡眠功能。当系统睡眠时，定时时间到来时，会唤醒系统。所以在控制系统进入睡眠过程中，要考虑定时器对睡眠的影响。

5 内存使用

5.1 功能介绍

平台提供通过管理一个数组来实现内存初始化，分配和释放的接口。需要动态申请和释放的空间通过数组定义，比较灵活。

一次性能申请的最大内存空间大小为：初始化数组大小-N （N 当前等于 168，会随平台而异，但很容易测试出来）。

这个内存和全局可读写变量都占用 app 的 RAM 空间，RAM 空间的大小请参考 **Flash** 章节中的 RAM 空间大小分配。

5.2 相关功能接口及示例

函数接口：

函数接口	功能
eat_mem_init	初始化内存块
eat_mem_alloc	内存申请
eat_mem_free	内存释放
APP_InitRegions	为 APP 初始化 RW 和 ZI 数据段

示例：

```
#define DYNAMIC_MEM_SIZE 1024*400
static unsigned char app_dynic_mem_test[DYNAMIC_MEM_SIZE]; /*空间，用于内存初始化*/
void* mem_prt=EAT_NULL;

/*内存初始化*/
eat_mem_init(app_dynic_mem_test, sizeof(app_dynic_mem_test));

/*内存申请*/
mem_prt = eat_mem_alloc(size);
```

```
…

/*内存释放*/
eat_mem_free(mem_prt);

/*APP 初始化 RW 和 ZI 数据段*/
void app_main(void *data)
{
    // Local variables defined
    //.....
    APP_InitRegions;// Init app RAM, first step
    App_init_clib();    //C library initialize,second step
    //...
}
```

5.3 使用注意事项

- APP_InitRegions 在 EAT TASK0 中执行，用于初始化 APP RAM 空间的 RW 和 ZI 段。只有执行完该函数后才能进行全局数据的读。写
- APP_init_clib 在 EAT TASK0 中执行，用于初始化 APP 的 C 库环境。如果未执行该函数进行 C 库初始化，则一些 C 库函数执行结果不正确。
- APP_InitRegions 和 APP_init_clib 只能并且必须在 EAT TASK0 中执行一次。

6 睡眠

eat_sleep_enable 控制是否允许系统进入睡眠，默认不允许系统进入睡眠。

允许系统进入睡眠

```
eat_sleep_enable( EAT_TRUE );
```

禁止系统进入睡眠

```
eat_sleep_enable(EAT_FALSE );
```

模块进入睡眠后，会周期性的自动唤醒，以和网络进行通信。

在未唤醒阶段，只有按键，外部中断（GPIO），定时器，来短信，来电话会唤醒系统。

睡眠功能详细描述请参考《SIM800 系列_Embedded AT 睡眠说明_V1.01》。

7 串口

7.1 功能介绍

串口参数设置

串口数据发送接收

串口功能模式设置

有三个串口可被 app 使用，用作数据收发端口。其中任意 1 个端口可被用作 AT 指令端口，也可被用作系统 DEBUG 口（SIM800H 有两个串口和一个 USB 口）。

不同平台可用的串口数量不一定相同，请对应平台功能说明确认。

7.2 消息

EAT_EVENT_MDM_READY_RD

Modem 侧向 EAT 发送数据的过程中，发送缓冲区由空状态转至非空状态时，激发此消息通知 EAT。该发送缓冲区大小为 5KB。

EAT_EVENT_MDM_READY_WR

Modem 侧接收 EAT 发送数据的过程中，接收缓冲区由满状态转至非满状态时，激发此消息通知 EAT。该接收缓冲区大小为 5KB。

EAT_EVENT_UART_READY_RD

UART 端口接收数据，接收缓冲区由空状态转至非空状态时，激发此消息。该接收缓冲区大小为 2KB。

EAT_EVENT_UART_READY_WR

UART 端口发送数据，发送缓冲区由满状态转至非满状态时，激发此消息。该发送缓冲区大小为 2KB。

EAT_EVENT_UART_SEND_COMPLETE

UART 端口发送数据，发送缓冲区由非空状态转至空状态，且底层 DMA FIFO 也为空状态时，激发此消息。

7.3 相关功能接口及示例

枚举变量：

```
typedef enum {
    EAT_UART_1,
    EAT_UART_2,
    EAT_UART_3,
    EAT_UART_NUM,
    EAT_UART_NULL = 99
} EatUart_enum;

typedef enum {
    EAT_UART_BAUD_9600          =9600,
    EAT_UART_BAUD_19200         =19200,
    EAT_UART_BAUD_38400         =38400,
    EAT_UART_BAUD_57600         =57600,
    EAT_UART_BAUD_115200        =115200,
    EAT_UART_BAUD_230400        =230400,
    EAT_UART_BAUD_460800        =460800
} EatUartBaudrate;

typedef enum {
    EAT_UART_DATA_BITS_5=5,
    EAT_UART_DATA_BITS_6,
    EAT_UART_DATA_BITS_7,
    EAT_UART_DATA_BITS_8
} EatUartDataBits_enum;

typedef enum {
    EAT_UART_STOP_BITS_1=1,
    EAT_UART_STOP_BITS_2,
    EAT_UART_STOP_BITS_1_5
} EatUartStopBits_enum;

typedef enum {
    EAT_UART_PARITY_NONE=0,
    EAT_UART_PARITY_ODD,
```

```
    EAT_UART_PARITY_EVEN,
    EAT_UART_PARITY_SPACE
} EatUartParity_enum;
```

结构体：

```
typedef struct {
    EatUart_enum uart;
} EatUart_st;

typedef struct {
    EatUartBaudrate    baud;
    EatUartDataBits_enum    dataBits;
    EatUartStopBits_enum    stopBits;
    EatUartParity_enum    parity;
} EatUartConfig_st;
```

函数接口：

函数接口	功能
eat_uart_open eat_uart_close	打开及关闭串口
eat_uart_set_config eat_uart_get_config	设置及获取串口参数
eat_uart_set_baudrate eat_uart_get_baudrate	设置及获取传输波特率
eat_uart_write eat_uart_read	发送及接收数据
eat_uart_set_debug	设置 debug 端口
eat_uart_set_at_port	设置 Core 系统 AT 端口
eat_uart_set_send_complete_event	设置是否回报发送完成消息
eat_uart_get_send_complete_status	获取发送完成状态
eat_uart_get_free_space	获取发送缓冲区剩余空间大小
eat_modem_write eat_modem_read	发送数据至 MODEM 侧 或从 MODEM 侧接收数据

示例：

打开及关闭串口

```
//打开串口
eat_uart_open(EAT_UART_1);
```

```
//关闭串口
eat_uart_close (EAT_UART_1);
```

设置串口参数

```
EatUartConfig_st uart_config;

uart_config.baud = 115200;
uart_config.dataBits = EAT_UART_DATA_BITS_8;
uart_config.parity = EAT_UART_PARITY_NONE;
uart_config.stopBits = EAT_UART_STOP_BITS_1;

eat_uart_set_config(EAT_UART_1, &uart_config);
```

获取串口参数

```
EatUartConfig_st uart_config;

eat_uart_get_config(EAT_UART_1, &uart_config);
```

设置波特率

```
eat_uart_set_baudrate (EAT_UART_1, EAT_UART_BAUD_115200);
```

获取波特率

```
EatUartBaudrate baudrate;

baudrate = eat_uart_get_baudrate (EAT_UART_1);
```

发送及接收数据

```
u16 len;
u8 rx_buf[EAT_UART_RX_BUF_LEN_MAX];

//接收数据
len = eat_uart_read(EAT_UART_1, rx_buf, EAT_UART_RX_BUF_LEN_MAX);
if(len != 0)
{
    //发送数据
    eat_uart_write(EAT_UART_1, rx_buf, len);
}
```

设置功能端口

```
eat_uart_set_at_port(EAT_UART_1);
eat_uart_set_debug(EAT_UART_2);
```

设置是否回报发送完成消息

```
//打开发送完成消息回报
eat_uart_set_send_complete_event (EAT_UART_1, EAT_TRUE);

//关闭发送完成消息回报
eat_uart_set_send_complete_event (EAT_UART_1, EAT_FALSE);
```

获取发送完成状态

```
eat_bool status;
status = eat_uart_get_send_complete_status (EAT_UART_1);
```

获取发送缓冲区剩余空间大小

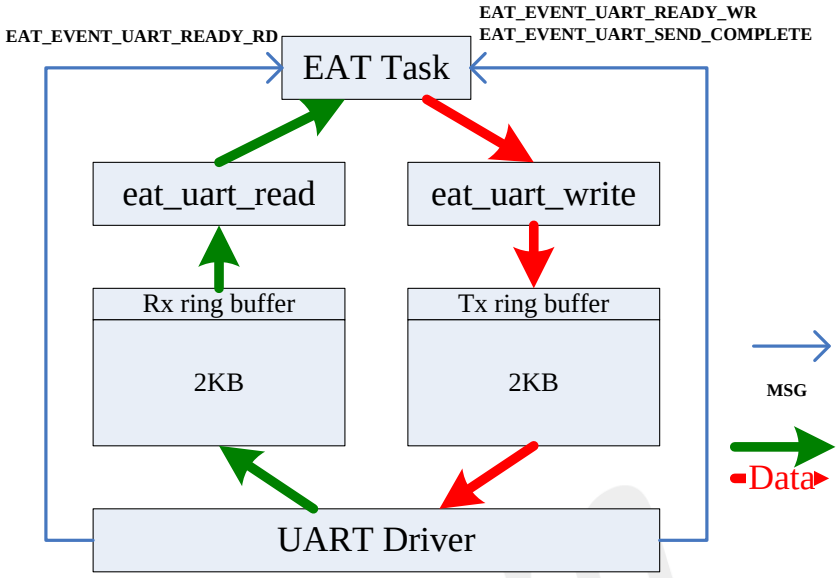
```
unsigned short size;
size = eat_uart_get_free_space (EAT_UART_1);
```

EAT 与 MODEM 侧数据传输

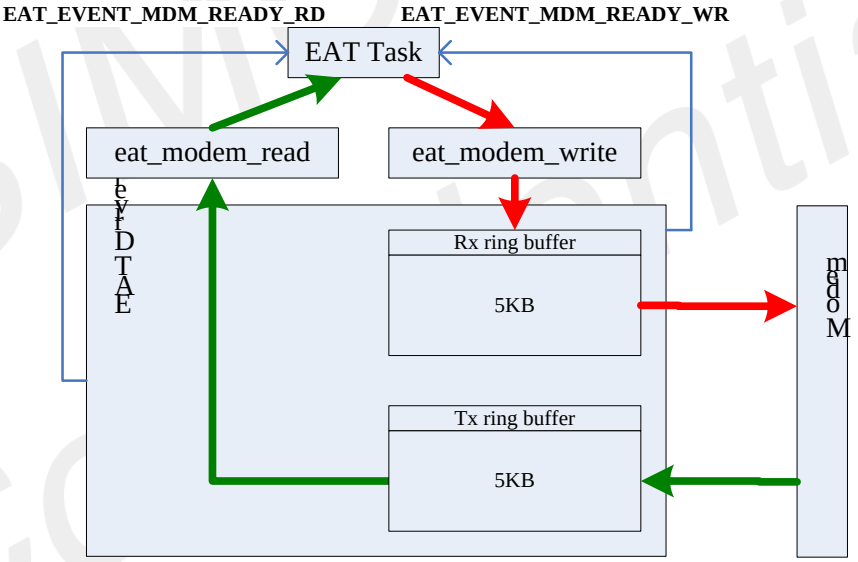
```
u16 len;
u8 rx_buf[5120];

//接收数据
len = eat_modem_read (EAT_UART_1, rx_buf, 5120);
//发送数据
eat_modem_write (EAT_UART_1, rx_buf, len);
```

7.4 串口数据流程

EAT 使用串口数据及消息流程框图：



EAT<=>Modem 端口数据及消息流程框图：



7.5 使用注意事项

- APP 收到 EAT_EVENT_MDM_READY_RD 或 EAT_EVENT_UART_READY_RD 消息后，通过 eat_modem_read 或 eat_uart_read 读取接收缓冲器的数据，如果数据没有读取完，即接收缓冲器还有数据，则下次接收缓冲器再次收到数据的时候，不会再上报 READY_RD 消息。

- 以下系列函数请在 eat_uart_open 之后，eat_uart_close 之前使用：

  - eat_uart_set_config

- ✧ eat_uart_get_config
- ✧ eat_uart_set_baudrate
- ✧ eat_uart_get_baudrate
- ✧ eat_uart_write
- ✧ eat_uart_read
- ✧ eat_uart_set_send_complete_event
- ✧ eat_uart_get_send_complete_status
- ✧ eat_uart_get_free_space

➢ 以下函数请在系统初始化时（即结构体 EatEntry_st 成员函数 func_ext1 中）调用：

- ✧ eat_uart_set_debug
- ✧ eat_uart_set_at_port
- ✧ eat_uart_set_debug_config

➢ eat_uart_set_at_port，在 1116B02V01 及之前版本上没有该功能。版本查询：
通过 EAT API eat_get_version() 和 eat_get_buildno()。

或者通过 AT 指令：

at+cgmr
Revision:1116B02SIM840W64_WINBOND_EMBEDDEDAT
OK

at+csub
+CSUB: V01
OK

8 Flash

8.1 功能介绍

提供了 Flash 的擦除、写、app 升级等功能。

8.2 相关功能接口及示例

函数接口：

函数接口	功能
eat_flash_erase	擦除 FLASH 块
eat_flash_write	数据写入 FLASH
eat_update_app	升级 app
eat_get_app_base_addr	获取 app 空间基地址
eat_get_app_space	获取 app 空间大小
eat_get_flash_block_size	获取 FLASH 块大小

8.3 空间规划

标准版本 EAT Flash 规划如下表所示。客户功能需求不同，地址规划可能会有所改变，以客户实际需求为准。

可以通过 eat_get_app_base_addr() 和 eat_get_app_space()接口获取 app 基地址和 app 空间大小。

SIM800 系列模块有多种 FLASH 版本，不同版本默认对应的地址规划如下表。

SIM800W 64M 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	08000000	083FFFFF	4M（0x400000）

APP	08400000	0867FFFF	2.5M (0x280000)
FS	08680000	087BFFFF	1.25M (0x140000)
RAM	F0220000	F03FFFFF	1.875M (0x1E0000)

SIM800V 128M 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	08000000	084FFFFF	5M (0x500000)
APP	08500000	086FFFFF	2M (0x200000)
FS	08700000	08EEFFFF	7.9M (0x7F0000)
RAM	F0220000	F03FFFFF	1.875M (0x1E0000)

SIM800H 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	101FFFFF	2M (0x200000)
APP	10200000	1037FFFF	1.5M (0x180000)
FS	10380000	103FDFFF	504K (0x7E000)
RAM	F0380000	F03FFFFF	512K (0x80000)

SIM800 32M 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	101FFFFF	2M (0x200000)
APP	10200000	1037FFFF	1.5M (0x180000)
FS	10380000	103FDFFF	504K (0x7E000)
RAM	F0380000	F03FFFFF	512K (0x80000)

SIM800 64M 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	103FFFFF	4M (0x200000)
APP	10400000	1066FFFF	2.43M (0x270000)
FS	10680000	107BFFFF	1.25M (0x140000)
RAM	F0600000	F077FFFF	1.5M (0x180000)

SIM800 BT 64M 版本：

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	105BFFFF	5.625M (0x5A0000)
APP	905A0000	9066FFFF	832K (0xD0000)
FS	10680000	107BFFFF	1.25M (0x140000)

RAM	F0600000	F077FFFF	1.5M (0x180000)

SIM800C 32M 版本

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	10308FFF	3.1M (0x309000)
APP	90309000	9039BFFF	588K (0x93000)
FS	1039E000	103FE000-1	384K(0x60000)
RAM	F0380000	F03EFFFF	448K (0x70000)

SIM808 版本

区间	起始地址	结束地址	大小（Byte）
Modem	10000000	10308FFF	3.1M (0x309000)
APP	90309000	9039BFFF	588K (0x93000)
FS	1039E000	103FE000-1	384K(0x60000)
RAM	F0380000	F03EFFFF	448K (0x70000)

APP：客户的 app 程序 ROM 空间

FS：文件系统空间，包含系统参数，校准参数等。给客户提供的文件系统空间也包含在这部分区域。系统占用了约 200K 的空间，所以用户所使用的空间大小约为 FS Size-200K。

注意：不同平台因客户需求不同，所以 Flash 规划地址及 RAM 空间大小也不一定相同。以发布的版本为准。

8.4 APP 升级

提供了运行时升级 app 的功能。

模组内部有升级标记，只有升级 app 成功后，标记才清除，所以如果在升级过程中模组掉电，则下次上电时，会根据标记来再次升级，不会出现升级过程中掉电而导致无法开机的情况出现。

升级成功后，模组会启动 app，会传递给 app_main 参数，表示升级结果。

app 升级流程如下示例：

```
u32 APP_DATA_RUN_BASE;        //app 运行地址
u32 APP_DATA_STORAGE_BASE;        //app  升级程序保存地址
const unsigned char app_new_data[] = {
#include app_new
```

```
};//升级程序数据

void update_app_test_start（）
{
        ……..
    //获取地址
    APP_DATA_RUN_BASE = eat_get_app_base_addr(); //获取 app 地址
    app_space_value = eat_get_app_space(); //获取 app 空间大小
    APP_DATA_STORAGE_BASE = APP_DATA_RUN_BASE + (app_space_value>>1); //存放 app 升级文件地址

    //擦除升级保存地址所在的 flash 空间区域
    eat_flash_erase(APP_DATA_STORAGE_BASE,update_app_data_len);
    ……..
    //将升级程序写入 flash 空间区域，起始地址为 APP_DATA_STORAGE_BASE
    eat_flash_write(APP_DATA_STORAGE_BASE, app_new_data, app_datalen);
     ……
    //升级程序
    eat_update_app((void*)(APP_DATA_RUN_BASE),        (void*)(APP_DATA_STORAGE_BASE), app_datalen, EAT_PIN_NUM, EAT_PIN_NUM, EAT_FALSE);
}
void app_main(void* data)
{
  ......
    初始化
    APP_DATA_RUN_BASE = eat_get_app_base_addr(); //获取 app 地址
    app_space_value = eat_get_app_space(); //获取 app 空间大小
    APP_DATA_STORAGE_BASE = APP_DATA_RUN_BASE + (app_space_value>>1); //存放 app 升级文件地址
 ......
}
```

调用 eat_update_app 后的升级过程如下：

- 将程序升级相关标识写入 flash 的 APP updata flag 区域。
- 模组重启，判断 APP updata flag 的值，将保存在 APP_DATA_STORAGE_BASE 地址区域的升级程序搬移到 app 运行地址 APP_DATA_RUN_BASE，并设置 flag 值。
- 模组再次重启，运行新程序。在新程序中客户通过 app_main(void param)传递的参数判断升级是否成功，并清除 APP updata flag 区域，避免相关标识影响正常程序。

```
void app_main(void *data)
{
    EatEvent_st event;
```

```
    EatEntryPara_st *para;

    APP_InitRegions();//Init app RAM

    para = (EatEntryPara_st*)data;

    memcpy(&app_para, data, sizeof(EatEntryPara_st));
    if(app_para.is_update_app && app_para.update_app_result)
{
    //APP update succeed
        eat_update_app_ok(); //clear update APP flag
}
.....
}
```

升级成功后要在新程序“app_main”中使用“eat_update_app_ok()”接口清除升级标志，如果不清除升级标志，则再次开机时，会把 APP_DATA_STORAGE_BASE 的数据再次烧写到 APP_DATA_RUN_BASE 区域。

8.5 使用注意事项

- Flash 的擦除是按照 “块” 擦除的。使用 eat_flash_erase(const void *address, unsigned int size)接口擦除 flash 时，地址应为块大小的整数倍，最小擦除一个 flash 块。如果地址不是块大小的整数倍，EAT 内部会做处理，把地址设置为擦除地址所在块的起始地址。如果 size 值不是一个 flash 块大小的整数倍，则会擦除 (size/Block size)+1 个 Flash 块。

获取 Flash 块大小接口为 eat_get_flash_block_size。

- 如果这个地址之前被写过一次，再次写之前需要先擦除。

9 文件系统

9.1 功能介绍

提供文件系统相关接口，主要完成文件系统相关操作。

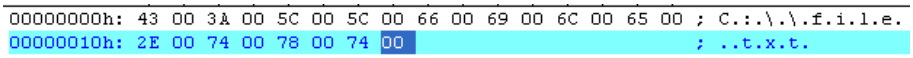
9.2 相关功能接口及示例

函数接口：

函数接口	功能
eat_fs_Open	打开或创建文件
eat_fs_Close	关闭打开的文件
eat_fs_Read	读文件
eat_fs_Write	写文件
eat_fs_Seek	查找文件指针
eat_fs_Commit	同步文件
eat_fs_GetFileSize	获取文件大小
eat_fs_GetFilePosition	获取当前文件指针
eat_fs_GetAttributes	获取文件属性
eat_fs_SetAttributes	设置文件属性
eat_fs_Delete	删除文件
eat_fs_CreateDir	创建文件目录
eat_fs_RemoveDir	删除文件目录
eat_fs_Truncate	截取文件
eat_fs_GetDiskFreeSize	获取文件系统剩余空间
eat_fs_GetFolderSize	获取文件夹大小

9.3 注意事项

- eat_fs_Open 打开或创建文件，只提供 FS_READ_WRITE、FS_READ_ONLY 、FS_CREATE 、FS_CREATE_ALWAYS 四种操作；最多支持同时打开或创建 24 个文件；创建的文件名需 two-byte 对齐和 UCS2 编码。例如直接打开一个文件不能使用 “C:\\file.txt”，而要使用 L“C:\\file.txt”，实际的值为：



Char 到 unicode 的转换示例：

```
for(i=0;i<filename_len;i++)
{
    filename_l[i*2] = filename[i];
    filename_l[i*2+1] = 0x00;
}
```

- eat_fs_GetDiskFreeSize 支持获得内部文件系统和 T 卡剩余可用空间大小。若没有外接 T 卡，则返回相应错误值。

- eat_fs_Write 对一次写多大数据没有限制，但如果超过文件系统剩余空间会出错。实际写入数据长度以该接口最后一个返回参数值为准。

- 内部文件系统盘符为“C:”，根目录为 “C:\\”。 T 卡盘符为 “D:”，T 卡根目录为 “D:\\”。文件操作时，不加盘符，则默认为内部文件系统。

10外设接口

10.1 功能介绍

外设接口主要提供一些 API 方便对 LCD，KEYPAD，ADC 等外设进行控制读写。主要提供下面这些功能：

GPIO 读写控制接口
中断配置接口
模拟 SPI 接口
PWM 输出控制接口
读 ADC 接口
PoweyKey，LED 控制接口

10.2 相关功能接口及示例

10.2.1 GPIO 的读写和控制接口

SIM800W PIN 枚举定义，其他模块平台请参考具体头文件 eat_periphery.h 中定义。

```
typedef enum    {
    EAT_PIN6_ADC0 = 6,              // ADC
    EAT_PIN8_GPIO1 = 8,             // GPIO
    EAT_PIN9_I2C_SDA = 9,           // GPIO, KEY_ROW, EINT, I2C_SDA
    EAT_PIN10_I2C_SCL = 10,         // GPIO, KEY_COL, I2C_SCL
    EAT_PIN11_KPLED = 11,           // KPLED
    EAT_PIN16_NETLIGHT = 16,// PWM
    EAT_PIN28_GPIO2 = 28,           // GPIO, EINT
    EAT_PIN29_KBC5 = 29,            // GPIO, KEY_COL, EINT
    EAT_PIN30_KBC4 = 30,            // GPIO, KEY_COL
    EAT_PIN31_KBC3 = 31,            // GPIO, KEY_COL
    EAT_PIN32_KBC2 = 32,            // GPIO, KEY_COL
    EAT_PIN33_KBC1 = 33,            // GPIO, KEY_COL
    EAT_PIN34_KBC0 = 34,            //KEY_COL
    EAT_PIN35_KBR5 = 35,            // GPIO, KEY_ROW, EINT
```

```
    EAT_PIN36_KBR4 = 36,          // GPIO, KEY_ROW
    EAT_PIN37_KBR3 = 37,          // GPIO, KEY_ROW
    EAT_PIN38_KBR2 = 38,          // GPIO, KEY_ROW
    EAT_PIN39_KBR1 = 39,          // GPIO, KEY_ROW
    EAT_PIN40_KBR0 = 40,          // GPIO, KEY_ROW, SPI_LSDI
    EAT_PIN45_GPIO3 = 45,         // GPIO, EINT
    EAT_PIN46_DISP_DATA = 46, // GPIO, SPI_LSDA
    EAT_PIN47_DISP_CLK = 47,    // GPIO, SPI_SCK
    EAT_PIN48_DISP_RST = 48,    // GPIO
    EAT_PIN49_DISP_DC = 49,     // GPIO, KEY_ROW, SPI_LSA
    EAT_PIN50_DISP_CS = 50,     // GPIO, SPI_LSCE
    EAT_PIN51_VDD_EXT = 51,     // VDD_EXT
    EAT_PIN52_PCM_SYNC = 52,    // GPIO
    EAT_PIN53_PCM_IN = 53,       // GPIO
    EAT_PIN54_PCM_CLK = 54,     // GPIO
    EAT_PIN55_PCM_OUT = 55,     // GPIO
    EAT_PIN57_GPIO4 = 57,         // GPIO, KEY_COL, EINT
    EAT_PIN58_RXD3 = 58,          // GPIO, UART3
    EAT_PIN59_TXD3 = 59,          // GPIO, UART3
    EAT_PIN60_RXD = 60,           // UART1
    EAT_PIN61_TXD = 61,           // UART1
    EAT_PIN62_DBG_RXD = 62,     // GPIO, UART2
    EAT_PIN63_DBG_TXD = 63,     // GPIO, UART2
    EAT_PIN65_LCD_LIGHT = 65, // LCD_LIGTH
    EAT_PIN_NUM = 68
} EatPinName_enum;

typedef enum {
    EAT_PIN_MODE_GPIO,
    EAT_PIN_MODE_KEY,
    EAT_PIN_MODE_EINT,
    EAT_PIN_MODE_UART,
    EAT_PIN_MODE_SPI,
    EAT_PIN_MODE_PWM,
    EAT_PIN_MODE_I2C,
    EAT_PIN_MODE_CLK,
    EAT_PIN_MODE_NUM
} EatPinMode_enum;

typedef enum {
    EAT_GPIO_LEVEL_LOW,
    EAT_GPIO_LEVEL_HIGH
} EatGpioLevel_enum;

typedef enum {
    EAT_GPIO_DIR_INPUT,
```

```
    EAT_GPIO_DIR_OUTPUT,
} EatGpioDir_enum;
```

函数接口：

函数接口	功能
eat_gpio_setup	配置 PIN 的 GPIO 属性
eat_gpio_write	写 GPIO 的电平
eat_gpio_read	读 GPIO 的电平
eat_gpio_write_ext	写 GPIO 的电平（SIM800W 对特定 PIN 可用
eat_pin_set_mode	设置 PIN 的模式

示例：

```
/*设置 PIN52 为 GPIO 输出模式，初始化为低电平*/
eat_gpio_setup(EAT_PIN52_PCM_SYNC,
  EAT_GPIO_DIR_OUTPUT, EAT_GPIO_LEVEL_LOW);

/*设置 PIN52 为高电平*/
eat_gpio_write(EAT_PIN52_PCM_SYNC, EAT_GPIO_LEVEL_HIGH);

/*读 PIN52*/
eat_gpio_read (EAT_PIN52_PCM_SYNC);

/*设置 PIN53 为高电平*/
eat_gpio_write_ext(EAT_PIN53_PCM_IN, EAT_GPIO_LEVEL_HIGH);

/*设置 PIN40 为键值模式*/
eat_pin_set_mode(EAT_PIN40_KBR0, EAT_PIN_MODE_KEY);
```

使用注意事项：

- 只有 SIM800W 中有 eat_gpio_write_ext 接口。eat_gpio_write_ext 相比较 eat_gpio_write 有更高效的写入速度，eat_gpio_write_ext 需要的时间大约是接近 1us 左右的时间，eat_gpio_write 需要的时间约是 eat_gpio_write_ext 的两倍左右，eat_gpio_write_ext 不做容错检查而且只能对一些指定的 PIN 写有效，目前支持的写 PIN 有:

  ```
  EAT_PIN8_GPIO1,
  EAT_PIN9_I2C_SDA,
  EAT_PIN10_I2C_SCL,
  EAT_PIN52_PCM_SYNC,
  EAT_PIN53_PCM_IN,
  EAT_PIN54_PCM_CLK,
  EAT_PIN55_PCM_OUT,
  EAT_PIN57_GPIO4,
  EAT_PIN58_RXD3
  ```

EAT_PIN59_TXD3,
EAT_PIN62_DBG_RXD
EAT_PIN63_DBG_TXD,

- 硬件设计需要注意一些管脚的上下拉状态，例如 SIM800W 系列模块需要特别注意如下的四个引脚，其状态值会影响开机。其他型号模块请参考对应的硬件设计文档。

  - PIN16，PIN34，PIN52 在开机之前不能强制上拉或下拉。

  - PIN64 如果做 SIM 卡检测的话(AT+CSDT=1)，必须加 10~100K 电阻上拉到 VDD_EXT。不做 SIM 卡检测的话（AT+CSDT=0），该脚可以悬空。AT+CSDT 设置可以使用 AT&W 保存该参数，下次开机时使用保存的参数。

Pin	名称	开机前状态	开机过程中强制改变电平状态可能出现的异常	说明
16	NETLIGHT	低	无法开机	内部已下拉
34	KBC0	高	无法开机	开机之前不能拉低，否则进入 USB 下载模式，导致无法正常开机
52	PCM_SYNC	高	无法开机	内部已经上拉，外部不能下拉
64	GPIO5	高	开机重启、读卡异常、不能注册网络等	用作 SIM 卡检测时不能悬空，必须在外部加 10-100K 电阻上拉到 VDD_EXT

10.2.2 模块中断配置接口

EVENT：

EAT_EVENT_INT

枚举定义：

```
typedef enum {
    EAT_INT_TRIGGER_HIGH_LEVEL, /* high level*/
    EAT_INT_TRIGGER_LOW_LEVEL, /* low level*/
    EAT_INT_TRIGGER_RISING_EDGE, /* rising edge*/
    EAT_INT_TRIGGER_FALLING_EDGE, /* falling edge*/
    EAT_INT_TRIGGER_NUM
} EatIntTrigger_enum; /* The GPIO EINT trigger mode */
```

结构体：

```
typedef struct {
    EatPinName_enum pin; /* the pin */
    EatGpioLevel_enum level; /* 1-high level; 0-low level*/
} EatInt_st; /* EAT_EVENT_INT data struct*/
```

回调函数：

```
typedef void (*eat_gpio_int_callback_func)(EatInt_st *interrupt);
```

函数接口：

函数接口	功能
eat_int_setup	设置中断
eat_int_setup_ext	设置中断，相比较 eat_int_setup 多一个参数用于设置是否自动设置中断触发极性
eat_int_set_trigger	设置中断的触发方式

示例：

```
/*定义中断回调函数*/
void int_test_handler_func (EatInt_st *interrupt)
{
    //do something
}
/*设置中断*/
eat_int_setup(EAT_PIN45_GPIO3, EAT_INT_TRIGGER_LOW_LEVEL,
100, int_test_handler_func);
/*如果没有定义中断回调函数（NULL）*/
    //Get INT EVENT
    eat_get_event(&event);
    if( EAT_EVENT_INT == event.event)
    {
        //do something
    }
/*重新设置中断触发方式*/
eat_int_set_trigger(EAT_PIN45_GPIO3, EAT_INT_TRIGGER_RISING_EDGE);
```

使用注意事项：

- eat_int_setup 的 debounce_ms 参数是以 10ms 为单位的，比如 debounce_ms=10 就是 100ms 左右，debounce 只对电平中断有效。
- 在中断触发后，如果需要进行中断反转，要在 callback 函数中进行中断反转设置。
- 边沿中断响应时间在 1ms 左右，电平中断的中断响应时间取决于 debounce_ms 的设置，大约在

debounce_ms+1ms 左右。

- 如果是电平中断，并且使用 EVENT 中断处理方式，需要在设置的时候检查当前电平，把触发条件设置和当前管脚上电平相反的电平，比如当前是 LOW_LEVEL，则触发电平要设置为 HIGH_LEVEL。因为设置电平中断时，如果触发电平和当前电平相同，则在 eat_int_setup 接口还没有退出时，中断已经触发，因为使用 EVENT 方式，所以中断会发消息给 task，但当前 task 被 GPIO 中断打断，无法进入 get_event 并设置反极性，所以电平中断会在内部一直触发。或者使用 eat_int_setup_ext 接口，最后一个参数设置电平中断触发后是否自动设置相反的触发电平。

- 在使用中断回调方式时，在中断回调函数中不要使用有阻塞及占用时间较长的操作，比如内存分配，互斥量信号量操作等。

10.2.3 SPI 接口

枚举定义：

```
typedef enum {
    EAT_SPI_3WIRE, /* 3 wire SPI */
    EAT_SPI_4WIRE /* 4 wire SPI */
} EatSpiWire_enum;

typedef enum {
    EAT_SPI_CLK_52M = 1, /* 52M clock */
    EAT_SPI_CLK_26M = 2, /* 26M clock   */
    EAT_SPI_CLK_13M = 4   /* 13M clock   */
} EatSpiClk_enum; /* Can turn down the freg if you need ,the scope is 1~1024 */

typedef enum {
    EAT_SPI_BIT8, /* 8 bit */
    EAT_SPI_BIT9, /* 9 bit */
    EAT_SPI_BIT16, /* 16 bit */
    EAT_SPI_BIT24, /* 24 bit */
    EAT_SPI_BIT32   /* 32 bit */
} EatSpiBit_enum;
```

函数接口：

函数接口	功能
eat_spi_init	SPI 初始化配置
eat_spi_write	写 SPI
eat_spi_read	读 SPI

示例：

```
/*初始化 SPI 配置*/
eat_spi_init(EAT_SPI_CLK_13M, EAT_SPI_4WIRE,
EAT_SPI_BIT8, EAT_FALSE, EAT_TRUE);
/*写数据或者写命令*/
static void lcd_write_cmd(unsigned char cmd)
{
    eat_spi_write(&cmd, 1, EAT_TRUE);
}
static void lcd_write_data(unsigned char data)
{
    eat_spi_write(&data, 1, EAT_FALSE);
}
/*读 len 个字节的数据*/
static void lcd_read_data(unsigned char *data,unsigned char len)
{
    eat_spi_read(data,len);
}
```

使用注意事项：

- 如果没有外接 DISP_CS 引脚来控制 SPI 读写，请在 app 中加以控制，eat_spi_init 的 enable_cs 参数为 false。

10.2.4 PWM 输出控制接口

函数接口：

函数接口	功能
eat_pwm_start	输出 PWM
eat_pwm_stop	停止 PWM 输出

示例：

```
/*输出 PWM*/
eat_pwm_start(200,50);

/*停止输出 PWM*/
eat_pwm_stop();
```

使用注意事项：

➢ 输出周期：0（全低）-100（全高）。

10.2.5 ADC 接口

EVENT：

EAT_EVENT_ADC

结构体：

```
typedef struct {
    EatPinName_enum pin; /* the pin */
    unsigned int v; /* ADC value,unit is mv*/
} EatAdc_st;
```

回调函数：

```
typedef void (*eat_adc_callback_func)(EatAdc_st *adc);
```

函数接口：

函数接口	功能
eat_adc_get	读 ADC

示例：

```
/*定义中断回调函数*/
void adc_test_handler_func (EatAdc _st * adc)
{
    //do something
}
/*读 ADC*/
eat_adc_get(EAT_PIN6_ADC0, 3000, adc_test_handler_func);
/*如果没有定义中断回调函数（NULL）*/
    //Get INT EVENT
    eat_get_event(&event);
    if( EAT_EVENT_ADC == event.event)
    {
        //do something
    }
```

使用注意事项：

- 目前可读 ADC 的只支持 **EAT_PIN6_ADC0（SIM800W）**这个 PIN，PIN 的外接电平范围都是 0-2.8V。
- 其他平台 ADC PIN 定义请参考 eat_periphery.h 中对应平台宏中定义。

10.2.6 PowerKey,LED 控制接口

函数接口：

函数接口	功能
eat_lcd_light_sw	LCD 背光控制接口
eat_kpled_sw	键盘背光控制接口
eat_poweroff_key_sw	PowerKey 键控制接口，如果不设置为使能，系统默认为禁止按键关机

示例：

```
/*点亮 LCD 背光*/
eat_lcd_light_sw(EAT_TRUE);//SIM800H 及 SIM800 多一个参数，可以设置电流大小

/*点亮键盘背光*/
eat_kpled_sw (EAT_TRUE);

/*允许长按 PowerKey 键关机*/
eat_poweroff_key_sw(EAT_TRUE);
```

使用注意事项：

- LCD 背光或者键盘背光点亮时将禁止系统睡眠。
- 长按 PowerKey 键 1 秒左右模块会关机。
- USB 保持连接或者 Powerkey 键一直按下，则无法关机，30 秒后会重新启动。

10.2.7 KEYPAD

按键通过消息 EAT_EVENT_KEY 上报。

EVENT：

EAT_EVENT_KEY

键值枚举定义：

```
typedef enum {
    EAT_KEY_C0R0,
    EAT_KEY_C0R1,
    EAT_KEY_C0R2,
    EAT_KEY_C0R3,
    EAT_KEY_C0R4,
    EAT_KEY_C0R5,
    EAT_KEY_C1R0,
    EAT_KEY_C1R1,
    EAT_KEY_C1R2,
    EAT_KEY_C1R3,
    EAT_KEY_C1R4,
    EAT_KEY_C1R5,
    EAT_KEY_C2R0,
    EAT_KEY_C2R1,
    EAT_KEY_C2R2,
    EAT_KEY_C2R3,
    EAT_KEY_C2R4,
    EAT_KEY_C2R5,
    EAT_KEY_C3R0,
    EAT_KEY_C3R1,
    EAT_KEY_C3R2,
    EAT_KEY_C3R3,
    EAT_KEY_C3R4,
    EAT_KEY_C3R5,
    EAT_KEY_C4R0,
    EAT_KEY_C4R1,
    EAT_KEY_C4R2,
    EAT_KEY_C4R3,
    EAT_KEY_C4R4,
    EAT_KEY_C4R5,
    EAT_KEY_C5R0,
    EAT_KEY_C5R1,
    EAT_KEY_C5R2,
    EAT_KEY_C5R3,
    EAT_KEY_C5R4,
    EAT_KEY_C5R5,
    EAT_KEY_POWER,
    EAT_KEY_NUM
} EatKey_enum;

/* EAT KEY configuration structure. */
typedef struct {
    EatKey_enum key_value; /* key value */
```

```
    eat_bool is_pressed; /* 1-key press down; 0-key release up */
} EatKey_st;
```

示例：

```
void app_main(void *data)
{
    EatEvent_st event;
    eat_get_event(&event);
    switch(event.event)
    {
        ....
        case EAT_EVENT_KEY:
        eat_trace("Get key value:%d pressed:%d", event.data.key.key_value, event.data.key.
is_pressed);
        break;
        ....
    }
```

11 音频

11.1 功能介绍

提供了 tone 音（键盘音、拨号音、忙音等）、音频数据流（MIDI 格式和 WAV 格式）的播放和停止接口。

11.2 相关功能接口及示例

函数接口：

函数接口	功能
eat_audio_play_tone_id	播放 tone 音
eat_audio_stop_tone_id	停止 tone 音
eat_audio_play_data	播放 MIDI 或 WAV 格式的数据流
eat_audio_stop_data	停止 MIDI 或 WAV 格式的数据流
eat_audio_set_custom_tone	构造自定义 tone 音

结构体：

```
typedef struct {
    unsigned short freq1; /* First frequency */
    unsigned short freq2;/* Second frequency */
    unsigned short on_duration;     /* Tone on duation, in ms unit, 0 for continuous tone */
    unsigned short off_duration;    /* Tone off duation, in ms unit, 0 for end of playing */
    unsignedchar next_operation;/*Index of the next tone */
} EatAudioToneData_st;
```

示例:

eat_audio_play_tone_id 使用

```
eat_audio_play_tone_id(EAT_TONE_DIAL_CALL_GSM, EAT_AUDIO_PLAY_INFINITE, 15, EAT_AUDIO_PATH_SPK1);
```

eat_audio_stop_tone_id 使用

```
eat_audio_stop_tone_id(EAT_TONE_DIAL_CALL_GSM);
```

eat_audio_play_data 使用

```
const unsigned char audio_test_wav_data[] = { /* ring2.wav */
   0x52,0x49,0x46,0x46,0x62,0x9B,0x04,0x00,0x57,0x41,0x56,0x45,0x66,0x6D,0x74,0x20,
   0x10,0x00,0x00,0x00,0x01,0x00,0x01,0x00,0x40,0x1F,0x00,0x00,0x80,0x3E,0x00,0x00,
   ......
}
eat_audio_play_data(audio_test_wav_data,sizeof(audio_test_wav_data),EAT_AUDIO_FORMAT_WAV,
EAT_AUDIO_PLAY_INFINITE , 12, EAT_AUDIO_PATH_SPK2);
```

eat_audio_stop_data 使用

```
eat_audio_stop_data();
```

11.3 使用注意事项

- tone 音中又分为 key tone 音和其他 audio tone 音，key tone 音的优先级比其他 audio tone 的优先级低。播放 audio tone 音会停止 key tone 音。
- 通话的优先级高于音频数据流的优先级，音频数据流的优先级高于 tone 音的优先级。所以：call 会中断音频数据流的播放，音频流播放会停止 tone 音的播放，call 也会停止 tone 音。
- 但是：call 播放时，可以播放 tone 音而不能播放音频数据流。
- 调用 eat_audio_play_tone_id() 和 eat_audio_stop_tone_id() 所用的 id 要匹配，以免没有正确停止。
- 播放音频数据流时，要保证数据是正确的格式，目前只支持 MIDI 和 WAV 格式。
- 常用 tone 音频率（参见结构体 EatAudioToneData_st）：

---忙音：EAT_TONE_BUSY_CALL_GSM：{ { 425, 0, 500, 500, 0 } }

---拨号音：EAT_TONE_DIAL_CALL_GSM, { { 425, 0, 0, 0, 0 } }

12 SOCKET

12.1 功能介绍

平台提供 SOCKET 接口，客户可以使用该接口进行 GPRS 数据传输，实现 TCP，UDP 功能

12.2 相关功能接口及示例

SOCKET 共提供 3 类接口，分别是 GPRS 承载，SOCKET 功能实现，DNS 查询。其中 GPRS 承载是基础，后面两类功能都要基于 GPRS 承载才能正常运行。

函数接口：

函数接口	功能
eat_gprs_bearer_open	激活 GPRS 承载
eat_gprs_bearer_hold	保持 GPRS 承载，如果没有调用该函数，当最后一个 SOCKET 被释放的时候，GPRS 承载也会被自动去激活。
eat_gprs_bearer_release	去激活 GPRS
eat_soc_create	创建 socket
eat_soc_notify_register	设置 socket 状态通知消息的回调函数，该函数只需设置一次，所有 socket 的状态通知都会调用该函数。
eat_soc_connect	连接 socket
eat_soc_setsockopt	设置 socket 选项
eat_soc_getsockopt	获取 socket 选项
eat_soc_bind	绑定本地端口号
eat_soc_listen	设置 socket 为服务器，等待客户端接入
eat_soc_accept	接受客户端的接入
eat_soc_send	发送数据，多用于 TCP 连接
eat_soc_recv	接受数据，多用于 TCP 连接
eat_soc_sendto	发送数据到指定地址，多用于 UDP 连接
eat_soc_recvfrom	从指定地址接受数据，多用于 UDP 连接
eat_soc_getsockaddr	得到当前 socket 的本地地址
eat_soc_close	关闭 socket

eat_soc_gethostbyname	DNS 解析，如果同时有多个 DNS 解析，设置该函数的最后一个参数用于区别不同的解析。
eat_soc_gethost_notify_register	注册 DNS 解析的回调函数，该函数只需设置一次，所有 DNS 解析后的结果都会调用该函数。不同的 DNS 解析可以使用 request_id 来区别

回调函数：

typedef void(*eat_soc_notify)(s8 s,soc_event_enum event,eat_bool result, u16 ack_size);
socket 状态通知

typedef void(*eat_bear_notify)(cbm_bearer_state_enum state,u8 ip_addr[4]);
gprs bear 状态通知

typedef void(*eat_hostname_notify)(u32 request_id,eat_bool result,u8 ip_addr[4]);
DNS 查询结果通知

结构体：

```
/* Bearer state */
typedef enum
{
    CBM_DEACTIVATED          = 0x01, /* deactivated */
    CBM_ACTIVATING           = 0x02, /* activating */
    CBM_ACTIVATED            = 0x04, /* activated */
    CBM_DEACTIVATING         = 0x08, /* deactivating */
    CBM_CSD_AUTO_DISC_TIMEOUT    = 0x10, /* csd auto disconnection timeout */
    CBM_GPRS_AUTO_DISC_TIMEOUT   = 0x20, /* gprs auto disconnection timeout */
    CBM_NWK_NEG_QOS_MODIFY       = 0x040, /* negotiated network qos modify notification */
    CBM_WIFI_STA_INFO_MODIFY     = 0x080, /* wifi hot spot sta number is changed */
    CBM_BEARER_STATE_TOTAL
} cbm_bearer_state_enum;

/* Socket return codes, negative values stand for errors */
typedef enum
{
    SOC_SUCCESS          = 0,    /* success */
    SOC_ERROR            = -1,   /* error */
    SOC_WOULDBLOCK       = -2,   /* not done yet */
    SOC_LIMIT_RESOURCE   = -3,   /* limited resource */
    SOC_INVALID_SOCKET   = -4,   /* invalid socket */
    SOC_INVALID_ACCOUNT  = -5,   /* invalid account id */
    SOC_NAMETOOLONG      = -6,   /* address too long */
    SOC_ALREADY          = -7,   /* operation already in progress */
    SOC_OPNOTSUPP        = -8,   /* operation not support */
    SOC_CONNABORTED      = -9,   /* Software caused connection abort */
```

```
    SOC_INVAL                   = -10,    /* invalid argument */
    SOC_PIPE                    = -11,    /* broken pipe */
    SOC_NOTCONN                 = -12,    /* socket is not connected */
    SOC_MSGSIZE                 = -13,    /* msg is too long */
    SOC_BEARER_FAIL             = -14,    /* bearer is broken */
    SOC_CONNRESET               = -15,    /* TCP half-write close, i.e., FINED */
    SOC_DHCP_ERROR              = -16,    /* DHCP error */
    SOC_IP_CHANGED              = -17,    /* IP has changed */
    SOC_ADDRINUSE               = -18,    /* address already in use */
    SOC_CANCEL_ACT_BEARER = -19      /* cancel the activation of bearer */
} soc_error_enum;

/* error cause */
typedef enum
{
    CBM_OK                      = 0,   /* success */
    CBM_ERROR                   = -1, /* error */
    CBM_WOULDBLOCK              = -2, /* would block */
    CBM_LIMIT_RESOURCE          = -3, /* limited resource */
    CBM_INVALID_ACCT_ID         = -4, /* invalid account id*/
    CBM_INVALID_AP_ID           = -5, /* invalid application id*/
    CBM_INVALID_SIM_ID          = -6, /* invalid SIM id */
    CBM_BEARER_FAIL             = -7, /* bearer fail */
    CBM_DHCP_ERROR              = -8, /* DHCP get IP error */
    CBM_CANCEL_ACT_BEARER    = -9, /* cancel the account query screen, such as always ask or
bearer fallback screen */
    CBM_DISC_BY_CM              = -10 /* bearer is deactivated by the connection management */
} cbm_result_error_enum;

/* Socket Type */
typedef enum
{
    SOC_SOCK_STREAM = 0,   /* stream socket, TCP */
    SOC_SOCK_DGRAM,         /* datagram socket, UDP */
    SOC_SOCK_SMS,           /* SMS bearer */
    SOC_SOCK_RAW            /* raw socket */
} socket_type_enum;

/* Socket Options */
typedef enum
{
    SOC_OOBINLINE       = 0x01 << 0,   /* not support yet */
    SOC_LINGER          = 0x01 << 1,   /* linger on close */
    SOC_NBIO            = 0x01 << 2,   /* Nonblocking */
    SOC_ASYNC           = 0x01 << 3,   /* Asynchronous notification */
```

```
    SOC_NODELAY          = 0x01 << 4,    /* disable Nagle algorithm or not */
    SOC_KEEPALIVE        = 0x01 << 5,    /* enable/disable the keepalive */
    SOC_RCVBUF           = 0x01 << 6,    /* set the socket receive buffer size */
    SOC_SENDBUF          = 0x01 << 7,    /* set the socket send buffer size */

    SOC_NREAD            = 0x01 << 8,    /* no. of bytes for read, only for soc_getsockopt */
    SOC_PKT_SIZE         = 0x01 << 9,    /* datagram max packet size */
    SOC_SILENT_LISTEN = 0x01 << 10, /* SOC_SOCK_SMS property */
    SOC_QOS              = 0x01 << 11, /* set the socket qos */

    SOC_TCP_MAXSEG       = 0x01 << 12, /* set the max segmemnt size */
    SOC_IP_TTL           = 0x01 << 13, /* set the IP TTL value */
    SOC_LISTEN_BEARER = 0x01 << 14, /* enable listen bearer */
    SOC_UDP_ANY_FPORT = 0x01 << 15, /* enable UDP any foreign port */

    SOC_WIFI_NOWAKEUP = 0x01 << 16, /* send packet in power saving mode */
    SOC_UDP_NEED_ICMP = 0x01 << 17, /* deliver NOTIFY(close) for ICMP error */
    SOC_IP_HDRINCL       = 0x01 << 18,    /* IP header included for raw sockets */
    SOC_IPSEC_POLICY          = 0x01 << 19, /* ip security policy */
    SOC_TCP_ACKED_DATA    = 0x01 << 20,    /* TCPIP acked data */
    SOC_TCP_DELAYED_ACK = 0x01 << 21, /* TCP delayed ack */
    SOC_TCP_SACK              = 0x01 << 22, /* TCP selective ack */
    SOC_TCP_TIME_STAMP    = 0x01 << 23,    /* TCP time stamp */
    SOC_TCP_ACK_MSEG    = 0x01 << 24     /* TCP ACK multiple segment */
} soc_option_enum;

/* event */
typedef enum
{
    SOC_READ       = 0x01,    /* Notify for read */
    SOC_WRITE      = 0x02,    /* Notify for write */
    SOC_ACCEPT     = 0x04,    /* Notify for accept */
    SOC_CONNECT = 0x08,    /* Notify for connect */
    SOC_CLOSE      = 0x10,      /* Notify for close */
    SOC_ACKED      = 0x20    /* Notify for acked */

} soc_event_enum;

/* socket address structure */
typedef struct
{
    socket_type_enum      sock_type; /* socket type */
    s16 addr_len; /* address length */
    u16 port; /* port number */
    u8   addr[MAX_SOCK_ADDR_LEN];
/* IP address. For keep the 4-type boundary,
```

```
     * please do not declare other variables above "addr"
    */
} sockaddr_struct;
```

示例 1：

```
/*TCP 客户端*/
/*定义 GPRS 承载回调函数*/
eat_bear_notify bear_notify_cb(cbm_bearer_state_enum state,u8 ip_addr[4])
{
    switch (state) {
        case CBM_DEACTIVATED:    /* GPRS 去激活 */
            break;
        case CBM_ACTIVATED :  /* GPRS 已经激活 */
             //这里可以根据 ip_addr 获取本地 IP 地址
            break;
        default:
            break;
    }                    /* -----  end switch  ----- */

}

/*定义 socket 回调函数*/
eat_soc_notify soc_notify_cb(s8 s,soc_event_enum event,eat_bool result, u16 ack_size)
{
    switch ( event ) {
        case SOC_READ:    /* socket 中接受到数据 */
            break;
        case SOC_WRITE:   /* socket 可以发送数据 */
            break;
        case SOC_ACCEPT:  /* 有客户端接入 */
            break;
        case SOC_CONNECT:    /* TCP 连接成功 */
            break;
        case SOC_CLOSE:   /* socket 连接断开 */
            break;
        case SOC_ACKED:   /* 对方已经确认收到的数据 */
            break;
        default:
            break;
    }                    /* -----  end switch  ----- */
}


//.....................................................................
/*激活 GPRS*/
```

```
ret = eat_gprs_bearer_open("CMNET",NULL,NULL,bear_notify_cb);   //激活 GPRS
ret = eat_gprs_bearer_hold() ; //设置 GPRS 保持

//.....................................................................
//bear_notify_cb 回调函数中，state == CBM_ACTIVATED，那么可以进行 socket 连接
/*创建 SOCKET 连接*/
eat_soc_notify_register(soc_notify_cb);   //注册回调函数
socket_id = eat_soc_create(SOC_SOCK_STREAM,0); //创建 TCP 类型的 SOCKET
val = (SOC_READ | SOC_WRITE | SOC_CLOSE | SOC_CONNECT|SOC_ACCEPT);
ret = eat_soc_setsockopt(socket_id,SOC_ASYNC,&val,sizeof(val));//设置异步通知消息
val = TRUE;
ret = eat_soc_setsockopt(socket_id, SOC_NBIO, &val, sizeof(val));//设置为非阻塞方式
address.sock_type = SOC_SOCK_STREAM;
address.addr_len = 4;
address.port = 5107;                        /* TCP server port */
address.addr[0]=116;                        /* TCP server ip address */
address.addr[1]=247;
address.addr[2]=119;
address.addr[3]=165;
ret = eat_soc_connect(socket_id,&address);//连接 TCP 服务器 116.247.119.165,端口号：5107

/*关闭 SOCKET 连接*/
ret = eat_soc_close(socket_id);

/*释放 GPRS*/
ret = eat_gprs_bearer_release();
```

示例 2：

```
/*TCP 服务器*/
/*定义 GPRS 承载回调函数*/
s8 newsocket= -1;
eat_bear_notify bear_notify_cb(cbm_bearer_state_enum state,u8 ip_addr[4])
{
    switch (state) {
        case CBM_DEACTIVATED:     /* GPRS 去激活 */
            break;
        case CBM_ACTIVATED :   /* GPRS 已经激活 */
             //这里可以根据 ip_addr 获取本地 IP 地址
            break;
        default:
            break;
    }                   /* -----   end switch   ----- */

}
```

```
/*定义 socket 回调函数*/
eat_soc_notify soc_notify_cb(s8 s,soc_event_enum event,eat_bool result, u16 ack_size)
{
    switch ( event ) {
        case SOC_READ:      /* socket 中接受到数据 */
            break;
        case SOC_WRITE:     /* socket 可以发送数据 */
            break;
        case SOC_ACCEPT:  /* 有客户端接入 */
            newsocket = eat_soc_accept(s, NULL); /*接受客户端接入并返回新的 socket id*/
            break;
        case SOC_CONNECT:      /* TCP 连接成功 */
            break;
        case SOC_CLOSE:     /* socket 连接断开 */
            if(s!=newsocket){
                /*error */
            }
            ret = eat_soc_close(s);
            newsocket= -1;
            break;
        case SOC_ACKED:    /* 对方已经确认收到的数据 */
            break;
        default:
            break;
    }                       /* -----    end switch    ----- */
}


//.....................................................................
/*激活 GPRS*/
ret = eat_gprs_bearer_open("CMNET",NULL,NULL,bear_notify_cb);   //激活 GPRS
ret = eat_gprs_bearer_hold() ; //设置 GPRS 保持
//.....................................................................
//bear_notify_cb 回调函数中，state == CBM_ACTIVATED，那么可以进行 socket 连接
/*创建 SOCKET 连接*/
eat_soc_notify_register(soc_notify_cb);   //注册回调函数
socket_id = eat_soc_create(SOC_SOCK_STREAM,0); //创建 TCP 类型的 SOCKET
address.sock_type = SOC_SOCK_STREAM;
address.addr_len = 0;/* 本地任意地址 */
address.port= 5107;/*本地端口*/
address.addr[0]=0;
address.addr[1]=0;
address.addr[2]=0;
address.addr[3]=0;
ret = eat_soc_bind(socket_id, &address); /*绑定本地地址*/
val = (SOC_READ | SOC_WRITE | SOC_CLOSE | SOC_CONNECT|SOC_ACCEPT);
```

```
ret = eat_soc_setsockopt(socket_id,SOC_ASYNC,&val,sizeof(val));//设置异步通知消息
val = TRUE;
ret = eat_soc_setsockopt(socket_id, SOC_NBIO, &val, sizeof(val));//设置为非阻塞方式
ret = eat_soc_listen(socket_id, 1);   /*创建一路监听队列*

/*关闭 SOCKET  连接*/
if(newsocket!= -1){    /*  关闭与客户端相连的连接*/
      ret = eat_soc_close(newsocket);
      newsocket= -1;
}
ret = eat_soc_close(socket_id);
/*释放 GPRS*/
ret = eat_gprs_bearer_release();
```

12.3 使用注意事项

- SOCKET 目前只支持 TCP 和 UDP 类型。
- 三个回调函数只需注册一次
- 多个 DNS 查询通过设置最后一个参数 request_id 来区分
- 目前 SOCKET 最多支持 6 路。
- 激活 GPRS 承载要在+CGREG: 1 之后，如果中途 GPRS 掉网，需要重新做激活动作。激活最大超时时间大概为 80 秒。
- TCP 每次最大发送长度为 1460，UDP 每次最大发送长度为 1472，注意不要传太大数据。
- socket 创建后，必须调用 eat_soc_close 接口去关闭，否则在多次创建后会造成资源不够。如果中途 GPRS 掉网，重新建立连接也需要先关闭 socket。

13短信

13.1 功能介绍

短信接口主要提供一些 API 对短信的发送、读取、删除等操作。提供的 API 在文件 eat_sms.h 中，相关例子在文件 app_demo_sms.c 中。

13.2 相关功能接口及示例

枚举定义：

```
typedef enum _eat_sms_storage_en_
{
    EAT_SM = 0, //SIM 卡
    EAT_ME = 1,//ME 存储
    EAT_SM_P = 2, //优先存储 SM
    EAT_ME_P = 3, //优先存储 ME
    EAT_MT = 4 //  存储到 SIM 卡和 ME 上
}EatSmsStorage_en;//短信存储类型

typedef enum
{
   EAT_SMSAL_GSM7_BIT = 0,     //7 bit  编码
   EAT_SMSAL_EIGHT_BIT,        // 8 bit 编码
   EAT_SMSAL_UCS2,             //UCS2  编码
   EAT_SMSAL_ALPHABET_UNSPECIFIED

} EatSmsalAlphabet_en;     //短信内容编码方式
```

结构体：

```
typedef struct _eat_sms_read_cnf_st_
{
    u8 name[EAT_SMS_NAME_LEN+1];//姓名
    u8 datetime[EAT_SMS_DATA_TIME_LEN+1];//时间
```

```
    u8 data[EAT_SMS_DATA_LEN+1];//短信内容
    u8 number[EAT_SMS_NUMBER_LEN+1];//号码
    u8 status;//短信状态
    u16 len;//短信长度
}EatSmsReadCnf_st;

typedef struct _eat_sms_new_message_st_
{
    EatSmsStorage_en storage; //短信存储类型
    u16 index;//短信存储索引号
}EatSmsNewMessageInd_st;//收到短信结构体
```

回调函数：

```
typedef void(* Sms_Ready_Ind)(eat_bool result); //短信初始化完成回调函数
typedef void (* Sms_Send_Completed)(eat_bool result);//短信发送回调函数
typedef void (* Sms_Read_Completed)(EatSmsReadCnf_st  smsReadContent); //读取短信回调函数
typedef void (* Sms_Delete_Completed)(eat_bool result); //删除短信回调函数
typedef void (* Sms_New_Message_Ind)(EatSmsNewMessageInd_st smsNewMessage); /收到短信回调函数
typedef void (* Sms_Flash_Message_Ind)(EatSmsReadCnf_st smsFlashMessage); //收到闪信回调函数
```

函数接口：

函数名称	描述
eat_get_SMS_sc	获取短信服务中心号码
eat_set_sms_sc	设置短信服务中心号码
eat_get_sms_ready_state	检查短信模块是否初始化完成
eat_acsii_to_ucs2	ACSII 字符转化为 UCS2 字符
eat_acsii_to_gsm7bit	ACSII 字符转化 7 bit 编码
eat_send_pdu_sms	发送 PDU 短信
eat_send_text_sms	发送 TEXT 短信
eat_get_sms_format	获取短信模式
eat_set_sms_format	设置短信模式
eat_set_sms_cnmi	设置 CNMI 命令参数
eat_set_sms_storage	设置短信存储类型
eat_get_sms_operation_mode	获取操作短信功能是通过 AT 方式还是 API 方式
eat_set_sms_operation_mode	设置操作短信功能是通过 AT 方式还是 API 方式
eat_read_sms	读取一条短信内容
eat_delete_sms	删除一条短信内容
eat_sms_decode_tpdu	解析 PDU 短信内容
eat_sms_orig_address_data_convert	转化 PDU 内容中原地址号码

eat_sms_register_send_completed_callback	注册发送短信回调函数
eat_sms_register_new_message_callback	注册收到短信回调函数
eat_sms_register_flash_message_callback	注册收到闪信回调函数
eat_sms_register_sms_ready_callback	注册短信模块初始化完成回调函数

示例

```
/*定义收到闪信回调函数*/
static eat_sms_flash_message_cb(EatSmsReadCnf_st smsFlashMessage)
{
    u8 format =0;

    eat_get_sms_format(&format);
    eat_trace("eat_sms_flash_message_cb, format=%d",format);
    if(1 == format)//TEXT 模式
    {
        eat_trace("eat_sms_read_cb, msg=%s",smsFlashMessage.data);
        eat_trace("eat_sms_read_cb, datetime=%s",smsFlashMessage.datetime);
        eat_trace("eat_sms_read_cb, name=%s",smsFlashMessage.name);
        eat_trace("eat_sms_read_cb, status=%d",smsFlashMessage.status);
        eat_trace("eat_sms_read_cb, len=%d",smsFlashMessage.len);
        eat_trace("eat_sms_read_cb, number=%s",smsFlashMessage.number);
    }
    else//PDU 模式
    {
        eat_trace("eat_sms_read_cb, msg=%s",smsFlashMessage.data);
        eat_trace("eat_sms_read_cb, len=%d",smsFlashMessage.len);
    }
}
/*定义收到短信回调函数*/
static eat_sms_new_message_cb(EatSmsNewMessageInd_st smsNewMessage)
{
    eat_trace("eat_sms_new_message_cb,
      storage=%d,index=%d",smsNewMessage.storage,smsNewMessage.index);
}
/*定义读取短信回调函数*/
static void eat_sms_read_cb(EatSmsReadCnf_st  smsReadCnfContent)
{
    u8 format =0;

    eat_get_sms_format(&format);
    eat_trace("eat_sms_read_cb, format=%d",format);
```

```
    if(1 == format)//TEXT 模式
    {
        eat_trace("eat_sms_read_cb, msg=%s",smsReadCnfContent.data);
        eat_trace("eat_sms_read_cb, datetime=%s",smsReadCnfContent.datetime);
        eat_trace("eat_sms_read_cb, name=%s",smsReadCnfContent.name);
        eat_trace("eat_sms_read_cb, status=%d",smsReadCnfContent.status);
        eat_trace("eat_sms_read_cb, len=%d",smsReadCnfContent.len);
        eat_trace("eat_sms_read_cb, number=%s",smsReadCnfContent.number);
    }
    else//PDU 模式
    {
        eat_trace("eat_sms_read_cb, msg=%s",smsReadCnfContent.data);
        eat_trace("eat_sms_read_cb, name=%s",smsReadCnfContent.name);
        eat_trace("eat_sms_read_cb, status=%d",smsReadCnfContent.status);
        eat_trace("eat_sms_read_cb, len=%d",smsReadCnfContent.len);
    }
}
/*定义删除短信回调函数*/
static void eat_sms_delete_cb(eat_bool result)
{
    eat_trace("eat_sms_delete_cb, result=%d",result);
}
/*定义删除短信回调函数*/
static void eat_sms_send_cb(eat_bool result)
{
    eat_trace("eat_sms_send_cb, result=%d",result);
}
/*定义短信模块初始化完成回调函数*/
static void eat_sms_ready_cb(eat_bool result)
{
    eat_trace("eat_sms_ready_cb, result=%d",result);
}
/*APP MAIN 启动，注册上面回调函数*/
void app_main(void *data)
{
    //do something

    //注册短信相关回调函数
    eat_set_sms_operation_mode(EAT_TRUE);//设置 API 方式操作短信模块
    eat_sms_register_sms_ready_callback(eat_sms_ready_cb);
    eat_sms_register_new_message_callback(eat_sms_new_message_cb);
    eat_sms_register_flash_message_callback(eat_sms_flash_message_cb);
    eat_sms_register_send_completed_callback(eat_sms_send_cb);

    while(EAT_TRUE)
    {
```

```
            //do something
        }
}
/*设置短信模式为 PDU 模式*/
eat_set_sms_format(0);
/*设置短信服务中心号码*/
u8 scNumber[40] = {"+8613800210500"};
eat_set_sms_sc(scNumber);
/*设置短信存储类型*/
u8 mem1, mem2, mem3;
eat_bool ret_val = EAT_FALSE;

mem1 = EAT_ME;
mem2 = EAT_ME;
mem3 = EAT_ME;
ret_val = eat_set_sms_storage(mem1, mem2, mem3);
/*设置 CNMI 命令参数*/
mode = 2;
mt = 1;
bm = 0;
ds = 0;
bfr = 0;
ret_val= eat_set_sms_cnmi(mode,mt,bm,ds,bfr);
/*读取指定短信内容*/
u16 index = 1;
ret_val = eat_read_sms(index,eat_sms_read_cb);
/*发送短信内容*/
u8 format = 0;
eat_bool ret_val = EAT_FALSE;

eat_get_sms_format(&format);
if(1 == format)
{
    ret_val = eat_send_text_sms("13681673762","123456789");
}
else
{
    ret_val = eat_send_pdu_sms("0011000D91683186613767F20018010400410042",19);
}

/*删除指定短信内容*/
u16 index = 1;
ret_val = eat_delete_sms(index,eat_sms_delete_cb);
/*解析收到短信内容*/
u8
ptr[]= "0891683108200105F0040D91683186613767F2000041301251658523063 1D98C56B301";
```

```
u8 len = strlen(ptr);
EatSmsalPduDecode_st sms_pdu = {0};
u8 useData[320] = {0};
u8 useLen = 0;
u8 phoneNum[43] = {0};

ret = eat_sms_decode_tpdu(ptr, len, &sms_pdu);
```

备注：

更具体、更全面示例，可以参考文件 app_demo_sms.c

13.3 使用注意事项

- 在 APP 的线程初始化时，需要注册短信模块所有回调函数，若未注册相应回调函数，则相关功能将失效。
- eat_sms_register_sms_ready_callback 注册的回调函数返回 EAT_TRUE 时，则表示短信初始化完成，否则操作短信模块 API 均失败。
- eat_set_sms_operation_mode (eat_bool mode)接口中参数设置 EAT_TRUE 时，只能使用所提供的 API 进行短信操作。若设置 EAT_FALSE，则只能使用 AT 命令操作短信。
- eat_send_text_sms 和 eat_send_pdu_sms 接口发送短信长度依据短信编码方式：7bit code 是 160；8bit code 是 140；UCS2 code 是 70。
- 操作短信 API 时，建议先使用 eat_get_sms_ready_state 接口判断短信模块是否初始化完，初始化完成后再执行短信操作。