SIM7020系列_Embedded AT_应用文档

LPWA 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

名称：	SIM7020系列_Embedded AT_应用文档
版本：	1.01
日期：	2020.6.10
状态：	发布

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	2019-04-08	宋孝坤	第一版
1.01	2020-06-10	来文洁	All

适用范围

本文档适用于以下产品型号：

型号	类别	尺寸（mm）	备注
SIM7020C	NB1	17.6*15.7	频段 1/3/5/8
SIM7020E	NB1	17.6*15.7	频段 1/3/5/8/20/28
SIM7030	NB1	16*18	频段 LTE FDD 1/3/5/8
SIM7060	NB1+GNSS	24*24	频段 LTE FDD 5/8
SIM7020G	NB2	17.6*15.7	频段 1/2/3/4/5/8/12/13/17/18/19/20/25/26/28/66/70/71/85
SIM7060G	NB2+GNSS	24*24	频段 1/2/3/4/5/8/12/13/17/18/19/20/25/26/28/66/70/71/85

目录

1 介绍

1.1 本文目的

基于 AT 指令手册扩展，本文主要介绍 EAT 的架构和开发应用流程。
参考此应用文档，开发者可以很快理解并快速开发应用。

1.2 参考文档

[1] SIM7020 Series_AT Command Manual

1.3 术语和缩写

2 Embedded AT 介绍

EMBEDDED AT 主要用于客户对 SIM7020 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 SIM7020 模块内部。这样可以不再需要外部 MCU，节省成本,目前已经广泛应用于 M2M 领域，例如智能家居，智慧城市，能源抄表等。

2.1 系统架构

Embedded AT 的软件基本架构原理如下图 1 所示：

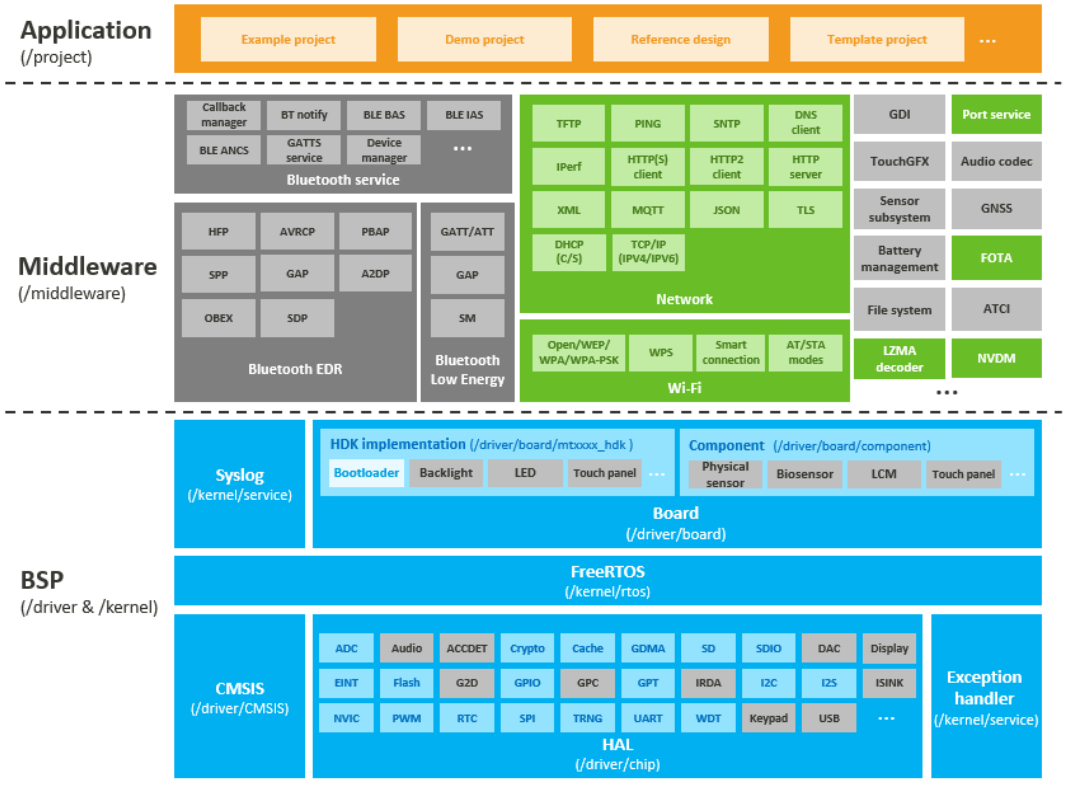
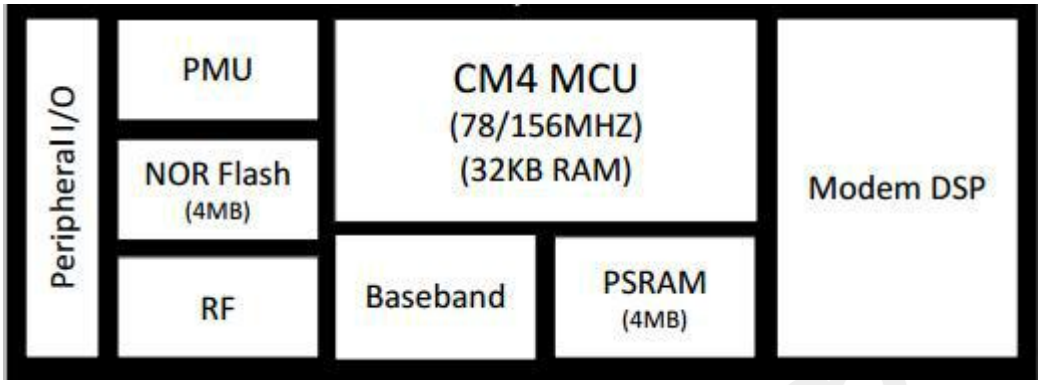


图 1: SIM7020 Embedded AT 系统架构

2.2 Open Source

2.2.1 处理器

32-bit ARM Cortext-M4 RISC 78MHz.



2.2.2 Memory Scheme

CODE Space: 256K

RAM Space: 256K

Data FLASH: 64KB

2. 3 开发环境

请参考 SIM7020 Series EAT Environment & Compilation & Burning Guide。

3 Embedded AT 多线程介绍

平台提供多线程功能，用于处理各个实时的任务。

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。

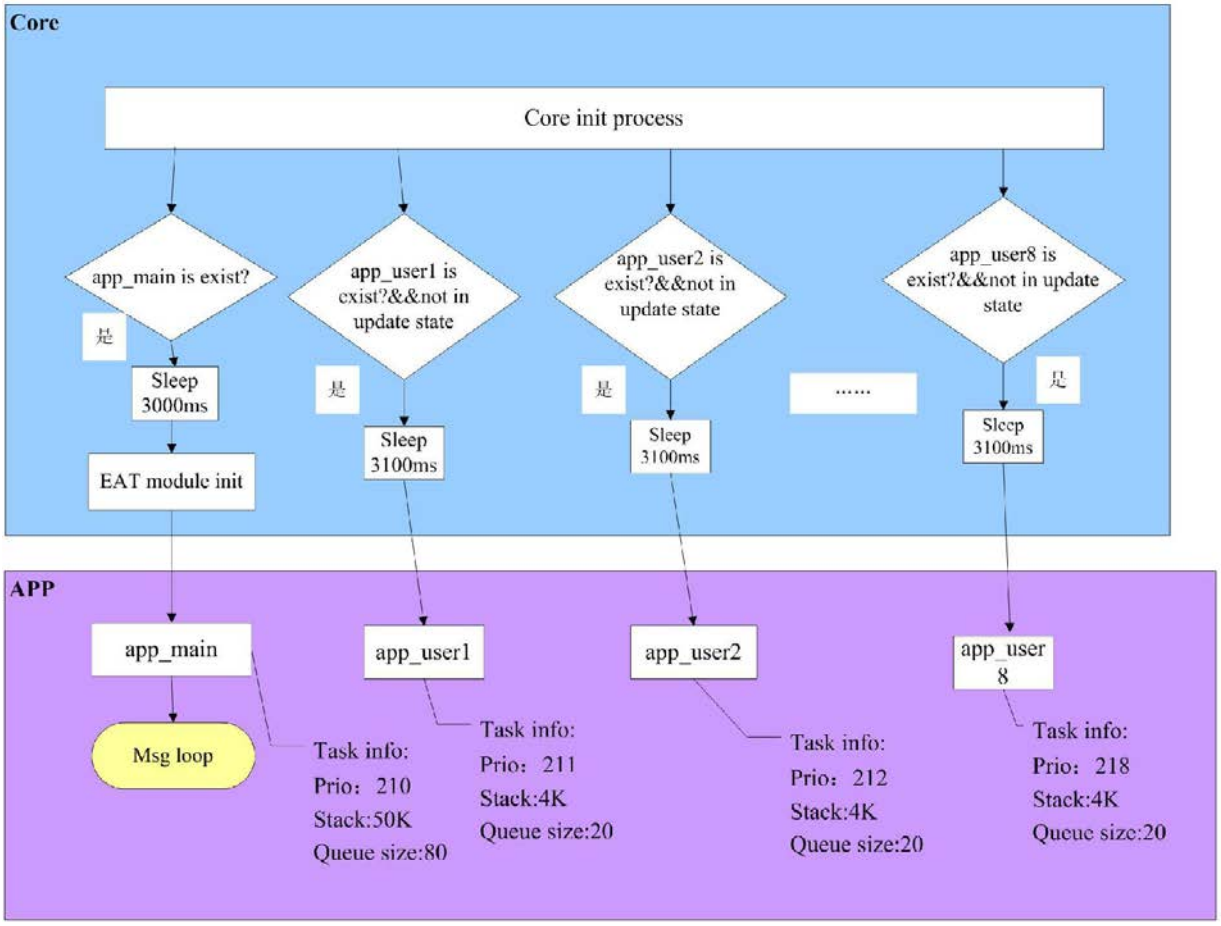


图 3 线程初始化信息

图 3 说明如下：

1) eat_task_main.c 中对应的宏定义

```
#define EAT_TASK_NAME           "EAT"
#define EAT_TASK_STACKSIZE      (1024 * 4) /*unit is bytes.*/
```

```
#define EAT_TASK_PRIO                TASK_PRIORITY_NORMAL
#define EAT_QUEUE_LENGTH             50
```

2) 具体 task 说明

用户可以自己创建自己的线程，然后在线程初始化的时候打开线程即可，这样系统就会运行相应的线程代码。如下所示，是在线程初始化函数中创建示例线程的代码

```
int    eat_task_init(void)
{
 int    xReturn = 0;
 eat_queue_handle = xQueueCreate(EAT_QUEUE_LENGTH, sizeof(eat_msg_t));
 if (eat_queue_handle == NULL)
 {
       LOG_E(common, "eat_queue create fail!");
       return 0;
 }
          xReturn  =   xTaskCreate(eat_task_main,  EAT_TASK_NAME,  EAT_TASK_STACKSIZE  /
          sizeof(portSTACK_TYPE), NULL, EAT_TASK_PRIO, &eat_task_handle);
 if (xReturn == 0)
 {
       vQueueDelete(eat_queue_handle);
       eat_queue_handle = NULL;
       LOG_E(common, "eat_task_main task create fail!");
       return 0;
 }
}
```

其中 task_main 是线程的具体实现，EAT_TASK_NAME 是线程的名称，EAT_TASK_PRIO 是线程的优先级大小。

NOTE

- 使用时请注意，不要在线程中使用大的数组，如确实需要，可以动态分配，以免栈溢出。

4 Embedded AT API 功能

4.1 系统 API

4.1.1 用法

本节介绍系统级编程中的一些重要操作和 API 函数

4.1.2 互斥

互斥对象是一个同步对象，其状态设置为在不由任务拥有事发出信号，在拥有时不发信号。任务一次性只能拥有一个互斥对象。例如为了防止两个任务同时访问共享内存，Embedded AT 任务在执行访问内存的代码之前等待互斥对象的所有权。写入共享内存后，该任务将释放互斥对象。

第 1 步：创建互斥锁。 开发人员可以调用 xSemaphoreCreateRecursiveMutex（）来创建互斥锁。

第 2 步：获取指定的互斥锁。 如果开发人员想要使用互斥机制进行编程，他们可以调用 xSemaphoreTake（）来获取指定的互斥锁 ID。

第 3 步：释放指定的互斥锁。 开发人员可以调用 xSemaphoreGive 来释放指定的互斥锁。

4.1.3 接口介绍

4.1.3.1 eat_reset_system

功能：系统复位

原型：
void eat_reset_system(void)

参数：
无

返回值：

无

4.1.3.2 eat_sleep

功能：此函数暂停当前任务的执行，直到超时间隔结束。睡眠时间不应太长，因为如果任务暂停时间过长，可能会在结束后收到太多的信息而导致错乱。

原型：
void eat_sleep(int delay)

参数：
delay[in]：int 型，表示延时时间，单位 ms，最小值为 5ms。

返回值：
无

4.1.3.3 eat_sleep_enable

功能：
使能或者失能模块进入 edrx/drx 睡眠模式。

原型：
void eat_sleep_enable(bool en)

参数：
en[in]：bool 型 true：允许模块进入 edrx/drx 睡眠模式。
false：禁止模块进入 edrx/drx 睡眠模式。

返回值：
无

4.1.3.4 eat_powerdown_system

功能：
系统关机

原型：
void eat_powerdown_system(void)

参数：
无

返回值
无

4.1.3.5 eat_app_tick_hook_open

功能：
打开一个 hook，可以被用作 app 看门狗来使用。

原型：
uint32_t eat_app_tick_hook_open(uint32_t cnt, void *func)

参数:
cnt[in]：uint32 型，单位为 ms，最小为 10ms。
func[in]：void * 型，如果函数地址非空，每 cnt/10ms 将会执行一次该函数。Func 的类型为 eat_app_tick_hook_cb_type， typedef void(*eat_app_tick_hook_cb_type)(void);

返回值：
1 表示调用成功。
0 表示调用失败。

4.1.3.6 eat_get_retention_data

功能：
获取保留数据，这个数据可以在 PSM 模式下一直保持。

原型：
void eat_get_retention_data(uint8_t *data, uint16_t len)

参数：
data[in]：uint8_t * 型，读取数据存放的目标地址
len[in] ：uint16_t 型，要读取的数据的长度，最大不能超过 32 字节。

返回值：
无

4.1.3.7 eat_set_retention_data

功能：
设置保留数据，此数据可以在 PSM 模式下一直保持。

原型：
void eat_get_retention_data(uint8_t *data, uint16_t len)

参数：
data[in]：uint8_t 型，被设置数据缓存数组地址，设置最多不能超过 32 字节。

len[in]：uint16_t 型，设置数据的长度，最大不能超过 32 个字节。

返回值：
无

4.1.3.8 eat_get_powerup_mode

功能：获取开机启动模式

原型：
uint8_t eat_get_powerup_mode(void)

参数：
无

返回值：
0 – 正常初始化(COLD-BOOT)
1 -- PSM 退出(DEEP-SLEEP-BOOT)

4.1.4 示例

```
static SemaphoreHandle_t test_mutex = NULL;
void test_mutex_create(void)
{
    if (test_mutex == NULL) {
        test_mutex = xSemaphoreCreateMutex();
    }
    if (test_mutex == NULL) {
        NW_LOG("test_mutex_creat error");
        return;
    }
    NW_LOG("test_mutex_creat success");
}


void test_mutex_take(void)
{
    if (xTaskGetSchedulerState() == taskSCHEDULER_RUNNING && test_mutex != NULL) {
        if (xSemaphoreTake(test_mutex, portMAX_DELAY) == pdFALSE) {
            NW_LOG("test_mutex_take error");
        }
        NW_LOG("test_mutex_take success");
```

```
    }
}


void test_mutex_give(void)
{
    if (xTaskGetSchedulerState() == taskSCHEDULER_RUNNING && test_mutex != NULL) {
        if (xSemaphoreGive(test_mutex) == pdFALSE) {
            NW_LOG("test_mutex_give error");
        }
        NW_LOG("test_mutex_give success");
    }
}
void test_Semaphore(void)    //Two task Run this function at the same time
{
    test_mutex_take()
    eat_sleep(3000);
    test_mutex_give();
}


static void eat_task_test(void *pvParameters)
{
    int flag = 0;
    char set_retention_data[32] = "test";
    char get_retention_data[32] = "";
    test_mutex_create();//creat a Semaphore
    eat_sleep_enable(true);
    while（1）
    {
        switch(flag)
        {
            case 0 : eat_set_retention_data(set_retention_data,32) ;// set retention data
                     break;

            case 1 : eat_get_retention_data(get_retention_data,32); // get retention data
                     break;

            case 2 : test_Semaphore();    //test Semaphore
                     break;

            case 3 :
                     if(1 == eat_get_powerup_mode())
                         eat_reset_system();//系统复位
                     else
                         eat_powerdown_system();//  系统关机
```

```
                        break;

                default: break;
            }
            eat_sleep(500);
            flag ++;
            if(flag > 3) flag = 0;
        }

}
```

4.2 定时器 API

EMBEDDED EAT 提供两种定时器，一种是通用功能定时器，一种是 RTC 定时器。

4.2.1 用法

开发人员可以通过 eat_start_rtc_timer()去打开一个 rtc 定时器，通过 eat_stop_rtc_timer()去关闭正在运行的 rtc 定时器。

4.2.2 接口介绍

4.2.2.1 eat_start_rtc_timer

功能：
打开一个 RTC 定时器，用于唤醒开机

原型：
uint32_t eat_start_rtc_timer(uint32_t lifetime)

参数：
lifetime[in]：unit32_t 型，单位为 S，最小设置值为 5S

返回值：
0 ： 打开 RTC 定时器失败
1 ： 打开 RTC 定时器成功

4.2.2.2 eat_stop_rtc_timer

功能：
关闭 RTC 定时器

原型：
void eat_stop_rtc_timer(void)

参数：
无

返回值：
无

4.2.3 示例

```
void test_eat_rtc(void)
{
    eat_sleep_enable(true);
    eat_at_input("AT+CPSMS=1\r\n",strlen("AT+CPSMS=1\r\n"));
    eat_start_rtc_timer(300);
}
```

4.3 AT API

EMBEDDED EAT 提供了内部 AT 操作的相关 API 函数。

4.3.1 用法

开发人员可以通过 eat_at_open（）打开串口，eat_at_input（）模块内部发送 AT 指令，eat_at_output（）接收 AT 指令返回数据。

4.3.2 接口介绍

4.3.2.1 eat_at_open

功能：
打开 AT port

原型：
unsigned int eat_at_open(void *func);

参数：
func[in]：void * 型，如果 func 非空，当收到 AT 数据时，将会回调 func，func 的类型为 eat_empty_cb_type
typedef void(* eat_empty_cb_type)(void);

返回值：
1 – 调用成功
0 –已经被其它打开

4.3.2.2 eat_at_input

功能：发送 AT 命令

原型：
unsigned int eat_at_input(unsigned char *data, unsigned int length)

参数：
data[in]：char * 型，所要发送的 AT 指令的缓存地址。
length[in]：int 型，所要发送的 AT 指令的长度。

返回值：
无

4.3.2.3 eat_at_output

功能：
接收 AT 指令以及 AT 指令返回的数据。

原型：
unsigned int eat_at_output(unsigned char *data, unsigned int length)

参数：
data[out]：char * 型，接收数据的缓存数组的地址，数组长度需要大于 2048 个字节，否则可能会导致数据丢失
length[in]：int 型，缓存数组的长度

返回值：

无

4.3.3 示例

```
uint8_t testbuf[1024];
void eat_at_not_empty_cb(void)
{
    uint8_t len = eat_at_output(testbuf,1024);
    testbuf[len] = 0x00;
    printf("##eat_at_not_empty_cb %s\r\n",testbuf );
}

static void eat_test_timer_handle( TimerHandle_t tmr )
{
    static int flag = 0;

    switch(flag)
    {
        case 0:
          eat_at_input("ATI\r\n",strlen("ATI\r\n"));
          break;

        case 1:
          eat_at_input("AT+CSQ\r\n",strlen("AT+CSQ\r\n"));
          break;

        case 2:
          eat_at_input("AT+CREG?\r\n",strlen("AT+CREG?\r\n"));
          break;
    }

    flag++;
    if(flag > 3)
    {
        flag = 0;
    }
    xTimerStart(eat_test_timer, 0);
}

static void eat_task_main(void *pvParameters)
{
    int mode = 0 ;
    int rt = 0;
```

```
    eat_test_timer = xTimerCreate( "eat_test_timer", (1000/portTICK_PERIOD_MS), /* interval 1 second.
*/ pdFALSE, NULL,eat_test_timer_handle);
    xTimerStart(eat_test_timer, 0);

    eat_at_open(eat_at_not_empty_cb);
    while(1)
    {
        eat_sleep(3000);
        printf("##eat_task_main %d\r\n",mode++ );
    }
}
```

4.4Flash API

EMBEDDED EAT 提供了 64KB 大小的空间可供用户自己使用，以及 500KB FOTA 区域用于 FOTA 升级。

4.4.1 用法

开发人员可以通过 eat_flash_erase（）擦除相应的 flash 区块，eat_flash_write（）像相应的地址写入数据，eat_flash_read（）读取相应地址的数据，eat_get_flash_block_size（）获取区块的大小。

4.4.2 空间规划

标准版本 Embedded AT 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。

SIM 7020 版本：

区间	起始地址	结束地址	大小（Byte）
RAM	00000000	001FFFFF	2M(0x00200000)
RTOS	08012000	0830EFFF	3060K(0x2FD000)
FOTA	0830F000	083A4FFF	600K(0x96000)
FS	03F00000	03F0FFFF	64K(0x10000)

NOTE

- 目前只开放了 64K 的区域的 flash 读写 API，以及用于 FOTA 升级的 600K 的区域的 flash 读写 API

4.4.3 接口介绍

4.4.3.1 eat_flash_erase

功能：
擦除相应的 flash 区域

原型：
bool eat_flash_erase(const void *address, unsigned int size)

参数：
address[in]：void * 型，所要擦除的 flash 区域的起始地址。
size[in]：int 型，擦除的大小

返回值：
true – 表示擦除成功
false – 表示擦除失败

4.4.3.2 eat_flash_write

功能：
往 flash 相应的地址写入相应长度的数据

原型：
bool eat_flash_write(const void *address, const void *data, unsigned int len)

参数：
address[in]：void * 型，所要写入的 flash 区域的起始地址。
data[in]：void * 型，所要写入的数据的缓存的地址。
len[in]：int 型，所要写入的数据的长度

返回值：
true – 表示写入成功
false – 表示写入失败

4.4.3.3 eat_flash_read

功能：
从 flash 特定的地址读取特定长度的数据

原型：
bool eat_flash_read(void *buffer, const void *address, unsigned int len);

参数：
buffer[in]：void * 型，读取的 flash 数据的存放的目标地址。
data[in]：void * 型，所要读取的 flash 的源地址。
len[in]：int 型，所要读取的数据的长度

返回值：
true – 表示读取成功
false – 表示读取失败

4.4.3.4 eat_get_flash_block_size

功能：
获取 flash 区块的大小

原型：
unsigned int eat_get_flash_block_size(void)

参数：
无

返回值：
flash 区块的大小

4.4.3.5 eat_fota_flash_erase

功能：
擦除 FOTA flash 区域

原型：
bool eat_fota_flash_erase(const void *address, unsigned int size);
参数：
address[in]：void * 型，所要擦除的 flash 区域的起始地址。
size[in]：int 型，擦除的大小

返回值：
true – 表示擦除成功
false – 表示擦除失败

4.4.3.6 eat_fota_flash_write

功能：
往 FOTA flash 区域相应的地址写入相应长度的数据

原型：
bool eat_fota_flash_write(const void *address, const void *data, unsigned int len)

参数：
address[in]：void * 型，所要写入的 flash 区域的起始地址。
data[in]：void * 型，所要写入的数据的缓存的地址。
len[in]：int 型，所要写入的数据的长度

返回值：
true – 表示写入成功
false – 表示写入失败

4.4.3.7 eat_fota_flash_read

功能：
从 FOTA flash 区域特定的地址读取特定长度的数据

原型：
bool eat_flash_read(void *buffer, const void *address, unsigned int len);

参数：
buffer[in]：void * 型，读取的 flash 数据的存放的目标地址。
data[in]：void * 型，所要读取的 flash 的源地址。
len[in]：int 型，所要读取的数据的长度

返回值：
true – 表示读取成功
false – 表示读取失败

4.4.4 示例

```
uint8_t flash_test_buf[1024] = "test eat flash";
uint8_t flash_store_buf[1024] = "";
void flash_api_test(uint8_t param1, uint8_t param2)
{
    if(1 == param1)
    {
```

```
        eat_trace("GPIO test param1=%d,param2=%d\n",param1,param2);
        if(1 = param2)
        {
            eat_flash_earse(EAT_FLASH_BASE,EAT_FLASH_LENGTH);
        }
        if(2 == param2)
        {
            eat_flash_write(EAT_FLASH_BASE,flash_test_buf,1024);
        }
        if( 3 == param2)
        {
            eat_flash_read(flash_store_buf,EAT_FLASH_BASE,1024);
        }
    }

}
```

4.5 外设接口

4.5.1 GPIO

有 14 个 I / O 引脚可配置为通用 I / O，具体定义在 eat_periphery.h 中。所有引脚均可通过 API 函数在 Embedded AT 下访问。详情请参考 demo 例程，app_demo_gpio.c。

4.5.2 EINT

Embedded AT 支持外部中断输入。 所有 I / O 引脚均可配置为外部中断输入。
但 EINT 不能用于高频率中断检测的目的，以避免模块的不稳定性。详情请参考 demo 例程，app_demo_eint.c。

4.5.3 PWM

有三个 I / O 引脚可配置为 PWM 引脚， 32K 和 13M 时钟源可用。详情请参考 demo 例程，app_demo_pwn.c。

4.5.4 ADC

模拟输入引脚可配置为 ADC。采样周期和计数可以由 API 配置。详情请参考 demo 例程，app_demo_adc.c。

4.5.5 IIC

Embedded AT 提供了一个硬件 IIC 接口，具体配置使用均可使用 API 实现，详情请参考 demo 例程 app_demo_iic.c。

4.5.6 SPI

Embedded AT 提供了一个硬件 SPI 接口，具体的配置和使用均可由 API 实现。详情请参考 demo 例程 app_demo_spi.c。

4.5.7 示例

```
static void eint_sample(void)
{
    hal_eint_config_t eint_config;
    /* Test HAL_EINT_NUMBER_1 */
    irq_num = HAL_EINT_NUMBER_1;
    eat_trace("\r\n ---eint_example begin---\r\n");

    hal_gpio_init(HAL_GPIO_1);
    /* Call hal_pinmux_set_function() to set GPIO pinmux, if EPT tool was not used to configure the related pinmux */
    hal_pinmux_set_function(HAL_GPIO_1, HAL_GPIO_1_EINT1);
    /* Set direction as input and disable pull of corresponding GPIO */
    hal_gpio_set_direction(HAL_GPIO_0, HAL_GPIO_DIRECTION_INPUT);
    hal_gpio_disable_pull(HAL_GPIO_1);


    /* Define the EINT trigger mode by the signal characteristic.
        It supports the following five types.
          a) level and high      // A high-level triggered interrupt, which is triggered when the input signal is at high, and is continuously triggered as long as the input signal is at high.
          b) level and low       // A low-level triggered interrupt, which is triggered when the input signal is
```

```
at low, and is continuously triggered as long as the input signal is at low.
            c) edge and rising      // A rising-edge triggered interrupt, which is triggered when the input signal transitions from low to high.
            d) edge and falling    // A falling-edge triggered interrupt, which is triggered when the input signal transitions from hig to low.
            e) dual edge              // A dual edge triggered interrupt, which is triggered when the input signal transitions from low to high or from high to low.
        */
        eint_config.trigger_mode = HAL_EINT_EDGE_RISING;

        /* The input signal will be ignored if the signal cannot remain stable beyond the de-bounce times setting. The unit of de-bounce time is millisecond. The de-bounce is disabled when the de-bounce time is set to 0. */
        eint_config.debounce_time = 5;

/*This option is used to provide API to mask/unmask dedicated EINT source.*/
#ifdef HAL_EINT_FEATURE_MASK
        /* Mask EINT first to prevent the interrupt misfiring */
        hal_eint_mask(irq_num);
#endif
        hal_eint_init(irq_num, &eint_config);
        hal_eint_register_callback(irq_num, eint_irq_handler, NULL);

/*This option is used to provide API to mask/unmask dedicated EINT source.*/
#ifdef HAL_EINT_FEATURE_MASK
        /* Unmask EINT    */
        hal_eint_unmask(irq_num);
#endif

        eat_trace("\r\n ---eint_example finished!!!---\r\n");
}
```