

API

MC300 API 参考手册

2016-10-10

目录

1. Linux 标准 C 函数.....	6
2. 系统错误码定义	7
3. AT 命令 API	9
● hfat_get_words.....	9
● hfat_send_cmd.....	9
● hfat_uart_send	10
4. DEBUG API.....	11
● HF_Debug.....	11
● hfdbg_get_level.....	11
● hfdbg_set_level.....	12
5. GPIO 控制 API	14
● hfgpio_configure_fpin	14
● hfgpio_fconfigure_get	15
● hfgpio_fpin_add_feature.....	16
● hfgpio_fpin_clear_feature.....	16
● hfgpio_fpin_is_high.....	17
● hfgpio_fset_out_high.....	18
● hfgpio_fset_out_low	18
6. WiFi API	20
● hfsmtlk_start.....	20
● hfsmtlk_stop.....	20

●	hfwifi_scan	21
7.	串口 API	23
●	hfuart_send	23
8.	定时器 API	24
●	hftimer_start	24
●	hftimer_create	24
●	hftimer_change_period	25
●	hftimer_delete	26
●	hftimer_get_timer_id	26
●	hftimer_stop	27
9.	多任务 API	28
●	PROCESS	28
10.	网络 API	29
●	hfnet_start_uart	29
●	hfnet_start_socketa	29
●	hfnet_start_socketb	30
●	hfnet_tcp_listen	31
●	hfnet_tcp_unlisten	32
●	hfnet_tcp_close	32
●	hfnet_tcp_connect	33
●	hfnet_tcp_disconnect	34
●	hfnet_tcp_send	34

●	hfnet_udp_create	35
●	hfnet_udp_close	36
●	hfnet_udp_sendto	36
11.	系统函数	38
●	hfmem_free	38
●	hfmem_malloc	38
●	hfmem_realloc	39
●	hfsys_get_reset_reason	39
●	hfsys_get_run_mode	40
●	hfsys_get_time	41
●	hfsys_nvm_read	41
●	hfsys_nvm_write	42
●	hfsys_register_system_event	43
●	hfsys_reload	44
●	hfsys_reset	44
●	hfsys_softreset	45
●	hfsys_switch_run_mode	45
12.	用户 Flash API	47
●	hfuf flash_erase_page	47
●	hfuf flash_read	47
●	hfuf flash_write	48
13.	用户文件操作 API	50

●	hffile_userbin_read	50
●	hffile_userbin_size	50
●	hffile_userbin_write	51
●	hffile_userbin_zero	51
14.	自动升级 API	53
●	hfupdate_complete	53
●	hfupdate_start	53
●	hfupdate_write_file	54
附录 A :	硬件定时器	56
附录 B :	GPIO 中断	60

1. LINUX 标准 C 函数

HSF MC300 兼容标准 c 库的函数，例如内存管理，字符串，时间，标准输入输出等，有关函数的说明请参考标准 c 库函数说明。

注：

在 HSF MC300 系统中不建议直接调用 libc 中的内存管理函数，可能链接将不通过,内存管理函数当前只提供 3 个函数，参考 hfmem_malloc、hfmem_free、hfmem_realloc。本 API 文档适用于采用 HF-MC300 SOC 芯片的系列模组(HF-LPB120、HF-LPT120、HF-LPT120G、HF-LPT220、HF-LPB125) 和 HF-SIP120 芯片。

2. 系统错误码定义

API 函数返回值（特别说明除外）规定，成功 HF_SUCCESS，或者>0、失败<0。错误码为 4Bytes 有符号整数，返回值为错误码的负数。31-24bit 为模块索引，23-8 保留，7-0，为具体的错误码。

```
#define MOD_ERROR_START(x) ((x < 16) | 0)
/* Create Module index */
#define MOD_GENERIC 0
/** HTTPD module index */
#define MOD_HTTPDE 1
/** HTTP-CLIENT module index */
#define MOD_HTTPC 2
/** WPS module index */
#define MOD_WPS 3
/** WLAN module index */
#define MOD_WLAN 4
/** USB module index */
#define MOD_USB 5

/*0x70~0x7f user define index*/
#define MOD_USER_DEFINE (0x70)
/* Globally unique success code */
#define HF_SUCCESS 0

enum hf_errno {
/* First Generic Error codes */
    HF_GEN_E_BASE = MOD_ERROR_START(MOD_GENERIC),
    HF_FAIL,
    HF_E_PERM, /* Operation not permitted */
    HF_E_NOENT, /* No such file or directory */
    HF_E_SRCH, /* No such process */
    HF_E_INTR, /* Interrupted system call */
    HF_E_IO, /* I/O error */
    HF_E_NXIO, /* No such device or address */
    HF_E_2BIG, /* Argument list too long */
    HF_E_NOEXEC, /* Exec format error */
    HF_E_BADF, /* Bad file number */
    HF_E_CHILD, /* No child processes */
    HF_E_AGAIN, /* Try again */
}
```

```
HF_E_NOMEM, /* Out of memory */
HF_E_ACCES, /* Permission denied */
HF_E_FAULT, /* Bad address */
HF_E_NOTBLK, /* Block device required */
HF_E_BUSY, /* Device or resource busy */
HF_E_EXIST, /* File exists */
HF_E_XDEV, /* Cross-device link */
HF_E_NODEV, /* No such device */
HF_E_NOTDIR, /* Not a directory */
HF_E_ISDIR, /* Is a directory */
HF_E_INVAL, /* Invalid argument */
HF_E_NFILE, /* File table overflow */
HF_E_MFILE, /* Too many open files */
HF_E_NOTTY, /* Not a typewriter */
HF_E_TXTBSY, /* Text file busy */
HF_E_FBIG, /* File too large */
HF_E_NOSPC, /* No space left on device */
HF_E_SPIPE, /* Illegal seek */
HF_E_ROFS, /* Read-only file system */
HF_E_MLINK, /* Too many links */
HF_E_PIPE, /* Broken pipe */
HF_E_DOM, /* Math argument out of domain of func */
HF_E_RANGE, /* Math result not representable */
HF_E_DEADLK, /*Resource deadlock would occur*/
};
```

头文件：

hferrno.h

3. AT 命令 API

● hfat_get_words

函数原型：

```
int hfat_get_words((char *str,char *words[],int size);
```

说明：

获取 AT 命令或者响应的每一个参数值

参数：

str：指向 AT 命令请求或者响应;对应的 RAM 地址一定可读写；
words：保存每一个参数值；
size：word 的个数

返回值：

<=0：str 对应的字符串不是正确的 AT 命令或者非法响应;
>0：对应字符串中包含 Word 的个数；

备注：

AT 命令以" , " , " = " , " " " \r\n" 分隔；

例子：

Example/attest.c

头文件：

hfath.h

● hfat_send_cmd

函数原型：

```
int hfat_send_cmd(char *cmd_line,int cmd_len,char *rsp,int len) ;
```

说明：

发送 AT 命令，结果返回到指定的 buffer

参数：

cmd_line: 包含 AT 命令字符串；
格式为 AT+CMD_NAME[=][arg ,]...[argn];
cmd_len: cmd_line 的长度，包括结束符；

rsp: 保存 AT 命令执行结果的 buffer;

Len: rsp 的长度;

返回值:

HF_success: 设置成功, HF_FAIL: 执行失败

备注:

函数执行和通过串口发送 AT 命令一样, 当前不支持“AT+H”和“AT+WSCAN”;wifi 扫描可以参考 hfwifi_scan, AT 命令执行结果保存在 rsp 中, rsp 是一个字符串, 具体格式请参考串口 AT 命令集帮助文档; 通过这个函数可以获取设置系统配置。

注意这个函数放送不了通过 user_define_at_cmds_table 扩展的 AT 命令, 因为自己扩展的 AT 命令可以直接调用, 不需要在通过发送 AT 命令实现, 如果用户通过 user_define_at_cmds_table 扩展了已经存在的 AT 命令例如“AT+VER”,

如果在程序中发送 hfat_send_cmd(“AT+VER\r\n”, sizeof(“AT+VER\r\n”), rsp, 64); 返回的将是自带的 AT+VER 而不是自己扩展的。

例子:

参考 example 下的 attest.c

头文件:

hfat.h

● hfat_uart_send

函数原型:

```
int hfat_uart_send(hfuart_handle_t handle, char *data, uint32_t bytes);
```

说明:

发送数据到串口 0(HFUART0 句柄)或者串口 1(HFUART1 句柄), 功能同 hfuart_send.

4. DEBUG API

● HF_Debug

函数原型：

```
void HF_Debug(int debug_level , const char *format , ...);
```

说明：

输出调试信息到串口

参数：

Debug_level:调试等级，可以为

```
#define DEBUG_LEVEL_LOW 1
```

```
#define DEBUG_LEVEL_MID 2
```

```
#define DEBUG_LEVEL_HI 3
```

或者其他更大值，配合 hfdbg_set_level 设置的调式等级可以只输出设置的等级以上的 log 信息，log 信息输出需要先使能。

Format: 格式化输出，和 printf 一样，内容最多 250 字节，若内容超过此值，请调用多次进行打印。

返回值：

无

备注：

AT+NDBGL=X,Y 可使能 debug 信息输出，X 代表调试等级(0:关闭)，Y 代表串口号(0：串口 0，1:串口 1)，推荐调试信息输出到串口 1（串口 1 引脚请详见各模块手册），串口 0 用于正常交互通讯。程序发布后要动态打开调试，就可以用 AT+NDBGL 命令打开，不需要调试的时候用 AT+NDBGL=0 关闭。

例子：

无

头文件：

hfdebug.h

● hfdbg_get_level

函数原型：

```
int hfdbg_get_level();
```

说明：

获取当前设置的调试等级

参数：

无

返回值：

返回当前设置的调试等级

备注：

无

例子：

无

头文件：

hfdebug.h

● hfdbg_set_level

函数原型：

```
void hfdbg_set_level (int debug_level);
```

说明：

设置调试信息输出等级，或者关闭调试信息输出

参数：

debug_level:调试级别，可以为

0：关闭 debug 信息输出

```
#define DEBUG_LEVEL_LOW 1
```

```
#define DEBUG_LEVEL_MID 2
```

```
#define DEBUG_LEVEL_HI 3
```

返回值：

无

备注：

推荐使用串口 AT+NDBGL 命令动态使能或关闭 debug 信息输出，这样需要查看 log 的时候可以随时查看，而不需要修改程序。

例子：

无

头文件：

hfdebug.h

5. GPIO 控制 API

● hfgpio_configure_fpin

函数原型：

```
int hfgpio_configure_fpin (int fid,int flag);
```

说明：

根据 fid(功能码) , 配置对应的 PIN 脚

参数：

fid 功能码

```
enum HF_GPIO_FUNC_E
```

```
{  
    //fix/////////////////////////////////  
    HFGPIO_F_JTAG_TCK=0,  
    HFGPIO_F_JTAG_TDO=1,  
    HFGPIO_F_JTAG_TDI,  
    HFGPIO_F_JTAG_TMS,  
    HFGPIO_F_USBDP,  
    HFGPIO_F_USBDM,  
    HFGPIO_F_UART0_TX,  
    HFGPIO_F_UART0_RTS,  
    HFGPIO_F_UART0_RX,  
    HFGPIO_F_UART0_CTS,  
    HFGPIO_F_SPI_MISO,  
    HFGPIO_F_SPI_CLK,  
    HFGPIO_F_SPI_CS,  
    HFGPIO_F_SPI_MOSI,  
    HFGPIO_F_UART1_TX,  
    HFGPIO_F_UART1_RTS,  
    HFGPIO_F_UART1_RX,  
    HFGPIO_F_UART1_CTS,  
    ///////////////////////////////////  
    HFGPIO_F_NLINK,  
    HFGPIO_F_NREADY,  
    HFGPIO_F_NRELOAD,  
    HFGPIO_F_SLEEP_RQ,  
    HFGPIO_F_SLEEP_ON,  
    HFGPIO_F_WPS,  
    HFGPIO_F_IR,  
    HFGPIO_F_RESERVE2,  
    HFGPIO_F_RESERVE3,  
    HFGPIO_F_RESERVE4,  
}
```

```
    HFGPIO_F_RESERVE5,  
    HFGPIO_F_USER_DEFINE  
};
```

也可以为用户自定义功能吗，用户自定义功能码从 HFGPIO_F_USER_DEFINE 开始.
flags:PIN 脚属性，可以为下面一个或者多个值进行“|”运算

HFPIO_DEFAULT	默认
HFM_IO_TYPE_INPUT	输入模式
HFM_IO_OUTPUT_0	输出为低电平
HFM_IO_OUTPUT_1	输出为高电平

返回值：

HF_SUCCESS:设置成功，HF_E_INVALID: fid 非法,或者它对应的 PIN 脚非法,
HF_E_ACCESS:对应的 PIN 不具备要设置的属性(flags)，例如 HFGPIO_F_JTAG_TCK 对应的
PIN 脚是一个外设 PIN 脚，不是 GPIO 脚，不能配置 HFPIO_DEFAULT 以外的任何属性.

备注：

在设置之前，先要清楚功能码对应的 PIN 脚的属性，每个 PIN 脚的属性请查看相关数
据手册，如果给一个 PIN 配置它不具备的属性，将返回 HF_E_ACCESS 错误。

例子：

gpiotest.c

头文件：

hfgpio.h

● hfgpio_fconfigure_get

函数原型：

```
int HSF_API hfgpio_fconfigure_get(int fid);
```

说明：

获取功能码对应的 PIN 脚对应的属性值;

参数：

fid: 功能码，参考 HF_GPIO_FUNC_E,也可以为用户自定义功能吗。

返回值：

成功返回 PIN 对应的属性值，属性值可以参考 hfgpio_configure_fpin，HF_E_INVALID:
fid 非法,或者它对应的 PIN 脚非法

备注：

无

例子：

gpiotest.c

头文件：

hfgpio.h

● hfgpio_fpin_add_feature

函数原型：

hfgpio_fpin_add_feature(int fid,int flags);

说明：

对功能码对应的 PIN 脚添加属性值;

参数：

fid: 功能码，参考 HF_GPIO_FUNC_E,也可以为用户自定义功能码;

flags:参考 hfgpio_configure_fpin flags;

返回值：

HF_SUCCESS:设置成功， HF_E_INVALID: fid 非法,或者它对应的 PIN 脚非法

备注：

无

例子：

gpiotest.c

头文件：

hfgpio.h

● hfgpio_fpin_clear_feature

函数原型：

int HSF_API hfgpio_fpin_clear_feature (int fid,int flags);

说明：

清除功能码对应的 PIN 脚的一个或者多个属性值;

参数：

fid: 功能码，参考 HF_GPIO_FUNC_E,也可以为用户自定义功能码;

flags:参考 hfgpio_configure_fpin flags;

返回值：

HF_SUCCESS:设置成功， HF_E_INVAL: fid 非法,或者它对应的 PIN 脚非法

备注：

无

例子：

gpiotest.c

头文件：

hfgpio.h

● hfgpio_fpin_is_high

函数原型：

```
int hfgpio_fpin_is_high(int fid);
```

说明：

判断功能码对应的 PIN 脚是否为高电平;

参数：

fid: 功能码 ,参考 HF_GPIO_FUNC_E,也可以为用户自定义功能码,fid 对应的 PIN 脚一定具有 F_GPO 或者 F_GPI 属性。

返回值：

如果对应的 PIN 脚为低电平返回 0，如果为高电平返回 1;如果小于 0 说明 fid 对应的 PIN 脚非法.

备注：

无

例子：

参考 example 下的 gpiotest.c

头文件：

hfgpio.h

● hfgpio_fset_out_high

函数原型：

```
int hfgpio_fset_out_high(int fid);
```

说明：

把功能码对应的 PIN 脚，设置为输出高电平

参数：

fid:参考 HF_GPIO_FUNC_E，也可以为用户自定义功能码。

返回值：

HF_SUCCESS:设置成功，HF_E_INVALID: fid 非法,或者它对应的 PIN 脚非法,

HF_FAIL:设置失败；HF_E_ACCESS:对应的 PIN 属性不支持输出

备注：

这个函数等价于 hfgpio_configure_fpin(fid,HFM_IO_OUTPUT_1|HFPIO_DEFAULT);

例子：

gpiotest.c

头文件：

hfgpio.h

● hfgpio_fset_out_low

函数原型：

```
int hfgpio_fset_out_low(int fid);
```

说明：

把功能码对应的 PIN 脚设置为输出低电平;

参数：

fid: 功能码，参考 HF_GPIO_FUNC_E,也可以为用户自定义功能码。

返回值：

HF_SUCCESS:设置成功 , HF_E_INVALID: fid 非法,或者它对应的 PIN 脚非法

备注:

这个函数等价于 `hfgpio_configure_fpin(fid,HFM_IO_OUTPUT_0|HFPIO_DEFAULT);`

例子:

`gpiotest.c`

头文件:

`hfgpio.h`

6. WIFI API

● hfsmtlk_start

函数原型：

```
int HSF_API hfsmtlk_start(void);
```

说明：

启动 smartlink

参数：

无

返回值：

成功返回 HF_SUCCESS,否则失败

备注：

调用这个函数后程序马上软重启。

例子：

无

头文件：

hfsmtlk.h

● hfsmtlk_stop

函数原型：

```
int HSF_API hfsmtlk_stop(void);
```

说明：

停止 smartlink.

参数：

无

返回值：

成功返回 HF_SUCCESS,否则失败

备注：

无

例子：

无

头文件：

hfsmtlk.h

● hfwifi_scan

函数原型：

```
int HSF_API hfwifi_scan(hfwifi_scan_callback_t p_callback);
```

说明：

扫描附近的存在的 AP。

参数：

hfwifi_scan_callback_t:设备扫描到周围的 AP 的时候 ,通过这个回调告诉用户这个 AP 的具体信息。

```
typedef int (*hfwifi_scan_callback_t)( PWIFI_SCAN_RESULT_ITEM );
typedef struct _WIFI_SCAN_RESULT_ITEM
{
    uint8_t auth; //认证方式
    uint8_t encry; //加密方式
    uint8_t channel; //工作信道
    uint8_t rssi; //信号强度
    char ssid[32+1]; //AP 的 SSID
    uint8_t mac[6]; //AP 的 mac 地址
    int rssi_dbm; //信号强度的 dBm 值
    int sco;
}WIFI_SCAN_RESULT_ITEM,*PWIFI_SCAN_RESULT_ITEM;
```

```
#define WSCAN_AUTH_OPEN 0
#define WSCAN_AUTH_SHARED 1
#define WSCAN_AUTH_WPA2PSK 2
#define WSCAN_AUTH_WPA2PSK 3
#define WSCAN_AUTH_WPA2PSK 4
#define WSCAN_ENC_NONE 0
#define WSCAN_ENC_WEP 1
```

```
#define WSCAN_ENC_TKIP 2
#define WSCAN_ENC_AES 3
#define WSCAN_ENC_TKIPAES 4
```

返回值：

成功返回 HF_SUCCESS,否则失败。

备注：

当接收到空指针回调时说明扫描结束。

例子：

wifitest.c

头文件：

hfwifi.h

7. 串口 API

● hfuart_send

函数原型：

```
int HSF_API hfuart_send(hfuart_handle_t huart, char *data, uint32_t bytes,  
uint32_t timeouts);
```

说明：

发送数据到串口

参数：

huart: 串口设备对象，可选 HFUART0 或者 HFUART1（串口 0 或者串口 1）

data: 要发送的数据的缓存区

bytes: 发送数据的长度

timeouts: 超时时间，暂时无效值，默认填 0 即可

返回值：

成功返回为实际发送的数据，失败返回错误码；

备注：

无

例子：

无

头文件：

hfuart.h

8. 定时器 API

● hftimer_start

函数原型：

```
int HSF_API hftimer_start(hftimer_handle_t htimer);
```

说明：

启动一个定时器

参数：

htimer:由 hftimer_create 创建；

返回值：

成功返回 HF_SUCCESS,否则返回 HF_FAIL;

备注：

无

例子：

参考 example 下的 timertest.c

头文件：

hftimer.h

● hftimer_create

函数原型：

```
hftimer_handle_t HSF_API hftimer_create(const char *name, int32_t period,  
bool auto_reload, uint32_t timer_id, hf_timer_callback p_callback, uint32_t flags );
```

说明：

创建一个定时器

参数：

name: 定时器的名称

period:定时器触发的周期，以 ms 为单位;

auto_reload: 指定自动还是手动，如果为 true，只需要调用一次 hftimer_start 一次，定时器触发后,不需要再次调用 hftimer_start;如果为 false,触发后要再次触发要再次调用 hftimer_start.

timer_id: 指定一个唯一 ID，代表这个定时器，当多个定时器使用一个回调函数的时候可以用这个值来区分定时器;

flags: 当前可以为 0。

返回值：

函数执行成功，放回指向一个定时器对象的指针,否则返回 NULL;

备注：

定时器创建后，不会马上启动，直到调用 hftimer_start 定时器才会启动.如果制定定时器为手动，定时器触发后要想再次触发要重新调用 hftimer_start,如果是自动不需要，定时器会在下一个额周期自动触发。

例子：

timertest.c

头文件：

hftimer.h

● hftimer_change_period

函数原型：

```
void HSF_API hftimer_change_period(hftimer_handle_t htimer,int32_t new_period);
```

说明：

修改定时器的周期

参数：

htimer:由 hftimer_create 创建；

new_period: 新的周期，单位 ms.如果创建的定时器为硬件定时器，单位为微秒

返回值：

无

备注：

修改定时器的周期，调用这个函数后，定时器将以新的周期运行.

例子：

参考 example 下的 timertest.c

头文件：

hftimer.h

● hftimer_delete

函数原型：

```
void HSF_API hftimer_delete(hftimer_handle_t htimer);
```

说明：

销毁一个定时器

参数：

htimer:要删除的定时器，由 hftimer_create 创建；

返回值：

无

例子：

参考 example 下的 timertest.c

头文件：

hftimer.h

● hftimer_get_timer_id

函数原型：

```
uint32_t HSF_API hftimer_get_timer_id( hftimer_handle_t htimer );
```

说明：

获取定时器的 ID

参数：

htimer:由 hftimer_create 创建；

返回值：

成功返回定时器的 ID，由 hftimer_create 指定.失败返回 HF_FAIL;

备注：

这个函数一般在定时器回调的时候调用，又来区分多个 timer 使用一个回调函数的情况

况。

例子：

参考 example 下的 timertest.c

头文件：

hftimer.h

● hftimer_stop

函数原型：

```
void HSF_API hftimer_stop(hftimer_handle_t htimer);
```

说明：

停止一个定时器

参数：

htimer:由 hftimer_create 创建；

返回值：

无

备注：

调用这个函数后，定时器将不再触发，直到再次调用 hftimer_start;

例子：

参考 example 下的 timertest.c

头文件：

hftimer.h

9. 多任务 API

● PROCESS

MC300 HSF 采用 Contiki 操作多任务系统，系统中没有线程概念，全多通过任务进行调度，在主任务函数中需要注意：

- 1.不可使用 switch/case 语句；
- 2.慎用局部变量，主任务函数运行过程中会退出，局部变量会被释放；

PROCESS(name, strname)

声明进程 name 的主体函数，并定义一个进程 name;

AUTOSTART_PROCESSES(...)

定义一个进程指针数组 autostart_processe;

PROCESS_THREAD(name, ev, data)

进程 name 的定义或声明，取决于宏后面是";"还是"{}";

PROCESS_BEGIN()

进程的主体函数从这里开始;

PROCESS_EXIT()

进程的主体函数从这里结束;

PROCESS_WAIT_EVENT_UNTIL(c)

等待相应的消息；

int process_post(struct process *p, process_event_t ev, void* data);

发送消息给 process;

void process_post_synch(struct process *p, process_event_t ev, void* data);

发送消息给 process，并立刻进行任务切换;

例子：

processtest.c，更多用法可以参考 contiki 系统使用方法；

头文件：

hsf.h

10. 网络 API

● hfnet_start_uart

函数原型：

```
int hfnet_start_uart(uint32_t uxpriority,hfnet_callback_t p_uart_callback);
```

说明：

启动 HSF 自带 uart 串口收发控制服务。

参数：

uxpriority:uart 服务对应的线程的优先级;请参考 hfthread_create 参数 uxpriority
p_uart_callback: 串口回调函数, 可选, 如果不需要请设置为 NULL,当串口收到数据的时候调用,回调函数的定义和参数请参考 hfnet_start_socketa;

返回值：

成功返回 HF_SUCCESS, HF_FAIL 表示失败

备注：

当串口接收数据的时候,如果 p_uart_callback 不为 NULL,先调用 p_uart_callback,如果工作在透传模式,把接收的数据发给 socketa,socketb 服务(如果这两个服务器存在),如果工作在命令模式把接收到的命令交给命令解析程序。

在透传模式下,用户可以通过这个回调函数和 socketa,socketb 服务的回调,实现数据的加解密,或者二次处理;在命令模式下,用户可以通过回调实现自定义 AT 命令名称和格式;

例子：

参考 example 下的 callbacktest.c

头文件：

hfnet.h

● hfnet_start_socketa

函数原型：

```
int hfnet_start_socketa(uint32_t uxpriority,hfnet_callback_t p_callback);
```

说明：

启动 HSF 自带 socketa 服务。

参数：

uxpriority: socketa 服务优先级，请参考 hfthread_create 参数 uxpriority;

p_callback：回调函数，可选，如果不需要回调把这个值设置为 NULL,当 socketa 服务接收到数据包或者状态发送变化的时候触发;

int socketa_recv_callback_t(uint32_t event,void *data,uint32_t len,uint32_t buf_len);

event:事情 ID ；

data:指向接收数据的 buffer,用户可以在回调函数中修改 buffer 里面的值；当工作在 UDP 模式的时候 data+len 之后 6 个 bytes 放置的为发送端的 4Bytes ip 地址和 2 Bytes 端口号，如果是 socketa 工作在 TCP 服务器端模式，data+len 后面 4 个 Bytes 为客户端的 cid，可以通过 hfnet_socketa_get_client 或者详细信息。

len:接收到数据的长度;

buf_len:data 指向的 buffer 的实际长度,这个值大于等于 len;

回调函数返回值，为用户处理过数据的长度，如果用户不对数据进行修改，只是读，放回值应该等于 len;

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败

备注：

当 socketa 服务接收到网络发过来的数据的时候，调用 p_callback，再把 p_callback 处理的值发到串口，用户可以利用 p_callback 对接收的数据进行解析，或者二次处理，例如加密解密等,把处理的数据返回给 socketa 服务。

例子：

参考 example 下的 callbacktest.c

头文件：

hfnet.h

● hfnet_start_socketb**函数原型：**

int hfnet_start_socketb(uint32_t uxpriority,hfnet_callback_t p_callback);

说明：

启动 HSF 自带 socketb 服务。

参数：

uxpriority:socketb 服务对应的线程的优先级;请参考 hfthread_create 参数 uxpriority
p_callback:可选，不使用回调传 NULL,请参考 hfnet_start_socketa

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败

备注：

无

例子：

callbacktest.c

头文件：

hfnet.h

● hfnet_tcp_listen

函数原型：

```
int HSF_API hfnet_tcp_listen(struct tcp_socket *socket);
```

说明：

建立 TCP Serer，等待远程用户接入。

参数：

socket：TCP Socket 结构，其中需要指定项为：
listen_port：listen 端口;
recv_callback：接收数据回调;
accept_callback：接入连接回调;
send_callback：发送完成回调;
close_callback：连接关闭回调;
recv_data_maxlen：接收最长字节（如果不设置，则为缺省值 2048）;

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

建立成功后，如果有远端 TCP 连接，则回调 accept_callback 会触发，并告知目前使用

的 Socket 索引，用户可以在此 Socket 上进行数据发送。

例子：

tcpservertest.c

头文件：

hfnet.h

● hfnet_tcp_unlisten

函数原型：

```
int HSF_API hfnet_tcp_unlisten(struct tcp_socket *socket);
```

说明：

关闭 TCP Serer。

参数：

socket：TCP Socket 结构，应与创立 TCP Server 时所用结构相同

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

关闭后，相关资源释放，不会再工作，如要继续使用，需调用 hfnet_tcp_listen 重新建立一个 Server。

例子：

tcpservertest.c

头文件：

hfnet.h

● hfnet_tcp_close

函数原型：

```
int HSF_API hfnet_tcp_close(NETSOCKET socket_id);
```

说明：

关闭连接到 TCP Serer 的 TCP Client。

参数：

socket_id : TCP Client 的 Socket 索引;

返回值：

成功返回 HF_SUCCESS , HF_FAIL 表示失败。

备注：

关闭后，TCP Server 任然可以接受新的 TCP 连接请求。

例子：

tcpservertest.c

头文件：

hfnet.h

● hfnet_tcp_connect

函数原型：

NETSOCKET HSF_API hfnet_tcp_connect(struct tcp_socket *socket);

说明：

建立一个 TCP 连接 (TCP Client) 。

参数：

socket : TCP Socket 结构，其中需要指定项为：

l_port : 本地端口;

r_ip : 需要连接的远端 IP 地址;

r_port : 需要连接的远端 IP 端口;

recv_callback : 接收数据回调;

connect_callback : 连接回调;

send_callback : 发送完成回调;

close_callback : 连接关闭回调;

recv_data_maxlen : 接收最长字节 (如果不设置，则为缺省值 2048) ;

返回值：

创建好的 Socket 索引

备注：

建立成功后，用户可以在创建好的 Socket 索引上进行数据发送。

例子：

tcpclienttest.c

头文件：

hfnet.h

● hfnet_tcp_disconnect

函数原型：

```
int HSF_API hfnet_tcp_disconnect(NETSOCKET socket_id);
```

说明：

断开 TCP 连接。

参数：

socket_id：正在使用的 Socket 索引；

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

无

例子：

tcpclienttest.c

头文件：

hfnet.h

● hfnet_tcp_send

函数原型：

```
int HSF_API hfnet_tcp_send(NETSOCKET socket_id, char *data, unsigned short datalen);
```

说明：

发送 TCP 数据。

参数：

socket_id：正在使用的 Socket 索引
data：发送数据
datalen：发送数据长度；

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

发送成功只是代表数据已计入发送队列中，正式的发送成功后，系统会调用 send_callback 回调函数；
数据在发送成功之前不能释放，系统不会缓存数据，只会引用。

例子：

参考 example 下的 tcpclienttest.c

头文件：

hfnet.h

● hfnet_udp_create

函数原型： NETSOCKET HSF_API hfnet_udp_create(struct udp_socket *socket);

说明：

建立一个 UDP。

参数：

UDP Socket 结构，其中需要指定项为：
l_port：本地端口
recv_callback：接收数据回调
connect_callback：连接回调
recv_data_maxlen：接收最长字节（如果不设置，则为缺省值 2048）；

返回值：

创建好的 Socket 索引。

备注：

建立成功后，用户可以在创建好的 Socket 索引上进行数据发送。

例子：

参考 example 下的 udptest.c

头文件：

hfnet.h

● hfnet_udp_close

函数原型：

```
int HSF_API hfnet_udp_close(NETSOCKET socket_id);
```

说明：

关闭一个 UDP。

参数：

socket_id：正在使用的 Socket 索引。

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

无

例子：

参考 example 下的 udptest.c

头文件：

hfnet.h

● hfnet_udp_sendto

函数原型：

```
int HSF_API hfnet_udp_sendto(NETSOCKET socket_id, char *data, unsigned short datalen, uip_ipaddr_t *peeraddr, unsigned short peerport);
```

说明：

发送 UDP 数据。

参数：

socket_id：正在使用的 Socket 索引
data：发送数据
datalen：发送数据长度
peeraddr：接收端的 IP 地址
peerport：接收端的端口。

返回值：

成功返回 HF_SUCCESS，HF_FAIL 表示失败。

备注：

发送成功代表数据已发送出去，不会另外调用发送回调；
数据在发送成功后即可释放，系统不会缓存数据，只会引用。

例子：

参考 example 下的 udptest.c

头文件：

hfnet.h

11. 系统函数

● hfmem_free

函数原型：

```
void HSF_API hfmem_free(void *pv);
```

说明：

释放由 hfsys_malloc 分配的内存

参数：

pv:指向要释放内存地址;

返回值：

无

备注：

不要使用 libc 中的 free 函数.

例子：

无

头文件：

hfsys.h

● hfmem_malloc

函数原型：

```
void *hfmem_malloc(size_t size)
```

说明：

动态分配内存

参数：

size:分配内存的大小

返回值：

如果为 NULL,说明系统没有空闲的内存；成功返回内存的地址;

备注:

不要使用 libc 中的 malloc 函数

头文件:

hfsys.h

● hfmem_realloc

函数原型:

```
void HSF_API *hfmem_realloc(void *pv,size_t size) ;
```

说明:

重新分配内存

参数:

pv:指向原先用 hfmem_malloc 分配地址的指针;

size:重新分配内存的大小

返回值:

无

备注:

请参考 libc 的 realloc，程序中不能直接调用 realloc 的函数，只能用这个 API。

例子:

无

头文件:

hfsys.h

● hfsys_get_reset_reason

函数原型:

```
uint32_t HSF_API hfsys_get_reset_reason (void);
```

说明:

获取模块重启的原因

参数：

无

返回值：

返回模块重启的原因,可以是下面表中的一个或者多个（做或运算）

HFSYS_RESET_REASON_NORMAL	模块是由于断电再启动
HFSYS_RESET_REASON_ERESET	模块是由于硬件看门狗和外部 Reset 按键重启
HFSYS_RESET_REASON_IRESET0	模块是由于程序内部调用 hfsys_softreset 重启(软件看门狗重启, 或者程序段错误, 内存访问错误)
HFSYS_RESET_REASON_IRESET1	模块是由于内部调用 hfsys_reset 重启
HFSYS_RESET_REASON_WPS	模块是由于 WPS 而重启
HFSYS_RESET_REASON_SMARTLINK_START	模块是由于 SmartLink 启动而重启
HFSYS_RESET_REASON_SMARTLINK_OK	模块是由于 SmartLink 配置成功而重启

备注：

一般在入口函数调用这个函数来判断一下，这次启动是重启，还是断电启动，以及重启的原因，根据不同的重启原因来进行恢复行的操作。

例子：

无

头文件：

hfsys.h

● hfsys_get_run_mode

函数原型：

```
int hfsys_get_run_mode()
```

说明：

获取系统当前运行模式

参数：

无

返回值：

返回当前运行的模式,运行模式可以为下面的值:

```
enum HFSYS_RUN_MODE_E
{
    HFSYS_STATE_RUN_THROUGH=0,
    HFSYS_STATE_RUN_CMD=1,
    HFSYS_STATE_MAX_VALUE
};
```

头文件：

hfsys.h

● hfsys_get_time

函数原型：

```
uint32_t HSF_API hfsys_get_time (void);
```

说明：

获取系统从启动到现在所花的时间（毫秒）

参数：

无

返回值：

返回系统运行到现在所花的毫秒数

备注：

无

例子：

无

头文件：

hfsys.h

● hfsys_nvm_read

函数原型：

```
int HSF_API hfsys_nvm_read(uint32_t nvm_addr, char* buf, uint32_t length);
```

说明：

从 NVM 里面读数据

参数：

nvm_addr: NVM 的地址，可以为(0-99);
buf: 保存从 NVM 读到数据的缓存区；
length: 长度和 nvm_addr 的和小于 100;

返回值：

成功返回 HF_SUCCESS，否则返回小于零。

备注：

当模块重启，软重启，NVM 的数据不会被清除，提供了 100Bytes 的 NVM，如果模块断电 NVM 的数据会被清除。

例子：

无

头文件：

hfsys.h

● hfsys_nvm_write

函数原型：

```
int HSF_API hfsys_nvm_write(uint32_t nvm_addr, char* buf, uint32_t length);
```

说明：

向 NVM 里面写数据

参数：

nvm_addr: NVM 的地址，可以为(0-99);
buf: 保存从 NVM 读到数据的缓存区;
length: 长度和 nvm_addr 的和小于 100;

返回值：

成功返回 HF_SUCCESS，否则返回小于零

备注：

当模块重启，软重启，NVM 的数据不会被清除，提供了 100Bytes 的 NVM，如果模块断电 NVM 的数据会被清除

例子：

无

头文件：

hfsys.h

● hfsys_register_system_event

函数原型：

```
int HSF_API hfsys_register_system_event( hfsys_event_callback_t p_callback );
```

说明：

注册系统事件回调

参数：

p_callback:指向用户制定的系统事情回调函数的地址;

返回值：

如果返回 HF_SUCCESS，系统按照默认动作处理这个事情，否则返回小于零，这个时候系统不会对事情进行相应的处理

备注：

在回调函数中不能调用有延时的 API 函数，不能延时，处理后应该立刻返回，否则会影响系统正常运行。当前支持的系统事情有：

HFE_WIFI_STA_CONNECTED	当 STA 连接成功的时候触发
HFE_WIFI_STA_DISCONNECTED	当 STA 断开的时候触发
HFE_CONFIG_RELOAD	当系统执行 reload 的时候触发
HFE_DHCP_OK	当 STA 连接成功,并且 DHCP 拿到地址的时候触发
HFE_SMTLK_OK	当 SMTLK 配置拿到密码的时候触发，默认动作重启，如果回调返回不是 HF_SUCCESS，将不会重启，用户可以手动重启。

例子：

参考 example 下的 tcpclienttest.c

头文件：

hfsys.h

● **hfsys_reload**

函数原型：

void HSF_API hfsys_reload() ;

说明：

系统恢复成出厂设置

参数：

无

返回值：

无

备注：

无

例子：

无

头文件：

hfsys.h

● **hfsys_reset**

函数原型： void HSF_API hfsys_reset(void);

说明：

重启系统,IO 电平不保持

参数：

无

返回值：

无

备注：

无

例子：

无

头文件：

hfsys.h

● hfsys_softreset

函数原型：

```
void HSF_API hfsys_softreset(void);
```

说明：

软重启系统，IO 电平保持

参数：

无

返回值：

无

备注：

无

例子：

无

头文件：

hfsys.h

● hfsys_switch_run_mode

函数原型：

```
int hfsys_switch_run_mode(int mode);
```

说明：

切换系统运行模式

参数：

mode:要切换的运行模式，系统当前支持的运行模式有

enum HFSYS_RUN_MODE_E

```
{  
    HFSYS_STATE_RUN_THROUGH=0,  
    HFSYS_STATE_RUN_CMD=1,  
    HFSYS_STATE_MAX_VALUE  
};
```

HFSYS_STATE_RUN_THROUGH：透传模式

HFSYS_STATE_RUN_CMD：命令模式

返回值：

HF_SUCCESS:成功，否则失败;

头文件：

hfsys.h

12. 用户 FLASH API

● hfuflash_erase_page

函数原型：

```
int HSF_API hfuflash_erase_page(uint32_t addr, int pages);
```

说明：

擦写用户 flash 的页

参数：

addr: 用户 flash 逻辑地址,不是 flash 物理地址 ;
pages: 要擦除的 flash 页数 ;

返回值：

成功返回 HF_SUCCESS,失败返回 HF_FAIL;

备注：

用户 flash 为物理 flash 的某一块 128KB 的区域,用户只能通过 API 操作这一块区域 ,
API 操作地址为用户 flash 的逻辑地址 , 我们不需要关心它的实际地址。

例子：

参考 example 下的 uflashtest.c

头文件：

hfflash.h

● hfuflash_read

函数原型：

```
int HSF_API hfuflash_read(uint32_t addr, char *data, int len);
```

说明：

从用户文件中读数据

参数：

addr: 用户 flash 的逻辑地址(0- HFUFLASH_SIZE-2) ;
data: 从 flash 的数据的缓存区读取数据 ;
len: 缓存区的大小;

返回值：

小于零失败，否则返回实际从 flash 读到的 Bytes 数;

备注：

无

例子：

参考 example 下的 uflashtest.c

头文件：

hfflash.h

● hfuflash_write

函数原型：

```
int HSF_API hfuflash_write(uint32_t addr, char *data, int len);
```

说明：

向用户文件中写数据

参数：

addr: 用户 flash 的逻辑地址(0- HFUFLASH_SIZE-2) ;

data : 保存要写到 flash 中的数据缓存区 ;

len : 缓存区的大小;

返回值：

如果小于零失败，否则返回实际写入到 flash 的 Bytes 数;

备注：

在对 flash 写之前，如果写的地址已经写入了数据，一定要先进行擦写动作。

data 地址不能是在程序区(ROM) ,只能在 ram 不然调用这个函数会卡死或者程序会返回- HF_E_INVALID,下面代码是不允许的:

错误的写法 1 : " Test" 放在 ROM 区 ;

```
hfuflash_write (Offset,"Test",4);
```

错误的写法 2 : const 修饰的 初始化之后的变量放在程序区(ROM).


```
const uint8_t Data[] = "Test";  
hfuflash_write (Offset,Offset,Data,4);  
正确写法：  
Uint8_t Data[]=" Test" ;  
hfuflash_write (Offset,Offset,Data,4);
```

例子：

参考 example 下的 uflashtest.c

头文件：

hfflash.h

13. 用户文件操作 API

● hffile_userbin_read

函数原型：

```
int HSF_API hffile_userbin_read(uint32_t offset,char *data,int len);
```

说明：

从用户文件中读数据

参数：

offset: 文件偏移量；

data：保存从文件读取到的数据的缓存区；

len：缓存区的大小；

返回值：

如果小于零失败，否则返回实际从文件读到的 Bytes 数；

例子：

无

头文件：

hffile.h

● hffile_userbin_size

函数原型：

```
int HSF_API hffile_userbin_size(void);
```

说明：

从用户文件读 bin 文件的大小。

参数：

无

返回值：

小于零失败，否则返回文件的大小；

备注：

无

例子：

无

头文件：

hffile.h

● hffile_userbin_write

函数原型：

```
int HSF_API hffile_userbin_write(uint32_t offset,char *data,int len);
```

说明：

把数据写入到用户文件

参数：

offset: 文件偏移量；

data：保存要写入到文件数据的缓存区；

len：缓存区的大小；

返回值：

如果小于零失败，否则返回实际写入到文件的 Bytes 数；

备注：

用户配置文件是一个固定大小的文件，文件保存在 flash 中，可以保存用户数据。用户配置文件有备份的功能，用户不需要当心在写的工程中断电，如果写的过程中断电，会自动恢复到写之前的内容。

例子：

无

头文件：

hffile.h

● hffile_userbin_zero

函数原型：

```
int HSF_API hffile_userbin_zero (void);
```

参数：

无

说明：

把整个文件的内容快速清零

返回值：

小于零失败，否则返回文件的大小；

备注：

调用这个函数能够非常快速的把整个文件内容清零；比通过 `hffile_userbin_write` 要快；

例子：

无

头文件：

`hffile.h`

14. 自动升级 API

● hfupdate_complete

函数原型：

```
int hfupdate_complete(HFUPDATE_TYPE_E type,uint32_t file_total_len);
```

说明：

升级完成

参数：

type:升级类型

file_total_len:升级文件的长度

返回值：

成功返回 HF_SUCCESS,否则失败

备注：

当升级文件全下载完成后调用这个函数来执行升级动作。

例子：

参考 example 下的 updatetest.c

头文件：

hfupdate.h

● hfupdate_start

函数原型：

```
int hfupdate_start(HFUPDATE_TYPE_E type);
```

说明：

开始升级.

参数：

type:升级类型

typedef enum HFUPDATE_TYPE

{

HFUPDATE_SW=0,//升级软件

```
HFUPDATE_CONFIG=1, //升级默认配置，暂不支持
HFUPDATE_WIFIFW, //升级 WIFI 固件
HFUPDATE_WEB, //升级 web，暂不支持
}HFUPDATE_TYPE_E;
```

返回值：

成功返回 HF_SUCCESS, 否则失败

备注：

当前只支持 HFUPDATE_SW. 在开始下载升级文件之前先调用这个函数进行初始化。

例子：

参考 example 下的 updatetest.c

头文件：

hfupdate.h

● hfupdate_write_file

函数原型：

```
int hfupdate_write_file(HFUPDATE_TYPE_E type, uint32_t offset, char *data, int len);
```

说明：

把升级文件数据写到升级区.

参数：

type: 升级类型

offset: 升级文件的偏移量

data: 要写入的升级文件数据

len: 升级文件数据的长度

返回值：

大于等于零成功，时间写入的长度，否则失败。

备注：

当前只支持 HFUPDATE_SW.

例子：

参考 example 下的 updatetest.c

头文件：

hfupdate.h

附录 A：硬件定时器

本笔记说明 MC300 硬件 Timer 的使用，MC300 共有 5 个硬件定时器，其中 4 个微秒定时器，1 个毫秒定时器。

Timer 相关头文件为：drv_timer.h

TimerID

5 个定时器 ID 的定义，US_xxx 为微秒定时器，MS_xxx 为毫秒定时器。其中 US_TIMER2 已被系统 Clock 使用，MS_TIMER1 被系统 WatchDog 使用。

```
#define US_TIMER0                (TU0_US_REG_BASE + 0)

#define MS_TIMER0                (TM0_MS_REG_BASE + 0)

#define MS_TIMER1                (TM0_MS_REG_BASE + 0x10)

#define MS_TIMER2                (TM0_MS_REG_BASE + 0x20)

#define MS_TIMER3                (TM0_MS_REG_BASE + 0x30)
```

Timer 启动函数

```
int hwtmr_start(HW_TIMER *tmr, unsigned int count, irq_handler tmr_handle,
               void *m_data, enum hwtmr_op_mode mode);
```

参数说明：

tmr: 为 TimerID，US_xxx 或 MS_xxx

count :为定时时间。最大值为 0xFFFF，所以 US 定时最大一次时间为 65535/1000=65ms。

MS 为 65535/1000=65 s。

tmr_handle : 为中断服务程序, 应放在 RAM 中, 不应有过多的函数调用嵌套, 处理时间应尽量短。

m_data : 为 tmr_handle 的参数

mode : 可取值 HTMR_ONESHOT 为单次定时, HTMR_PERIODIC 循环定时。

测试结果

1. 如果定时时间短, 如 n 百毫秒, 累计误差比较大。如果 60 秒中断一次, 误差很小
2. 中断时间较小时, 如 <100 ms, 会有丢中断的现象。
3. 用软件时钟的 event 定时的不确定性很大, 尽量不要用

例程

// 启动定时器

```
static void test_timer_start(void)
```

```
{
```

```
    u_printf("To init timer...");
```

```
    // MS_TIMER2 定时 60 秒, 循环定时
```

```
    hwtmr_start(MS_TIMER2,60000,ms_timer_callback,NULL, HTMR_PERIODIC);
```

```
    // US_TIMER0 定时 50 ms, 循环定时
```

```
    hwtmr_start(US_TIMER0,50000,us_timer_callback,NULL, HTMR_PERIODIC);
```

```
}
```

// MS_TIMER2 的定时服务程序

```
ATTRIBUTE_SECTION_KEEP_IN_SRAM static void ms_timer_callback( void *arg )
```

```
{
```

```
    u_printf("1");
```

```
}
```

```
// US_TIMER0 的定时服务程序
```

```
ATTRIBUTE_SECTION_KEEP_IN_SRAM static void us_timer_callback( void *arg )
```

```
{
```

```
    static int state=0;
```

```
    // 关中断
```

```
    irq_mask_disable(IRQ_US_TIMER0);
```

```
    if (state==0)
```

```
    {
```

```
        drv_gpio_write(GPIO_18,1);
```

```
        drv_gpio_Output(GPIO_18,0,0);
```

```
        state=1;
```

```
    }
```

```
    else
```

```
    {
```

```
        drv_gpio_write(GPIO_18,0);
```

```
        drv_gpio_Output(GPIO_18,0,0);
```

```
        state=0;
```

```
}  
  
// 开中断  
  
irq_mask_enable(IRQ_US_TIMER0);  
  
}
```

附录 B : GPIO 中断

本笔记说明 MC300 GPIO 中断的使用及 GPIO 按键相应处理。

GPIO 相关头文件为：

drv_gpio.h

gpio_api.h

GPIO 中断初始化函数

```
S32 gpio_irq_enable(GPIO_ID id, GPIO_TRIGGER_MODE mode,  
  
                    void (*callbackfn)handle, void *data);
```

参数说明：

id: 为 GPIOID , 在 drv_gpio 中定义

```
typedef enum t_GPIO_ID  
{  
  
    GPIO_1      = 0,  
  
    GPIO_2,  
  
    GPIO_3,  
  
    GPIO_5,  
  
    GPIO_6,  
  
    GPIO_8,  
  
    GPIO_15,  
  
    GPIO_18,
```

```

        GPIO_19,

        GPIO_20,

        GPIO_MAX

    } GPIO_ID;

```

mode：为中断模式，只有 RISING_EDGE 有效，即上升沿触发。

handle：为中断服务程序，**应放在 RAM 中，不应有过多的函数调用嵌套，处理时间应尽量短。**

data：为 handle 的参数

注意 GPIO 中断处理函数只能有一个，如果要多个 GPIO 做中断，只能用同一个 handle，可以通过参数 data 识别具体是哪个 GPIO 产生的中断。

按键点 LED 例程

// 初始化 GPIO 中断

```
void gpio_interrupt_init()
```

```
{
```

```
    u_printf("To init gpio interrupt...\n");
```

```
    gpio_irq_enable(GPIO_2,RISING_EDGE,gpio_interrupt_cb,(void*)GPIO_2);
```

```
}
```

// GPIO 中断服务程序

```
ATTRIBUTE_SECTION_KEEP_IN_SRAM static void gpio_interrupt_cb(void *data)
```

```
{
```

```
    static int state=0;
```

```

    irq_mask_disable(IRQ_GPIO);

    u_printf("1");

    if (state==0)

    {

        drv_gpio_write(GPIO_18,1);

        drv_gpio_Output(GPIO_18,0,0);

        state=1;

    }

    else

    {

        drv_gpio_write(GPIO_18,0);

        drv_gpio_Output(GPIO_18,0,0);

        state=0;

    }

    irq_mask_enable(IRQ_GPIO);

}

```

例程去抖动处理

通常物理按键在动作过程中，上面代码没做去抖动处理，可能出现：一次按键后连续进两次中断，这样灯的状态就不对了。修改代码如下：

```

ATTRIBUTE_SECTION_KEEP_IN_SRAM static void gpio_interrupt_cb(void *data)

{

```

```

    irq_mask_disable(IRQ_GPIO);

    // 启动 50 ms 单次定时

    hwtmr_stop(US_TIMER0);

    hwtmr_start(US_TIMER0,50000,us_timer_callback,NULL,HTMR_ONESHOT);

    irq_mask_enable(IRQ_GPIO);
}

// US_TIMER0 的定时服务程序

ATTRIBUTE_SECTION_KEEP_IN_SRAM static void us_timer_callback( void *arg )
{
    static int state=0;

    // 关中断

    irq_mask_disable(IRQ_US_TIMER0);

    if (state==0)
    {
        drv_gpio_write(GPIO_18,1);

        drv_gpio_Output(GPIO_18,0,0);

        state=1;
    }

    else

    {

        drv_gpio_write(GPIO_18,0);
    }
}

```

```

        drv_gpio_Output(GPIO_18,0,0);

        state=0;

    }

    // 开中断

    irq_mask_enable(IRQ_US_TIMER0);

}

```

长按判断例程

// 定义 1 个全局变量

```
static int long_press=0;           // 是否处于长按状态，默认为 0
```

```

ATTRIBUTE_SECTION_KEEP_IN_SRAM static void ms_timer_callback( void *arg )
{

    static int count=0;

    irq_mask_disable(IRQ_MS_TIMER2);

    if (drv_gpio_read(GPIO_2)==0)

    {

        if (long_press>0)

        {

            if (count==0)

            {

                u_printf("lp:%d s\n", long_press);

```



```

    }

    count++;

    if (count>10)        // 1s
    {
        count=0;

        long_press++;
    }
}

else

{

    count++;

    if (count>=10)      // 1s
    {
        long_press=1;

        count=0;
    }
}

}

else

{

    count=0;

}

```

```

    irq_mask_enable(IRQ_MS_TIMER2);

}

static void test_timer_start(void)

{

    // MS_TIMER2 定时 100ms , 循环定时

    hwtmr_start(MS_TIMER2,100,ms_timer_callback,NULL, HTMR_PERIODIC);

}

ATTRIBUTE_SECTION_KEEP_IN_SRAM static void gpio_interrupt_cb(void *data)

{

    irq_mask_disable(IRQ_GPIO);

    // 启动 50 ms 单次定时

    hwtmr_stop(US_TIMER0);

    hwtmr_start(US_TIMER0,50000,us_timer_callback,NULL,HTMR_ONESHOT);

    irq_mask_enable(IRQ_GPIO);

}

// US_TIMER0 的定时服务程序

ATTRIBUTE_SECTION_KEEP_IN_SRAM static void us_timer_callback( void *arg )

{

    static int state=0;

```

```

    irq_mask_disable(IRQ_US_TIMER0);

    // 按键放开，判断是是否长按，中断服务程序串口打印尽量短

    if (long_press==0)

        u_printf("sp\n");

    else

        u_printf("lp rls\n");

    irq_mask_enable(IRQ_US_TIMER0);
}

void gpio_interrupt_init()

{

    gpio_irq_enable(GPIO_2,RISING_EDGE,gpio_interrupt_cb,(void*)GPIO_2);

}

```