OneMO | 中国移动 China Mobile

ML307A
外围接口开发指导手册

版本：V1.0.0
发布日期：2022/12/8

China Mobile IoT Co., Ltd.
iot.10086.cn

服务与支持

如果您有任何关于模组产品及产品手册的评论、疑问、想法，或者任何无法从本手册中找到答案的疑问，请通过以下方式联系我们。

OneMO官网： onemo10086.com

邮箱： SmartModule@cmiot.chinamobile.com

客户服务热线：400-110-0866

文档声明

注意

本手册描述的产品及其附件特性和功能，取决于当地网络设计或网络性能，同时也取决于用户预先安装的各种软件。由于当地网络运营商、ISP，或当地网络设置等原因，可能也会造成本手册中描述的全部或部分产品及其附件特性和功能未包含在您的购买或使用范围之内。

责任限制

除非合同另有约定，中移物联网有限公司对本文档内容不做任何明示或暗示的声明或保证，并且不对特定目的适销性及适用性或者任何间接的、特殊的或连带的损失承担任何责任。

在适用法律允许的范围内，在任何情况下，中移物联网有限公司均不对用户因使用本手册内容和本手册中描述的产品而引起的任何特殊的、间接的、附带的或后果性的损坏、利润损失、数据丢失、声誉和预期的节省而负责。

因使用本手册中所述的产品而引起的中移物联网有限公司对用户的最大赔偿（除在涉及#身伤害的情况中根据适用法律规定的损害赔偿外），不应超过用户为购买此产品而支付的金额。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。公司保留随时修改本手册中任何信息的权利，无需进行提前通知且不承担任何责任。

商标声明

中国移动 China Mobile 为中国移动注册商标。

本手册和本手册描述的产品中出现的其他商标、产品名称、服务名称和公司名称，均为其各自所有者的财产。

进出口法规

出口、转口或进口本手册中描述的产品（包括但不限于产品软件和技术数据），用户应遵守相关进出口法律和法规。

隐私保护

关于我们如何保护用户的个人信息等隐私情况，请查看相关隐私政策。

操作系统更新声明

操作系统仅支持官方升级；如用户自己刷非官方系统，导致安全风险和损失由用户负责。

固件包完整性风险声明

固件仅支持官方升级；如用户自己刷非官方固件，导致安全风险和损失由用户负责。

版权所有©中移物联网有限公司。保留一切权利。

关于文档

修订记录

版本	描述
V1.0.0	初版

目录

1. 概述

本文档介绍了OpenCPU SDK外围接口资源的硬件说明、API说明、使用示例以及注意事项，外围接口包括GPIO、I2C、UART、PWM等。

1.1. 适用范围

Table 1. 适用模组

模组系列	模组子型号
ML307A	ML307A-DCLN/ML307A-DSLN

1.2. 写作目的

本文档介绍了模组OpenCPU的外围接口函数的使用示例和使用注意事项，帮助软件开发者更好理解和使用SDK的接口函数，实现外设相关功能开发。

2. 外设接口总述

本文档主要介绍了外设软件接口，通过这些接口可以实现外设的连接，但是在使用这些函数接口之前，需要阅读相关硬件手册实现硬件的正确连接，在使用时需要仔细阅读外设的注意事项，以便实现外设的正确连接和操作。

3. GPIO

本章主要介绍OpenCPU SDK中外设GPIO的相关使用情况。

3.1. 硬件说明

OpenCPU GPIO引脚映射详见对应版本的资源综述和硬件设计手册。

3.2. API说明

SDK中提供一套完整的GPIO用户编程接口，具体接口定义和说明请查阅cm_gpio.h头文件。

GPIO参数结构体

```
/** 引脚ID */
typedef enum{
 CM_GPIO_NUM_0,
 CM_GPIO_NUM_1,
 CM_GPIO_NUM_2,
 CM_GPIO_NUM_3,
 CM_GPIO_NUM_4,
 CM_GPIO_NUM_5,
 CM_GPIO_NUM_6,
 CM_GPIO_NUM_7,
 CM_GPIO_NUM_8,
 CM_GPIO_NUM_9,
 CM_GPIO_NUM_10,
 CM_GPIO_NUM_11,
 CM_GPIO_NUM_12,
 CM_GPIO_NUM_13,
 CM_GPIO_NUM_14,
 CM_GPIO_NUM_15,
 CM_GPIO_NUM_16,
 CM_GPIO_NUM_17,
 CM_GPIO_NUM_18,
 CM_GPIO_NUM_19,
 CM_GPIO_NUM_20,
 CM_GPIO_NUM_21,
 CM_GPIO_NUM_22,
 CM_GPIO_NUM_23,
 CM_GPIO_NUM_MAX
}cm_gpio_num_e;

/* *GPIO工作模式 */
typedef enum
{
 CM_GPIO_MODE_NUM/*!<不可用 工作模式数量*/
} cm_gpio_mode_e;


/** I/O方向 */
typedef enum{
 CM_GPIO_DIRECTION_INPUT = 0,
 CM_GPIO_DIRECTION_OUTPUT,
}cm_gpio_direction_e;


/** 上/下拉 */
typedef enum{
 CM_GPIO_PULL_NONE,
 CM_GPIO_PULL_DOWN,
 CM_GPIO_PULL_UP,
}cm_gpio_pull_e;
```

```
/** 边沿+电平触发 */
typedef enum{
 CM_GPIO_IT_EDGE_RISING,
 CM_GPIO_IT_EDGE_FALLING,
 CM_GPIO_IT_EDGE_BOTH,
 CM_GPIO_IT_LEVEL_HIGH,/*!<不支持*/
 CM_GPIO_IT_LEVEL_LOW,/*!<不支持*/
}cm_gpio_interrupt_e;


/** 高低电平 */
typedef enum{
 CM_GPIO_LEVEL_LOW,
 CM_GPIO_LEVEL_HIGH,
}cm_gpio_level_e;


/** 命令码 */
typedef enum{
 CM_GPIO_CMD_SET_PULL, /*!< 上下拉设置命令码*/
 CM_GPIO_CMD_GET_PULL, /*!< 上下拉获取命令码*/
 CM_GPIO_CMD_SET_LEVEL, /*!< 驱动能力设置命令码*/
 CM_GPIO_CMD_GET_LEVEL, /*!< 驱动能力获取命令码*/
 CM_GPIO_CMD_SET_DIRECTION, /*!< 输入输出设置命令码*/
 CM_GPIO_CMD_GET_DIRECTION, /*!< 输入输出获取命令码*/
} cm_gpio_cmd_e;


/** 配置 */
typedef struct{
 cm_gpio_mode_e mode;/*!< 不支持*/
 cm_gpio_direction_e direction;
 cm_gpio_pull_e pull;
} cm_gpio_cfg_t;
```

GPIO接口定义

```
/**
 * @brief 初始化GPIO
 *
 * @param [in] pin GPIO引脚ID
 * @param [in] cfg 配置
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details *********初始化之前一定要先设置引脚复用*********
 */
int32_t cm_gpio_init(cm_gpio_num_e gpio_num, cm_gpio_cfg_t *cfg);


/**
 * @brief 去初始化
```

```
 *
 * @param [in] pin GPIO引脚ID
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_deinit(cm_gpio_num_e gpio_num);


/**
 * @brief 设置输出电平
 *
 * @param [in] pin GPIO引脚ID
 * @param [in] level 输出电平
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_set_level(cm_gpio_num_e gpio_num, cm_gpio_level_e level);


/**
 * @brief 读取输入电平
 *
 * @param [in] pin GPIO引脚ID
 * @param [out] level 输入电平
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_get_level(cm_gpio_num_e gpio_num, cm_gpio_level_e *level);


/**
 * @brief 设置上/下拉
 *
 * @param [in] pin GPIO引脚ID
 * @param [in] type 上下拉配置
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_set_pull(cm_gpio_num_e gpio_num, cm_gpio_pull_e type);
```

```
/**
* @brief 读取上/下拉
*
* @param [in] pin GPIO引脚ID
* @param [out] type 上下拉配置
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details More details
*/
int32_t cm_gpio_get_pull(cm_gpio_num_e gpio_num, cm_gpio_pull_e *type);


/**
* @brief I/O方向模式配置
*
* @param [in] pin GPIO引脚ID
* @param [in] dir 输入/输出/高阻态
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details More details
*/
int32_t cm_gpio_set_direction(cm_gpio_num_e gpio_num, cm_gpio_direction_e dir);


/**
* @brief 获取I/O方向模式配置
*
* @param [in] pin GPIO引脚ID
* @param [out] dir 输入/输出/高阻态
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details More details
*/
int32_t cm_gpio_get_direction(cm_gpio_num_e gpio_num, cm_gpio_direction_e *dir);


/**
* @brief 注册中断回调函数
*
* @param [in] pin GPIO引脚ID
* @param [in] interrupt_cb 中断回调函数
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
```

```
 * @details More details
 */
int32_t cm_gpio_interrupt_register(cm_gpio_num_e gpio_num, void *interrupt_cb);


/**
 * @brief 使能中断
 *
 * @param [in] pin GPIO引脚ID
 * @param [in] intr_mode 中断触发方式
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_interrupt_enable(cm_gpio_num_e gpio_num, cm_gpio_interrupt_e intr_mode);


/**
 * @brief 失能中断
 *
 * @param [in] pin GPIO引脚ID
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_gpio_interrupt_disable(cm_gpio_num_e gpio_num);


/**
 * @brief GPIO控制
 *
 * @param [in] pin GPIO引脚ID
 * @param [in] cmd 命令码
 * @param [in] arg 命令
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details 根据不同的cmd值（命令码），arg参数类型须对应配置。\n
 * cmd = \ref GPIO_CMD_SET_PULL, arg传入\ref cm_gpio_pull_e * \n
 * cmd = \ref GPIO_CMD_GET_PULL, arg传入\ref cm_gpio_pull_e * \n
 * cmd = \ref GPIO_CMD_SET_LEVEL, arg传入\ref cm_gpio_level_e * \n
 * cmd = \ref GPIO_CMD_GET_LEVEL, arg传入\ref cm_gpio_level_e * \n
 * cmd = \ref GPIO_CMD_SET_DIRECTION, arg传入\ref cm_gpio_direction_e * \n
 * cmd = \ref GPIO_CMD_GET_DIRECTION, arg传入\ref cm_gpio_direction_e * \n
 */
int32_t cm_gpio_ioctl(cm_gpio_num_e gpio_num, int32_t cmd, void *arg);
```

3.3. 使用示例

GPIO DEMO提供了GPIO参数设置、输入输出设置、电平设置和中断设置等功能示例程序，详细信息参考SDK中的cm_demo_gpio.c文件。

电平输入

```
void cm_test_gpio_read(unsigned char **cmd, int len)
{
 cm_gpio_cfg_t cfg = {0};
 cm_gpio_level_e level;
 cm_gpio_num_e test_gpio = atoi((char *)cmd[2]);
 cm_iomux_pin_e pin;
 cm_iomux_func_e fun;
 cm_iomux_func_e fun1;
 cm_gpio_direction_e dir;
 char *stopstr;


 pin = strtol((char *)cmd[3], &stopstr, 16);
 fun = strtol((char *)cmd[4], &stopstr, 16);

 cfg.direction = CM_GPIO_DIRECTION_INPUT;
 cfg.pull = CM_GPIO_PULL_DOWN;

 cm_iomux_set_pin_func(pin, fun);//初始化之前一定要先设置引脚复用
 cm_iomux_get_pin_func(pin, &fun1);
 if(fun1 !=fun)
 {
 cm_demo_printf("read gpio_%d set iomux failed\n", test_gpio);
 return;
 }

 cm_gpio_init(test_gpio, &cfg);
 cm_iomux_get_pin_func(pin, &fun1);
 cm_gpio_get_level(test_gpio, &level);
 cm_demo_printf("read gpio_%d level = %d\n", test_gpio, level);
 cm_gpio_get_direction(test_gpio, &dir);
 cm_demo_printf("read gpio_%d dir = %d\n", test_gpio, dir);
 cm_gpio_ioctl(test_gpio, CM_GPIO_CMD_GET_DIRECTION, &dir);
 cm_demo_printf("read gpio_%d dir = %d\n", test_gpio, dir);
 cm_gpio_ioctl(test_gpio, CM_GPIO_CMD_GET_LEVEL, &level);
 cm_demo_printf("read gpio_%d level = %d\n", test_gpio, level);
 cm_gpio_deinit(test_gpio);
}
```

电平输出

```
void cm_test_gpio_write(unsigned char **cmd, int len)
{
 cm_gpio_cfg_t cfg = {0};
 cm_gpio_level_e level;
 cm_gpio_num_e test_gpio;
```

```
cm_iomux_pin_e pin;
cm_iomux_func_e fun;
char *stopstr;

test_gpio = atoi((char *)cmd[2]);
pin = strtol((char *)cmd[3], &stopstr, 16);
fun = strtol((char *)cmd[4], &stopstr, 16);
level = atoi((char *)cmd[5]);

if((level == CM_GPIO_LEVEL_HIGH) ||(level == CM_GPIO_LEVEL_LOW))
{
cfg.direction = CM_GPIO_DIRECTION_OUTPUT;
cfg.pull = CM_GPIO_PULL_UP;

cm_iomux_set_pin_func(pin, fun);//初始化之前一定要先设置引脚复用

cm_gpio_init(test_gpio, &cfg);
cm_gpio_set_level(test_gpio, level);
cm_demo_printf("set gpio_%d level = %d\n", test_gpio, level);
}
else
{
cm_demo_printf("set gpio_%d level = %d,please set 0 or 1\n", test_gpio, level);
}
}
```

中断

```
static void cm_test_gpio_irq_callback(void)
{
 cm_demo_printf("GPIO INTERRUPT!!!!!!!!");
}


void cm_test_gpio_irq(unsigned char **cmd, int len)
{
 cm_gpio_cfg_t cfg = {0};
 cm_gpio_num_e test_gpio = 0;
 cm_iomux_pin_e pin;
 cm_iomux_func_e fun;
 char *stopstr;

 test_gpio = atoi((char *)cmd[2]);
 pin = strtol((char *)cmd[3], &stopstr, 16);
 fun = strtol((char *)cmd[4], &stopstr, 16);

 cfg.direction = CM_GPIO_DIRECTION_INPUT;
 cfg.pull = CM_GPIO_PULL_DOWN;

 cm_iomux_set_pin_func(pin, fun);//初始化之前一定要先设置引脚复用

 cm_gpio_init(test_gpio, &cfg);
 cm_gpio_interrupt_register(test_gpio, cm_test_gpio_irq_callback);
 cm_gpio_interrupt_enable(test_gpio, CM_GPIO_IT_EDGE_BOTH);
}
```

3.4. 注意事项

使用GPIO应注意以下事项：

- 大多数 引脚都是多路复用的。这些引脚也称为多功能引脚，因为它们中的每一个都分配了多个功能。每个引脚都有一个专用的多功能引脚寄存器，用于控制其功能和行为。从分配给多功能引脚的几个功能中，选择出一个。其他的引脚是专用引脚，仅分配给一个功能。因此，使用前设置引脚复用情况，且同时仅有一个引脚复用为该功能，配置复用功能之前，确保每个引脚的功能都不一样。

4. ADC

本章主要介绍OpenCPU SDK中外设ADC的相关使用情况。

4.1. 硬件说明

OpenCPU ADC引脚映射详见对应版本的资源综述和硬件设计手册。

4.2. API说明

SDK中提供一套完整的ADC用户编程接口，具体接口定义和说明请查阅cm_adc.h头文件。

ADC参数结构体

```
/** ADC设备ID */
typedef enum
{
 CM_ADC_0 = 1, /**< ADC0 */
 CM_ADC_NUM, /**< ADC设备个数 */
} cm_adc_dev_e;
```

ADC接口定义

```
/**
* @brief 读取ADC电压
*
* @param [in] dev ADC设备ID
* @param [out] voltage 电压值, 单位: mv
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details 通过ADC设备读取测量电压值
*/
int32_t cm_adc_read(cm_adc_dev_e dev, int32_t *voltage);
```

4.3. 使用示例

ADC DEMO提供了ADC通道电压值读取功能示例程序，详细信息请参考SDK中的cm_demo_adc.c文件。

ADC通道电压读取

```
int cm_test_adc(unsigned char **cmd, int len)
{
 int32_t voltage;
 int32_t ret;

 CM_DEMO_ADC_LOG("adc test start!!\n");
 //测试ADC0
 ret = cm_adc_read(CM_ADC_0,&voltage);
 if(ret != RET_SUCCESS)
 {
  CM_DEMO_ADC_LOG("adcCM_ADC_0 read err,ret=0x%08x\n", ret);
  return -1;
 }
 CM_DEMO_ADC_LOG("adc CM_ADC_0 read:%ld(mv)!!\n",voltage);
 CM_DEMO_ADC_LOG("adc test end!!\n");

 return 0;
}
```

4.4. 注意事项

本节介绍ADC的相关注意事项。

Note: ADC获取电压量程为在0~1.2V。

5. UART

本章主要介绍OpenCPU SDK中外设UART的相关使用情况。

5.1. 硬件说明

OpenCPU UART引脚映射详见对应版本的资源综述和硬件设计手册。

5.2. API说明

SDK中提供一套完整的UART用户编程接口，具体接口定义和说明请查阅cm_uart.h头文件。

UART参数结构体

```
/** 设备ID */
typedef enum{
 CM_UART_DEV_0, /*!< 设备0*/
 CM_UART_DEV_1, /*!< 设备1*/
 CM_UART_DEV_NUM
} cm_uart_dev_e;


/** 数据位 */
typedef enum{
 CM_UART_BYTE_SIZE_8 = 8,
} cm_uart_byte_size_e;


/** 奇偶校验 */
typedef enum{
 CM_UART_PARITY_NONE,
 CM_UART_PARITY_ODD,
 CM_UART_PARITY_EVEN,
 CM_UART_PARITY_MARK, /*!< 不支持*/
 CM_UART_PARITY_SPACE /*!< 不支持*/
} cm_uart_parity_e;


/** 停止位 */
typedef enum{
 CM_UART_STOP_BIT_ONE,
 CM_UART_STOP_BIT_ONE_HALF,
 CM_UART_STOP_BIT_TWO
} cm_uart_stop_bit_e;


/** 流控制 */
typedef enum{
 CM_UART_FLOW_CTRL_NONE,
 CM_UART_FLOW_CTRL_HW, /*!< 当前不支持流控*/
} cm_uart_flow_ctrl_e;


/** 结果码 */
typedef enum{
 CM_UART_RET_OK = 0,
 CM_UART_RET_INVALID_PARAM = -1,
} cm_uart_ret_e;


/** 配置 */
typedef struct{
 cm_uart_byte_size_e byte_size; /*!< 数据位，枚举*/
```

```
 cm_uart_parity_e parity; /*!< 校验位，枚举*/
 cm_uart_stop_bit_e stop_bit; /*!< 停止位，枚举*/
 cm_uart_flow_ctrl_e flow_ctrl; /*!< 流控制，枚举*/
 int baudrate; /*!< 波特率，枚举*/
} cm_uart_cfg_t;


/** 事件类型 */
typedef enum
{
 CM_UART_EVENT_TYPE_RX_ARRIVED = (1 << 0), /*!< 接收到新的数据*/
 CM_UART_EVENT_TYPE_RX_OVERFLOW = (1 << 1), /*!< 接收FIFO缓存溢出*/
 CM_UART_EVENT_TYPE_TX_COMPLETE = (1 << 2) /*!< 不支持发送完成事件*/
}cm_uart_event_type_e;


/** 事件 */
typedef struct{
 uint32_t event_type; /*!< 事件类型，数据可读等*/
 void *event_param; /*!< 事件参数*/
 void *event_entry; /*!< 事件执行入口*/
} cm_uart_event_t;
```

API接口声明

```
/**
* @brief 打开串口
*
* @param [in] dev 串口设备ID
* @param [in] cfg 串口配置
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details open之前必须先对引脚复用功能进行设置
*/
int32_t cm_uart_open(cm_uart_dev_e dev, cm_uart_cfg_t *cfg);


/**
* @brief 注册串口事件
*
* @param [in] dev 串口设备ID
* @param [in] event 串口事件
*
* @return
* = 0 - 成功 \n
* < 0 - 失败, 返回值为错误码
*
* @details 事件包括串口数据可读/溢出等。需在open之前注册。
*     回调函数中不可输出LOG、串口打印、执行复杂任务或消耗过多资源。
*/
int32_t cm_uart_register_event(cm_uart_dev_e dev, void *event);
```

```
/**
 * @brief 关闭串口
 *
 * @param [in] dev 串口设备ID
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details
 */
int32_t cm_uart_close(cm_uart_dev_e dev);


/**
 * @brief 串口写数据
 *
 * @param [in] dev 串口设备ID
 * @param [in] data 待写入数据
 * @param [in] len 长度
 * @param [in] timeout 超时时间(ms)(无效参数)
 *
 * @return
 *  = 实际写入长度 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details
 */
int32_t cm_uart_write(cm_uart_dev_e dev, const void *data, int32_t len, int32_t timeout);


/**
 * @brief 串口读数据
 *
 * @param [in] dev 串口设备ID
 * @param [out] data 待读数据
 * @param [in] len 长度
 * @param [in] timeout 超时时间(ms)(无效参数)
 *
 * @return
 *  = 实际读出长度 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details
 */
int32_t cm_uart_read(cm_uart_dev_e dev, void* data, int32_t len, int32_t timeout);
```

5.3. 使用示例

UART DEMO提供了UART使用的功能示例程序，详细信息请参考SDK中的cm_demo_uart.c文件。

UART示例

```
/******************************************************************************
 * Pre-processor Definitions
 *****************************************************************************/
#define UART_BUF_LEN 1024

/******************************************************************************
 * Private Types
 *****************************************************************************/
static cm_uart_event_type_e event_type = 0;
static int rx_rev_len = 0;
static char rx_rev_data[UART_BUF_LEN] = {0};
static osThreadId_t OC_Uart_TaskHandle; //串口数据接收、解析任务Handle
static void* g_uart_sem = NULL;
extern osEventFlagsId_t cmd_task_flag;
cm_uart_cmd_recv_t gstUartCmdRecv = {0}; //串口命令缓冲区
/******************************************************************************
 * Private Function Prototypes
 *****************************************************************************/
void cm_demo_printf (char *str, ...);
static void cm_uart_recv_task(void *param);
static int __cm_cmd_engine(char * prefix,char * uart_buf,int* uart_buf_len,char **cmd_buf,int *cmd_len);

/******************************************************************************
 * Private Data
 *****************************************************************************/


/******************************************************************************
 * Private Functions
 *****************************************************************************/
static int __cm_cmd_engine(char * prefix,char * uart_buf,int* uart_buf_len,char **cmd_buf,int *cmd_len)
{
 char *p1,*p2,*p3,*temp1,*temp2;
 uart_buf[*uart_buf_len] = 0;//结尾
 p1 = strstr(uart_buf,prefix);
 if(p1 == 0)
 {
  *uart_buf_len = 0;
  return -1;
 }
 p2 = strstr(p1,"\r\n");
 if(p2 == 0)
 {
  *uart_buf_len = 0;
  return -1;
 }
 *cmd_len = 0;
 p3 = strchr(p1,':');
 if(p3 == 0)
```

```
{
 *uart_buf_len = 0;
 return -1;
}
cmd_buf[(*cmd_len)++] = cm_malloc(p3-p1+1+1);
memset(cmd_buf[*cmd_len-1],0,p3-p1+1);
if(cmd_buf[*cmd_len -1] == 0)
{
 *uart_buf_len = 0;
 return -1;
}
memcpy(cmd_buf[*cmd_len-1],p1,(int)(p3-p1));
temp1= p3;
while(1)
{
 if(*cmd_len >= 20)
 {
  *uart_buf_len = 0;
  for(int j = 0;j< *cmd_len;j++)
  {
   cm_free(cmd_buf[j]);
  }
  return -1;
 }
 temp2 = strchr(temp1+1,':');
 if(temp2 == 0)
  break;
 if((temp2-temp1) < 1)
 {
  for(int j = 0;j< *cmd_len;j++)
  {
   cm_free(cmd_buf[j]);
  }
  *uart_buf_len = 0;
  return -1;
 }
 cmd_buf[(*cmd_len)++] = cm_malloc(temp2-temp1+1);
 if(cmd_buf[*cmd_len -1] == 0)
 {
  for(int j = 0;j< *cmd_len;j++)
  {
   cm_free(cmd_buf[j]);
  }
  *uart_buf_len = 0;
  return -1;
 }
 memset(cmd_buf[*cmd_len-1],0,temp2-temp1);
 memcpy(cmd_buf[*cmd_len-1],temp1+1,temp2-temp1-1);
 temp1 = temp2;
}
if((p2-temp1-1) < 1)
{
 for(int j = 0;j< *cmd_len;j++)
 {
  cm_free(cmd_buf[j]);
 }
```

```
 *uart_buf_len = 0;
 return -1;
}

cmd_buf[(*cmd_len)++] = cm_malloc(p2-temp1+1);
memset(cmd_buf[*cmd_len-1],0,p2-temp1);
memcpy(cmd_buf[*cmd_len-1],temp1+1,p2-1-temp1);
*uart_buf_len = 0;
return 0;
}

/* 回调函数中不可输出LOG、串口打印、执行复杂任务或消耗过多资源 */
static void cm_serial_uart_callback(void *param, uint32_t type)
{

 if (CM_UART_EVENT_TYPE_RX_ARRIVED & type)
 {
  osSemaphoreRelease(g_uart_sem);
 }

 if(CM_UART_EVENT_TYPE_RX_OVERFLOW & type)
 {

 }
}


static void cm_uart_recv_task(void *param)
{
 int temp_len = 0;
 cm_uart_cmd_recv_t* pstUartCmdRecv = &gstUartCmdRecv;
 while (1)
 {
  if (g_uart_sem != NULL)
  {
   osSemaphoreAcquire(g_uart_sem, osWaitForever);//阻塞
  }
  if (rx_rev_len < UART_BUF_LEN)
  {
   temp_len = cm_uart_read(OPENCPU_MAIN_URAT, (void*)&rx_rev_data[rx_rev_len], UART_BUF_LEN - rx_rev_len, 1000);
   rx_rev_len += temp_len;
  }
  if (g_uart_sem != NULL && (strstr(rx_rev_data,"\r\n")))
  {

   //处理收到数据事件
   if(gstUartCmdRecv.cmd_execute==0)
   {
    gstUartCmdRecv.cmd_execute ++;
    if (rx_rev_len != 0)
    {
     if(pstUartCmdRecv->cmd_execute != 0)
     {
      //OC指令参数的格式是否正确
      char *trai_colon_pz = NULL;
```

```
    trai_colon_pz = strrchr((const char * )rx_rev_data, ':');

        if((trai_colon_pz != NULL)&&((*(trai_colon_pz+1) == '\0') || (*(trai_colon_pz+1) == '\r') ||(*(trai_colon_pz+1) == '\n')))
    {
     cm_demo_printf("formate error\n");
     memset((void*)rx_rev_data, 0, sizeof(rx_rev_data));
     rx_rev_len = 0;
     pstUartCmdRecv->cmd_execute = 0;
     continue;
    }

    //解析命令，如果未检测到正确的命令格式，则丢弃本次数据
    if(__cm_cmd_engine("CM",rx_rev_data,&rx_rev_len,(char **)(pstUartCmdRecv->buf),&pstUartCmdRecv->len) == -1)
    {
     cm_demo_printf("CM CMD error\n");
     memset((void*)rx_rev_data, 0, sizeof(rx_rev_data));
     rx_rev_len = 0;
     pstUartCmdRecv->cmd_execute = 0;
    }
    else
    {
     //清空数据缓冲，并提示主任务开始执行命令
     memset((void*)rx_rev_data, 0, sizeof(rx_rev_data));
     rx_rev_len = 0;
     osEventFlagsSet(cmd_task_flag, 0x00000001U);
    }
   }
  }
 }
 else
 {
 memset((void*)rx_rev_data, 0, sizeof(rx_rev_data));
 rx_rev_len = 0;
 cm_demo_printf("Uart busy\n");
 }
}
}

/*****************************************************************************
* Public Functions
*****************************************************************************/
void cm_demo_printf (char *str, ...)
{
 static char s[600]; //This needs to be large enough to store the string TODO Change magic number
 va_list args;
 int len;

 if ((str == NULL) || (strlen(str) == 0))
 {
  return;
 }

 va_start (args, str);
 len = vsnprintf ((char*)s, 600, str, args);
 va_end (args);
```

```
 cm_uart_write(OPENCPU_MAIN_URAT, s, len, 1000);
}

void cm_demo_uart(void)
{
 int32_t ret = -1;
 cm_uart_cfg_t config =
 {
  CM_UART_BYTE_SIZE_8,
  CM_UART_PARITY_NONE,
  CM_UART_STOP_BIT_ONE,
  CM_UART_FLOW_CTRL_NONE,
  CM_UART_BAUDRATE_9600
 };
 cm_uart_event_t event =
 {
  CM_UART_EVENT_TYPE_RX_ARRIVED|CM_UART_EVENT_TYPE_RX_OVERFLOW,
  "uart0",
  cm_serial_uart_callback
 };

 cm_log_printf(0, "uart NUM = %d demo start... ...", OPENCPU_MAIN_URAT);

 cm_iomux_set_pin_func(17, CM_IOMUX_FUNC_FUNCTION1);
 cm_iomux_set_pin_func(18, CM_IOMUX_FUNC_FUNCTION1);

 ret = cm_uart_register_event(OPENCPU_MAIN_URAT, &event);

 if(ret != RET_SUCCESS)
 {
  cm_log_printf(0, "uart register event err,ret=%d\n", ret);
  return;
 }
 ret = cm_uart_open(OPENCPU_MAIN_URAT, &config);

 if(ret != RET_SUCCESS)
 {
  cm_log_printf(0, "uart init err,ret=%d\n", ret);
  return;
 }

 cm_log_printf(0, "cm_uart_register_event start... ...\n");

 osThreadAttr_t uart_task_attr = {0};
 uart_task_attr.name = "uart_task";
 uart_task_attr.stack_size = 2048;
 uart_task_attr.priority= UART_TASK_PRIORITY;
 gstUartCmdRecv.cmd_execute = 0;
 if(g_uart_sem == NULL)
  g_uart_sem = osSemaphoreNew(1, 0, NULL);

 OC_Uart_TaskHandle = osThreadNew(cm_uart_recv_task,0,&uart_task_attr);
}

void cm_test_uart_close(char **cmd, int len)
{
```

```
cm_uart_dev_e dev = atoi(cmd[2]);

if(0 == cm_uart_close(dev))
{
 cm_demo_printf("uart%d close is ok\n", dev);
}
else
{
 cm_demo_printf("uart%d close is error\n", dev);
}
}
```

5.4. 注意事项

使用UART应注意以下事项：

- 无流控功能；
- 回调函数中不可输出LOG、串口打印、执行复杂任务或消耗过多资源，建议以信号量或消息队列形式控制其他线程执行任务。

6. SPI

本章主要介绍OpenCPU SDK中外设SPI的相关使用情况。

6.1. 硬件说明

OpenCPU SPI引脚映射详见对应版本的资源综述和硬件设计手册。

6.2. API说明

SDK中提供一套完整的SPI用户编程接口，具体接口定义和说明请查阅cm_spi.h头文件。

SPI参数结构体

```
/** 设备ID */
typedef enum
{
 CM_SPI_DEV_0, /*!< 设备1*/
 CM_SPI_DEV_1, /*!< 设备2*/
 CM_SPI_DEV_NUM
}cm_spi_dev_e;

/** SPI主从模式 */
typedef enum
{
 CM_SPI_MODE_SLAVE, /*!< SPI外设从机模式，暂不支持 */
 CM_SPI_MODE_MASTER/*!< SPI外设主机模式*/
} cm_spi_mode_e;

/** 模式 */
typedef enum
{
 CM_SPI_WORK_MODE_0, /*!< CPOL = 0,CPHA = 0*/
 CM_SPI_WORK_MODE_1, /*!< CPOL = 0,CPHA = 1*/
 CM_SPI_WORK_MODE_2, /*!< CPOL = 1,CPHA = 0*/
 CM_SPI_WORK_MODE_3 /*!< CPOL = 1,CPHA = 1*/
} cm_spi_work_mode_e;

/** SPI传输数据宽度 */
typedef enum
{
 CM_SPI_DATA_WIDTH_8BIT,/*!< SPI外设传输数据宽度8位 */
 CM_SPI_DATA_WIDTH_16BIT,/*!< SPI外设传输数据宽度16位 */
 CM_SPI_DATA_WIDTH_32BIT /*!< SPI外设传输数据宽度32位 */
} cm_spi_data_width_e;

/** SPI的NSS管理模式 */
typedef enum
{
 CM_SPI_NSS_SOFT, /*!< SPI外设软件控制NSS，暂不支持*/
 CM_SPI_NSS_HARD, /*!< SPI外设硬件控制NSS*/
} cm_spi_nss_e;

/** 配置 */
typedef struct
{
 cm_spi_mode_e mode; /** SPI主从模式 */
 cm_spi_work_mode_e work_mode; /** 模式 */
 cm_spi_data_width_e data_width; /** SPI传输数据宽度 */
 cm_spi_nss_e nss;/** SPI的NSS管理模式 */
 int clk; /** 时钟频率*/
} cm_spi_cfg_t;
```

SPI接口定义

```
/**
 * @brief 打开spi设备
 *
 * @param [in] dev SPI设备ID
 * @param [in] cfg 配置
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_spi_open(cm_spi_dev_e dev, cm_spi_cfg_t *cfg);

/**
 * @brief spi关闭，去初始化
 *
 * @param [in] dev spi设备
 *
 * @return
 *  = 0 - 成功.
 *  < 0 - 失败, 返回值为错误码.
 *
 * @details More details
 *
 */
int32_t cm_spi_close(cm_spi_dev_e dev);

/**
 * @brief SPI写数据
 *
 * @param [in] dev SPI设备ID
 * @param [in] data 待写数据
 * @param [in] len 待写数据长度
 *
 * @return
 *  = 实际写入长度 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_spi_write(cm_spi_dev_e dev, const void *data, int32_t len);

/**
 * @brief SPI读数据
 *
 * @param [in] dev SPI设备ID
 * @param [out] data 待读数据缓存
 * @param [in] len 长度
 *
 * @return
 *  = 实际读出长度 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
```

```
 * @details More details
 */
int32_t cm_spi_read(cm_spi_dev_e dev, void *data, int32_t len);

/**
 * @brief SPI写数据后读出数据
 *
 * @param [in] dev SPI设备ID
 * @param [in] w_data 待写数据
 * @param [in] w_len 写入长度
 * @param [out] r_data 待读缓存
 * @param [in] r_len 读取长度
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_spi_write_then_read(cm_spi_dev_e dev, const void *w_data, int32_t w_len, void *r_data, int32_t r_len);
```

6.3. 使用示例

SPI DEMO提供了SPI驱动外设的示例程序，详细信息请参考SDK中的cm_demo_spi.c文件。

SPI读写示例

```
/*****************************************************************************
 * Pre-processor Definitions
 ****************************************************************************/
#define CM_SPI_DEMO_LOG  cm_demo_printf
#ifndef min
#define min(A,B) ((A) <= (B) ? (A) : (B))
#endif

#ifndef max
#define max(A,B) ((A) < (B) ? (B) : (A))
#endif

#define SPI_FLASH_BUS  CM_SPI_DEV_1
#define SPI_PAGE_SIZE 256
#define SPI_SECTOR_SIZE 4096
#define SPI_RDID_CMD 0x9f
#define SPI_WREN_CMD 0x06
#define SPI_READ_CMD 0x03
#define SPI_PP_CMD 0x02
#define SPI_RDSR_CMD 0x05
#define SPI_SE_CMD 0x20

/*****************************************************************************
 * Private Types
 ****************************************************************************/

/*****************************************************************************
 * Private Function Prototypes
 ****************************************************************************/

/*****************************************************************************
 * Private Data
 ****************************************************************************/

/*****************************************************************************
 * Private Functions
 ****************************************************************************/

/*****************************************************************************
 * Public Functions
 ****************************************************************************/

static int spi_flash_read(uint32_t addr, char *data, int len);

static int spi_flash_read_status(uint8_t *status)
{
 char w_cmd[] = {SPI_RDSR_CMD};
 int ret = -1;
```

```
 ret = cm_spi_write_then_read(SPI_FLASH_BUS, w_cmd, sizeof(w_cmd), (char *)status, 1);
 osDelay(10);

 return ret;
}

static bool spi_flash_is_busy(void)
{
 uint8_t l_status = {0};

 spi_flash_read_status(&l_status);
 osDelay(10);
 if (l_status & 0x01)
 {
  return true;
 }
 else
 {
  return false;
 }
}

static int spi_flash_read_id(uint32_t *id)
{
 char w_cmd[] = {SPI_RDID_CMD};
 char r_data[10] = {0};
 int ret = -1;

 ret = cm_spi_write_then_read(SPI_FLASH_BUS, w_cmd, sizeof(w_cmd), r_data, 3);
 *id = (r_data[0] << 16) + (r_data[1] << 8) + r_data[2];
 return ret;
}


static int spi_flash_write_enable(void)
{
 char w_cmd[] = {SPI_WREN_CMD};
 osDelay(10);

 return cm_spi_write(SPI_FLASH_BUS, w_cmd, 1);
}

static int spi_flash_page_write(uint32_t addr, char *data, int len)
{
 char w_cmd[300] = {SPI_PP_CMD, (char)(addr >> 16), (char)(addr >> 8), (char)addr};
 int i = 0;
 int ret = -1;

 for (i = 0; i < len; i++)
 {
  w_cmd[i + 4] = *data++;
 }

 ret = spi_flash_write_enable();//enable write

 if (ret < 0)
```

```
 {
  return ret;
 }
 osDelay(10);
 return cm_spi_write(SPI_FLASH_BUS, w_cmd, len + 4);//写入数据
}

static int spi_flash_sector_erase(uint32_t addr)
{
 char w_cmd[4] = {SPI_SE_CMD, (char)(addr >> 16), (char)(addr >> 8), (char)addr};

 if (spi_flash_write_enable() < 0)
 {
  return -1;
 }

 return cm_spi_write(SPI_FLASH_BUS, w_cmd, 4);//写入数据
}

static int spi_flash_write(uint32_t addr, char *data, int len)
{

 int ret = -1;
 char *tmp = cm_malloc(SPI_SECTOR_SIZE);
 int sector_start = addr / SPI_SECTOR_SIZE;
 int sector_end = (addr + len) / SPI_SECTOR_SIZE;
 int i = 0, j = 0, k = 0;

 if ((len <= 0) || (data == NULL))
 {
  cm_free(tmp);
  return -1;
 }

 for (i = sector_start; (i <= sector_end) && (len > 0); i++)
 {
  uint32_t current_sector_addr = i * SPI_SECTOR_SIZE;

  //写入数据至缓冲区
  int current_len = SPI_SECTOR_SIZE - (addr % SPI_SECTOR_SIZE);
  int count = min(len, current_len);
  int addr_start = addr % SPI_SECTOR_SIZE;
  int addr_end = addr_start + count;

  for (j = addr_start; j < addr_end; j++)
  {
   tmp[j] = *data++;
  }
  //擦除sector
  ret = spi_flash_sector_erase(current_sector_addr);

  if (ret < 0)
  {
   CM_SPI_DEMO_LOG("spi erase err\r\n");
   cm_free(tmp);
   return ret;
```

```
    }

    //cm_sleep_ms(150);
    count =0;
    while (spi_flash_is_busy()) //等待擦除完成
    {
     osDelay(10);
     count++;
     if(count > 200)
     {
      cm_free(tmp);
      return -1;
     }
    }

    //缓冲区写入,4kbyte / 256 byte = 16 page
    for (k = 0; k < 16; k++)
    {
     ret = spi_flash_page_write(current_sector_addr + k * SPI_PAGE_SIZE, &tmp[k * SPI_PAGE_SIZE], SPI_PAGE_SIZE);
     if (ret < 0)
     {
      CM_SPI_DEMO_LOG("spi write err\r\n");
      cm_free(tmp);
      return ret;
     }
     count = 0;
     while (spi_flash_is_busy())
     {
      osDelay(10);
      count++;
      if(count > 200)
      {
       cm_free(tmp);
       return -1;
      }
     } //等待写入完成
    }
    //下一轮地址，长度
    addr = (i + 1) * SPI_SECTOR_SIZE;
    len -= count;
   }

   cm_free(tmp);
   return 0;
  }

  static int spi_flash_read(uint32_t addr, char *data, int len)
  {
   char w_cmd[4] = {0};
   int ret = -1;

   while (len > 0) //每次最多读取0xffff字节数据
   {
    w_cmd[0] = SPI_READ_CMD;
    w_cmd[1] = (char)(addr >> 16);
    w_cmd[2] = (char)(addr >> 8);
```

```
 w_cmd[3] = (char)addr;

 uint16_t count = min(0xffff, len);
 ret = cm_spi_write_then_read(SPI_FLASH_BUS, w_cmd, sizeof(w_cmd), data, count);

 if (ret != 0)
 {
  CM_SPI_DEMO_LOG("spi read err");
  return ret;
 }

 addr += 0xffff;
 len -= 0xffff;
 data += 0xffff;
}

 return 0;
}

/*****************************************************************************
* Public Functions
*****************************************************************************/
void cm_test_spi(unsigned char **cmd, int len)
{
 int ret = -1;
 cm_spi_cfg_t config = {CM_SPI_MODE_MASTER,CM_SPI_WOKR_MODE_0,
CM_SPI_DATA_WIDTH_8BIT,CM_SPI_NSS_HARD,CM_SPI_CLK_3_25MHZ};
 uint32_t spi_id = 0;
 uint32_t test_addr = 0;
 char test_str[] = "Hello world!";

 int test_len = strlen(test_str);
 char tmp[1000] = {0};
 CM_SPI_DEMO_LOG("spi test running:%d\r\n",SPI_FLASH_BUS);

 ret = cm_spi_open(SPI_FLASH_BUS, &config);
 if (ret != 0)
 {
  CM_SPI_DEMO_LOG("spi init err:%d\r\n", ret);
  return -1;
 }
 //读取spi id
 ret = spi_flash_read_id(&spi_id);
 if (ret != 0 || spi_id == 0xffffff)
 {
  CM_SPI_DEMO_LOG("spi read id ret:%d spi_id:0x%x err\r\n", ret, spi_id);
  goto spi_stop;
 }
 CM_SPI_DEMO_LOG("spi id =0x%x\n", spi_id);

 //写入测试数据
 ret = spi_flash_write(test_addr, test_str, test_len);
 if (ret != 0)
 {
  CM_SPI_DEMO_LOG("spi write str err:%d\r\n", ret);
  goto spi_stop;
```

```
    }
    CM_SPI_DEMO_LOG("spi write len = %d\n", test_len);
    //读取数据
    ret = spi_flash_read(test_addr, tmp, test_len);

    if (ret != 0)
    {
     CM_SPI_DEMO_LOG("spi read err:%d\r\n", ret);
     goto spi_stop;
    }
    CM_SPI_DEMO_LOG("spi read len = %d,%s\n", strlen(tmp), tmp);
spi_stop:
    cm_spi_close(SPI_FLASH_BUS);

    return 0;
}
```

6.4. 注意事项

使用SPI应注意以下事项：

- SPI仅支持主机模式。

7. I2C

本章主要介绍OpenCPU SDK中外设I2C的相关使用情况。

7.1. 硬件说明

OpenCPU I2C引脚映射详见对应版本的资源综述和硬件设计手册。

7.2. API说明

SDK中提供一套完整的I2C用户编程接口，具体接口定义和说明请查阅cm_i2c.h头文件。

I2C参数结构

```
/** 设备ID */
typedef enum
{
 CM_I2C_DEV_0 = 0, /*!< I2C0 */
 CM_I2C_DEV_1, /*!< I2C1 */
 CM_I2C_DEV_2, /*!< I2C2 */
 CM_I2C_DEV_NUM, /*!< I2C设备个数 */
} cm_i2c_dev_e;


/**
 * @brief I2C 支持主从模式
 */
typedef enum
{
 CM_I2C_MODE_MASTER = 0, /*!< 主模式*/
 CM_I2C_MODE_SLAVER, /*!< 从模式(不支持) */
} cm_i2c_mode_e;

/** 寻址 */
typedef enum
{
 CM_I2C_ADDR_TYPE_7BIT, /*!< 7bit地址*/
 CM_I2C_ADDR_TYPE_10BIT, /*!< 10bit地址(不支持)*/
} cm_i2c_addr_type_e;



/** 配置 */
typedef struct
{
 cm_i2c_addr_type_e addr_type;
 cm_i2c_mode_e mode;
 int clk;
} cm_i2c_cfg_t;
```

I2C接口定义

```
/**
 * @brief 打开I2C设备
 *
 * @param [in] dev I2C设备ID
 * @param [in] cfg 配置
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
```

```
 * @details 仅支持主机模式.
 */
int32_t cm_i2c_open(cm_i2c_dev_e dev, cm_i2c_cfg_t *cfg);


/**
 * @brief 关闭I2C设备
 *
 * @param [in] dev I2C设备ID
 *
 * @return
 * = 0 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_i2c_close(cm_i2c_dev_e dev);


/**
 * @brief I2C写数据
 *
 * @param [in] dev I2C设备ID
 * @param [in] slave_addr i2c设备地址
 * @param [in] data 待写数据
 * @param [in] len 待写数据长度
 *
 * @return
 * = 实际写入长度 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_i2c_write(cm_i2c_dev_e dev, uint16_t slave_addr, const void *data, int32_t len);


/**
 * @brief I2C读数据
 *

 * @param [in] dev i2c设备
 * @param [in] slave_addr i2c设备地址
 * @param [out] data 读取数据存放地址，需先申请内存
 * @param [in] len 读取数据长度
 *
 * @return
 * = 实际读出长度 - 成功 \n
 * < 0 - 失败, 返回值为错误码
 *
 * @details More details
 */
int32_t cm_i2c_read(cm_i2c_dev_e dev, uint16_t slave_addr, void *data, int32_t len);
```

7.3. 使用示例

I2C DEMO提供了I2C驱动外设的示例程序，详细信息请参考SDK中的cm_demo_i2c.c文件。

I2C读写示例

```
/*****************************************************************************
 * Pre-processor Definitions
 ****************************************************************************/
#define CM_DEMO_I2C_LOG cm_demo_printf

#define EPROM_DEV_ADDR 0x50
#define EPROM_I2C_ID  CM_I2C_DEV_0

/*****************************************************************************
 * Private Types
 ****************************************************************************/

/*****************************************************************************
 * Private Function Prototypes
 ****************************************************************************/

/*****************************************************************************
 * Private Data
 ****************************************************************************/


/*****************************************************************************
 * Private Functions
 ****************************************************************************/
/**
 * @brief eeprom写1字节数据
 *
 * @param [in] addr:地址，范围0-0x7fff
 * @param [in] data:待写入数据
 *
 * @return
 * = 0 -成功\n
 * < 0 - 失败, 返回值为错误码.
 *
 * @details More details
 */
static int32_t is24c256_write_byte(uint16_t addr, int8_t data)
{
 int8_t tmp[3] = {0};
 int32_t ret = -1;

 tmp[0] = (addr >> 8) & 0xff;
 tmp[1] = addr & 0xff;
 tmp[2] = data;

 ret = cm_i2c_write (EPROM_I2C_ID, EPROM_DEV_ADDR, tmp, 3);

 if(ret < 0)
 {
```

```
    cm_demo_printf("i2c%d write e2prom err:%d\r\n",g_dev_i2c, ret);
    return RET_ERROR;
  }

  return RET_SUCCESS;
}

/**
* @brief eeprom读1字节数据
*
* @param [in] addr:地址，范围0-0x7fff
* @param [out] data:待读取数据指针
*
* @return
* = 0 - 成功\n
* < 0 - 失败, 返回值为错误码.
*
* @details More details
*/
static int32_t is24c256_read_byte(uint16_t addr, int8_t* data)
{
  int8_t tmp[2] = {0};
  int32_t ret;

  if(data == NULL)
  {
    cm_demo_printf("is24c256_read_bytedata ptr err\r\n");
    return RET_ERROR;
  }

  tmp[0] = (addr >> 8) & 0xff;
  tmp[1] = addr & 0xff;

  ret = cm_i2c_write(EPROM_I2C_ID, EPROM_DEV_ADDR, tmp, 2);
  if(ret < 0)
  {
    cm_demo_printf("i2c read addr err(w):%d\r\n", ret);
    return RET_ERROR;
  }

  ret = cm_i2c_read(EPROM_I2C_ID, EPROM_DEV_ADDR, data, 1);

  if(ret < 0)
  {
    cm_demo_printf("i2c read addr err(r):%d\r\n", ret);
    return RET_ERROR;
  }

  return RET_SUCCESS;
}

/*****************************************************************************
* Public Functions
*****************************************************************************/
void cm_test_i2c(unsigned char **cmd, int len)
{
```

```
cm_i2c_cfg_t config =
{
 CM_I2C_ADDR_TYPE_7BIT,
 CM_I2C_MODE_MASTER, //目前仅支持模式
 CM_I2C_CLK_100KHZ
};//master模式,(100KHZ)

uint16_t eprom_addr = 0x100; //选取所支持的任意E2PROM地址进行测试,可修改
int8_t w_data = 'B',r_data =0;
int32_t ret;

CM_DEMO_I2C_LOG("i2c test start，i2c num:%d!!\n",EPROM_I2C_ID);

ret = cm_i2c_open(EPROM_I2C_ID, &config);
if(ret != 0)
{
 CM_DEMO_I2C_LOG("i2c init err,ret=%d\n", ret);
 return -1;
}
CM_DEMO_I2C_LOG("i2c init ok\n");

//写入测试数据
ret = is24c256_write_byte(eprom_addr, w_data);
if(ret != 0)
{
 CM_DEMO_I2C_LOG("i2c write e2prom err %d\n", (uint32_t)ret);
 cm_i2c_close(EPROM_I2C_ID);
 return -1;
}
CM_DEMO_I2C_LOG("i2c write e2prom ok\n");
osDelay(10);//延时等待写入完成

//读取数据
ret = is24c256_read_byte(eprom_addr, &r_data);
cm_i2c_close(EPROM_I2C_ID);

if(ret != 0)
{
 CM_DEMO_I2C_LOG("i2c read e2prom err:%d\n", ret);
 return -1;
}

CM_DEMO_I2C_LOG("i2c read e2prom,%c\n",r_data);
CM_DEMO_I2C_LOG("i2c test end!!\n");
return 0;
}
```

7.4. 注意事项

使用I2C应注意以下事项：

- I2C模块只支持主模式；
- I2C模块只支持7bit的从机地址；
- I2C模块仅支持100kbps和400kbps，符合标准I2C协议。

8. PWM

本章主要介绍OpenCPU SDK中外设PWM的相关使用情况。

8.1. 硬件说明

OpenCPU PWM引脚映射详见对应版本的资源综述和硬件设计手册。

8.2. API说明

SDK中提供一套完整的PWM用户编程接口，具体接口定义和说明请查阅cm_pwm.h头文件。

PWM参数结构体

```
/** 设备ID */
typedef enum{
 CM_PWM_DEV_0, /*!< 设备0*/
 CM_PWM_DEV_1, /*!< 设备1*/
 CM_PWM_DEV_2, /*!< 设备2*/
 CM_PWM_DEV_NUM
} cm_pwm_dev_e;
```

PWM接口定义

```
/**
 * @brief 打开PWM设备
 *
 * @param [in] dev PWM设备ID
 * @param [in] period 周期(ns)
 * @param [in] period_h 周期高电平占用时间(ns)
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
 *
 * @details PWM由于分频和算法设计存在一定的误差，受限于软件和硬件，因此period和period_h需根据下面备注进行设置\n
 * How to get the PV(The value of after scaled clock cycles per cycle/T): \n
 * InputClockT=13M;PWM_T=period;
 * InputClockT * ('PV(The value of before scaled clock cycles per cycle)) = PWM_T \n
 * 'PV = PWM_T / InputClockT \n
 *  = PWM_T * InputClockF \n
 * PV = 'PV /(prescale + 1) -1 \n
 * How to get the prescale: \n
 * Because the internal clock period counter is 10-bit, to avoid overun. We can use the prescale.\n
 * prescale real value = ('PV - 1(if > 1024, then need prescale)) / 1024 \n
 *
 * How to get the Duty cycle: \n
 * Duty cycle = (PV(the cycles per T) + 1) * ratio(Duty time/PWM_T)
 */
int32_t cm_pwm_open_ns(cm_pwm_dev_e dev, uint32_t period, uint32_t period_h);


/**
 * @brief 关闭PWM设备
 *
 * @param [in] dev PWM设备ID
 *
 * @return
 *  = 0 - 成功 \n
 *  < 0 - 失败, 返回值为错误码
```

```
*
* @details More details
*/
int32_t cm_pwm_close(cm_pwm_dev_e dev);
```

8.3. 使用示例

PWM DEMO提供了PWM使用的功能示例程序，详细信息请参考SDK中的cm_demo_pwm.c文件。

PWM示例程序

```
void cm_test_pwm_start(char **cmd,int len)
{
 uint32_t period,period_h;
 uint8_t dev = CM_PWM_DEV_2;

 cm_demo_printf("pwm test start... ...\n");


 //使用PWM0测试
 dev = atoi((char *)cmd[2]);
 period = atoi((char *)cmd[3]);
 period_h = atoi((char *)cmd[4]);

 cm_demo_printf("pwm=%d period=%d,period_h =%d\n", dev,period,period_h);

 if(dev==0)
 {
  cm_iomux_set_pin_func(OPENCPU_TEST_PWM0_IOMUX);
 }
 else if(dev==1)
 {
  cm_iomux_set_pin_func(OPENCPU_TEST_PWM1_IOMUX);
 }
 else if(dev==2)
 {
  cm_iomux_set_pin_func(OPENCPU_TEST_PWM2_IOMUX);
 }

 if(0 == cm_pwm_open_ns(dev, period,period_h))
 {
  osDelay(2*1000);
 }
 else
 {
  cm_demo_printf("pwm%d open error\n", dev);
 }


 if(cm_pwm_close(dev) == 0)
 {
  cm_demo_printf("[PWM]Stop Success\n");
 }
 else
 {
  cm_demo_printf("[PWM]Stop Fail\n");
 }
}
```

8.4. 注意事项

使用PWM应注意以下事项：

- PWM因为分频的原因，存在一定的误差，所以需要根据备注公式自行计算最优输入参数；
- PWM的主频为12.8M。

9. KEYPAD

本章主要介绍OpenCPU SDK中外设KEYPAD的相关使用情况。

9.1. 硬件说明

OpenCPU KEYPAD引脚映射详见对应版本的资源综述和硬件设计手册。

9.2. API说明

SDK中提供一套完整的KEYPAD用户编程接口，具体接口定义和说明请查阅cm_keypad.h头文件。

KEYPAD接口定义

```
/**
* @brief key map
*
*/
typedef uint32_t cm_keypad_map_t;


/**
* @brief key事件类型
*
*/
typedef enum
{
CM_KEY_EVENT_RELEASE = 0, /*!<按键被释放*/
CM_KEY_EVENT_PRESS = 1, /*!<按键被按下*/
} cm_keypad_event_e;


/**
* @brief keypad事件回调函数
*
* @param [in] key 键值
* @param [in] event 按键事件
*
*/
typedef void (*cm_key_event_callback_t)(cm_keypad_map_t key, cm_keypad_event_e event);


/**
* @brief keypad初始化
*
* @return
* = 0 - 成功 \n
* = -1 - 失败
*/
int32_t cm_keypad_init(void);


/**
* @brief keypad事件回调注册
*
* @param [in] cb keypad回调函数
*
* @return
* = 0 - 成功 \n
* = -1 - 失败
*
* @details 最大允许注册的回调函数个数为 10
```

```
*/
int32_t cm_keypad_register(cm_key_event_callback_t cb);


/**
* @brief keypad事件回调注销
*
* @param [in] cb keypad回调函数
*
* @return
* = 0 - 成功 \n
* = -1 - 失败
*
* @details 最大允许注册的回调函数个数为10
*/
int32_t cm_keypad_unregister(cm_key_event_callback_t cb);


/**
* @brief keypad去初始化
*
*/
void cm_keypad_deinit(void);
```

9.3. 使用示例

KEYPAD DEMO提供了KEYPAD的示例程序，详细信息请参考SDK中的cm_demo_keypad.c文件。

KEYPAD按键示例

```
static void cm_keypad_callback(cm_keypad_map_t key, cm_keypad_event_e event)
{
 cm_demo_printf("key:%02x,state:%d\n",key,event);
}


void cm_test_keypad(unsigned char **cmd, int len)
{
 cm_keypad_register(cm_keypad_callback);

 if(0 == cm_keypad_init())
 {
  cm_demo_printf("cm_keypad_init OK\n");
 }
 else
 {
  cm_demo_printf("cm_keypad_init FAILED\n");
 }
}
```

9.4. 注意事项

使用KEYPAD应注意以下事项：

- cm_keypad_register()接口最大允许注册的回调函数个数为10；
- 每个按键对应的键值（即`cm_keypad_map_t key`）与实际硬件电路接线相关；
- 检测到KEYPAD按键操作时，底层采用中断回调的机制将KEYPAD状态回调至上层应用，KEYPAD回调函数不可阻塞
- 按键初始化之后，形成5×5的矩阵键盘。

10. 编程设计注意

外围接口一般与硬件强相关，需要模组匹配合理的外设设备，且进行软件开发前需详细阅读关联的硬件设计手册。

编程设计时应注意以下几点：

- 对存在引脚复用的外设，应先确认引脚功能以及定义复用功能。
- 单个引脚仅复用其中一种功能，详情参考《资源综述》。