ECR6600 SDK 驱动 API 开发指南

修改记录

版本	时间	说明
1.0	2021-6-15	初始版本

目 录

第一篇 概述

1 引言

1.1 编写目的

本文适用于 ECR6600 芯片 SDK 客户研发人员。包含 ECR6600 RTC 的使用。

1.2 文档约定

无。

2 术语、定义和缩略语

2.1 术语、定义

本文使用的专用术语、定义见表 2.1。

表 2.1 本文使用的专用术语

术语/定义	英文	说 明

2.2 缩略语

本文使用的专用缩略语见表 2.2。

表 2.2 本文使用的专用缩略语

缩略语	原文	中文含义

3 编写依据

本文涉及的相关文档见表 3.1。

表 3.1 涉及的文档

文件名称	版本号	说明

第二篇 UART

4 UART 简介

UART（Universal Asynchronous Receiver/Transmitter）通用异步收发传输器，UART 作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位地传输。是在应用程序开发过程中使用频率最高的数据总线。

UART 串口的特点是将数据一位一位地顺序传送，只要 2 根传输线就可以实现双向通信，一根线发送数据的同时用另一根线接收数据。UART 串口通信有几个重要的参数，分别是波特率、起始位、数据位、停止位和奇偶检验位，对于两个使用 UART 串口通信的端口，这些参数必须匹配，否则通信将无法正常完成。

UART 串口传输的数据格式如下图所示：

- 起始位：表示数据传输的开始，电平逻辑为 “0” 。
- 数据位：可能值有 5、6、7、8、9，表示传输这几个 bit 位数据。一般取值为 8，因为一个 ASCII 字符值为 8 位。
- 奇偶校验位：用于接收方对接收到的数据进行校验，校验 “1” 的位数为偶数(偶校验)或奇数(奇校验)，以此来校验数据传送的正确性，使用时不需要此位也可以。
- 停止位： 表示一帧数据的结束。电平逻辑为 “1”。
- 波特率：串口通信时的速率，它用单位时间内传输的二进制代码的有效位(bit) 数来表示，其单位为每秒比特数 bit/s(bps)。常见的波特率值有 4800、9600、14400、38400、115200 等，数值越大数据传输的越快，波特率为 115200 表示每秒钟传输 115200 位数据。

5 UART 使用方法

应用程序可以通过下述接口访问 UART 硬件：

函数	描述
drv_uart_open()	打开 UART 设备
drv_uart_write()	UART 发送数据
drv_uart_read()	UART 接收数据
drv_uart_ioctrl()	控制 UART 设备
chip_uart0_pinmux_cfg()	设置 UART0 的硬件引脚复用
chip_uart1_pinmux_cfg()	设置 UART1 的硬件引脚复用
chip_uart2_pinmux_cfg()	设置 UART2 的硬件引脚复用

5.1 打开 UART 设备

使用 UART 前，需要打开 UART 设备。在打开 UART 设备时，接口函数会打开 UART 的时钟，并进行设备的初始化。

函数原型：

```
int drv_uart_open(E_DRV_UART_NUM uart_num, T_DRV_UART_CONFIG * cfg)
```

参数说明：

输入参数	说明
uart_num	指定打开的 UART 编号，采用 E_DRV_UART_NUM 类型
cfg	指定打开 UART 时使用的配置信息，采用 T_DRV_UART_CONFIG 的指针类型
返回	——
0	设备打开成功

其它	设备打开失败

E_DRV_UART_NUM 原型

```
typedef enum _E_DRV_UART_NUM
{
E_UART_NUM_0 = 0,
E_UART_NUM_1 = 1,
E_UART_NUM_2 = 2,
E_UART_NUM_MAX
} E_DRV_UART_NUM;
```

由于 6600 有三个 UART，所以对应枚举类型里有 UART0、UART1 和 UART2. 根据实际情况选择对应的 UART

T_DRV_UART_CONFIG 原型

```
typedef struct _T_DRV_UART_CONFIG
{
unsigned int uart_baud_rate;        /* 波特率 */
unsigned int uart_data_bits;        /* 数据位 */
unsigned int uart_stop_bits;        /* 停止位 */
unsigned int uart_parity_bit;       /* 奇偶校验位 */
unsigned int uart_flow_ctrl;        /* 流控 */
unsigned int uart_tx_mode;          /* 发送模式 */
unsigned int uart_rx_mode;          /* 接收模式位 */
unsigned int uart_rx_buf_size;      /* 接收模式 */
} T_DRV_UART_CONFIG;
```

打开 UART 设备前需要预先设置结构体里的配置信息，这些配置信息包括 UART 的常用硬件配置参数和软件使用 UART 的数据收发模式。

uart.h 头文件中提供的配置参数可以取值，下面进行具体的说明：

- ***uart_data_bits 的取值***

```
/* 数据位可取值 */
#define UART_DATA_BITS_5                0x00
```

```
#define UART_DATA_BITS_6        0x01
#define UART_DATA_BITS_7        0x02
#define UART_DATA_BITS_8        0x03
```

- *uart_stop_bits 的取值*

```
/* 停止位可取值 */
#define UART_STOP_BITS_1        0x00
#define UART_STOP_BITS_OTHER    0x04
```

对于 bit 为其它情况，如果数据位选择 UART_DATA_BITS_5，那么停止位是 1.5 bits ；如果数据位非 UART_DATA_BITS_5，那么停止位是 2 bits

- *uart_parity_bit 的取值*

```
/* 奇偶校验位可取值 */
#define UART_PARITY_BIT_NONE    0x00
#define UART_PARITY_BIT_ODD     0x08
#define UART_PARITY_BIT_EVEN    0x18
```

- *uart_flow_ctrl 的取值*

```
/* 流控可取值 */
#define UART_FLOW_CONTROL_DISABLE    0x00
#define UART_FLOW_CONTROL_ENABLE     0x20
```

- *uart_tx_mode 的取值*

```
/* 发送模式可取值 */
#define UART_TX_MODE_POLL       0x00    /* 轮询发送模式 */
#define UART_TX_MODE_STREAM     0x01    /* 流发送模式 */
#define UART_TX_MODE_INTR       0x02    /* 中断发送模式 */
#define UART_TX_MODE_DMA        0x03    /* DMA 发送模式 */
```

UART 发送数据包括 4 种模式： 轮询发送模式，流发送模式，中断发送模式，DMA 发送模式。

➢ 轮询发送模式： CPU 循环查询 UART fifo 状态，不满则发送一字节数据，直到数据发送完成。适用于批量数据的传输，对缓存属性没有限制，但是会相对消耗 CPU

- 流发送模式： 当输出的字符是 "\n" （对应 16 进制值为 0x0A）时，自动在前面输出一个 "\r"（对应 16 进制值为 0x0D） 做分行。适用于向串口终端输出字符串
- 中断发送模式： 通过 UART 中断激励发送数据。适用于低数据量的传输（如命令），但是实时性高
- DMA 发送模式： 通过预先配置 DMA，让硬件无需 CPU 参与自动完成数据的发送。适用于批量数据的传输，且基本不消耗 CPU，但是缓存属性有限制，必须是事先申请的 DMA 可访问缓存。

- ***uart_rx_mode 的取值***

```
/* 接收模式可取值 */
#define UART_RX_MODE_INTR 0x00          /* 中断接收模式 */
#define UART_RX_MODE_DMA 0x01           /* DMA 接收模式 */
#define UART_RX_MODE_USER 0x02          /* 自定义接收模式 */
```

UART 接收数据包括 3 种模式： 中断接收模式，DMA 接收模式，自定义接收。

- 中断接收模式： 以中断的方式接收数据。适用于批量数据的传输，对缓存属性没有限制，但是会相对消耗 CPU
- DMA 接收模式：通过预先配置 DMA 来接收指定长度的数据适用于批量数据的传输，基本不消耗 CPU，但是缓存必须要 DMA 可访问，并且最好预先知道数据长度
- 自定义接收模式： 目前仅用于实现控制台消息的接收

！注意事项 -- uart_rx_buf_size
只有选择中断接收模式时，uart_rx_buf_size 才有意义 打开设备时会按 uart_rx_buf_size 给接收分配一块缓冲区，打开之前大小无法动态修改 若一次性数据接收字节数很多，没有及时读取数据，那么缓冲区的数据将会被新接收到的数据覆盖，造成数据丢失，建议调大缓冲区

接口使用示例

1. 以中断接收及轮询发送模式打开串口

```
T_DRV_UART_CONFIG config;


config.uart_baud_rate = 115200;
```

```
config.uart_data_bits = UART_DATA_BITS_8;
config.uart_stop_bits = UART_STOP_BITS_1;
config.uart_parity_bit = UART_PARITY_BIT_NONE;
config.uart_flow_ctrl = UART_FLOW_CONTROL_DISABLE;
config.uart_tx_mode = UART_TX_MODE_POLL;
config.uart_rx_mode = UART_RX_MODE_INTR;
config.uart_rx_buf_size = 1024;

drv_uart_open(E_UART_NUM_1, &config);
```

2. 以 DMA 接收及 DMA 发送模式打开串口

```
T_DRV_UART_CONFIG config;

config.uart_baud_rate = 460800;
config.uart_data_bits = UART_DATA_BITS_8;
config.uart_stop_bits = UART_STOP_BITS_1;
config.uart_parity_bit = UART_PARITY_BIT_NONE;
config.uart_flow_ctrl = UART_FLOW_CONTROL_ENABLE;
config.uart_tx_mode = UART_TX_MODE_DMA;
config.uart_rx_mode = UART_RX_MODE_DMA;

drv_uart_open(E_UART_NUM_1, &config);
```

5.2 UART 发送数据

通过 UART 口发送数据可以通过如下接口完成

函数原型：

```
int drv_uart_write(E_DRV_UART_NUM uart_num, char * buf, unsigned int len)
```

参数说明：

输入参数	说明
uart_num	指定打开的 UART 编号，采用 E_DRV_UART_NUM 类型
buf	缓冲区指针，缓冲区存放带发送的数据
len	带发送数据的长度
返回	——
0	数据发送成功
其它	数据发送失败

E_DRV_UART_NUM 原型

```
typedef enum _E_DRV_UART_NUM
{
E_UART_NUM_0 = 0,
E_UART_NUM_1 = 1,
E_UART_NUM_2 = 2,
E_UART_NUM_MAX
} E_DRV_UART_NUM;
```

由于 6600 有三个 UART，所以对应枚举类型里有 UART0、UART1 和 UART2. 根据实际情况选择对应的 UART

! 注意事项

使用 DMA 模式发送时，缓冲区必须要 DMA 可访问。

申请 DMA 可访问缓存的方法，可参考如下：

```
char uart_loop_buffer[TEST_UART_LOOP_BUF_SIZE] __attribute__((section(".dma.data")));
```

增加 "__attribute__((section(".dma.data")))" 属性，可以让 DMA 能够直接访问 uart_loop_buffer 缓存

5.3 UART 接收数据

函数原型：

```
int drv_uart_read(E_DRV_UART_NUM uart_num, char * buf, unsigned int len, unsigned int ms_timeout)
```

参数说明：

输入参数	说明
uart_num	指定打开的 UART 编号，采用 E_DRV_UART_NUM 类型
buf	缓冲区指针，用于接收数据的缓冲区基地址
len	带发送数据的长度
ms_timeout	超时时间设置，单位毫秒
返回	——
非负值	实际收到的数据长度
其它	接收失败

E_DRV_UART_NUM 原型

```
typedef enum _E_DRV_UART_NUM
{
E_UART_NUM_0 = 0,
E_UART_NUM_1 = 1,
E_UART_NUM_2 = 2,
E_UART_NUM_MAX
} E_DRV_UART_NUM;
```

由于 6600 有三个 UART，所以对应枚举类型里有 UART0、UART1 和 UART2. 根据实际情况选择对应的 UART

! 注意事项

使用 DMA 模式发送时，缓冲区必须要 DMA 可访问。

申请 DMA 可访问缓存的方法，可参考如下：

```
char uart_loop_buffer[TEST_UART_LOOP_BUF_SIZE] __attribute__((section(".dma.data")));
```

增加 "__attribute__((section(".dma.data")))" 属性，可以让 DMA 能够直接访问 uart_loop_buffer 缓存

5.4 控制 UART 设备

函数原型：

```
int drv_uart_ioctrl(E_DRV_UART_NUM uart_num, int event, void * arg)
```

参数说明：

输入参数	说明
uart_num	指定打开的 UART 编号，采用 E_DRV_UART_NUM 类型
event	控制事件的类型
arg	控制参数，具体含义根据 event 来定
返回	——
0	执行成功
其它	执行失败

E_DRV_UART_NUM 原型

```
typedef enum _E_DRV_UART_NUM
{
E_UART_NUM_0 = 0,
E_UART_NUM_1 = 1,
E_UART_NUM_2 = 2,
E_UART_NUM_MAX
} E_DRV_UART_NUM;
```

由于 6600 有三个 UART，所以对应枚举类型里有 UART0、UART1 和 UART2. 根据实际情况选择对应的 UART

event 参数可取值

```
#define DRV_UART_CTRL_SET_BAUD_RATE 1
```

设置 UART 的波特率，相应 arg 参数则为指向波特率值的指针

5.5 设置 UART 硬件 PIN 脚

5.5.1 设置 UART0 的 PIN 脚

UART0 的硬件引脚有两组，使用时需要提前设置，并二选一。

函数原型：

```
int chip_uart0_pinmux_cfg(int rx_num, int tx_num, int flow_ctrl_en)
```

参数说明：

输入参数	说明
rx_num	设置接收使用 PIN 脚
tx_num	设置发送使用 PIN 脚
flow_ctrl_en	设置 UART 流控 PIN 脚
返回	——
0	设置成功
其它	设置失败

rx_num 可设置取值：

#define UART0_RX_USED_GPIO5 0 #define UART0_RX_USED_GPIO21 1

tx_num 可设置取值：

#define UART0_TX_USED_GPIO6 0 #define UART0_TX_USED_GPIO22 1

flow_ctrl_en 可设置取值：

如果设置为 1，就会额外设置 UART 的 CTS 和 RTS 脚；

如果设置为 0，不会有任务额外操作

5.5.2 设置 UART1 的 PIN 脚

UART1 只有一组固定的 PIN 脚，支持流控。

函数原型：

```
void chip_uart1_pinmux_cfg(int flow_ctrl_en)
```

参数说明：

输入参数	说明
flow_ctrl_en	设置 UART 流控 PIN 脚
返回	——
无	

flow_ctrl_en 可设置取值：

如果设置为 1，就会额外设置 UART 的 CTS 和 RTS 脚；

如果设置为 0，不会有任务额外操作

5.5.3 设置 UART2 的 PIN 脚

UART2 只有一组固定的 PIN 脚，且不支持流控。

函数原型：

```
void chip_uart2_pinmux_cfg(void)
```

参数说明：

输入参数	说明
无	
返回	——
无	

6 UART 使用示例

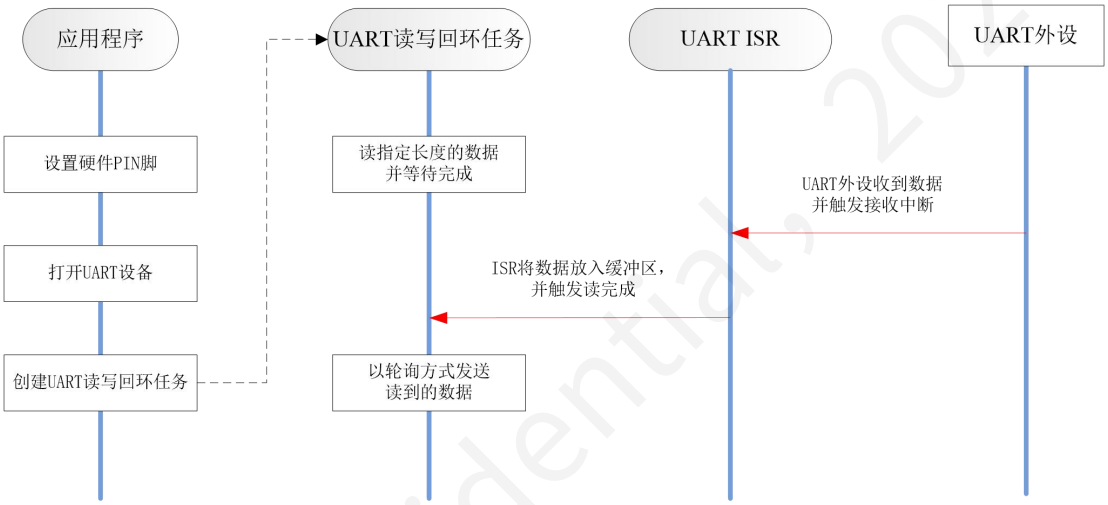
6.1 中断接收及轮询发送

示例代码的主要步骤如下所示：

1. 设置 UART 的硬件引脚复用功能为 UART
2. 预设 UART 的硬件配置参数，以及软件收发使用的模式

3. 根据预设参数打开 UART
4. 创建 UART 回环任务
5. 回环任务会先使用 UART 读取指定长度的数据。如何缓冲区没有足够的数据，则任务挂起等待，当 UART ISR 接收到足够的数据时会唤醒任务；当有足够数据时，则读取直接完成，不会阻塞。
6. 以轮询的方式将数据从 UART 口发送出去

运行程序如下图：



参考代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "cli.h"
#include "oshal.h"
#include "uart.h"
#include "chip_pinmux.h"


#define TEST_UART_LOOP_BUF_SIZE 1024
```

```
static                  char                  uart_loop_buffer[TEST_UART_LOOP_BUF_SIZE]
        __attribute__((section(".dma.data")));

void test_uart_loop_task(void)
{
int len;

cli_printf("test_uart_task: entry~\n");

while(1)
{
        len = drv_uart_read(E_UART_NUM_1, uart_loop_buffer,
                TEST_UART_LOOP_BUF_SIZE, 80000);
        if (len > 0)
        {
                cli_printf("test_uart_task: get %d Bytes\n", len);
                drv_uart_write(E_UART_NUM_1, uart_loop_buffer, len );
        }
        else
        {
                cli_printf("test_uart_task: no data & timeout\n");
        }
 }
 }

 int test_uart_loop(cmd_tbl_t *h, int argc, char *argv[])
{
T_DRV_UART_CONFIG config;

chip_uart1_pinmux_cfg(0);
config.uart_baud_rate = 115200;
config.uart_data_bits = UART_DATA_BITS_8;
```

```
config.uart_stop_bits = UART_STOP_BITS_1;
config.uart_parity_bit = UART_PARITY_BIT_NONE;
config.uart_flow_ctrl = UART_FLOW_CONTROL_DISABLE;
config.uart_tx_mode = UART_TX_MODE_DMA;
config.uart_rx_mode = UART_RX_MODE_INTR;
config.uart_rx_buf_size = 1024;

drv_uart_open(E_UART_NUM_1, &config);

os_task_create("uart-loop-task",  FHOST_CLI_PRIORITY,  FHOST_CLI_STACK_SIZE,
        (main_t)test_uart_loop_task, NULL);

return CMD_RET_SUCCESS;
}

CLI_CMD(uartloop, test_uart_loop, "uart loop test", "uartloop");
```

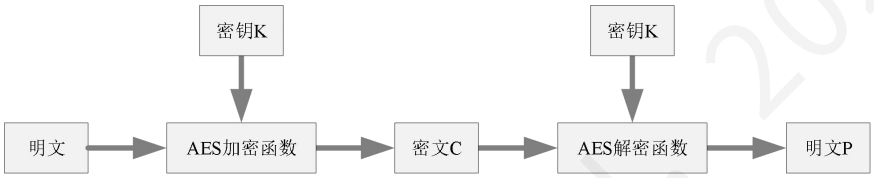
！注意事项
本示例采用了中断接收和轮询发送的模式，实际上本例中接收和发送采用其它模式的配置也是同样可行的

第三篇 CRYPTO

7 AES

7.1 AES 简介

高级加密标准(AES, Advanced Encryption Standard)为最常见的对称加密算法，一种对称秘钥分组加/解密算法，相较非对称加密算法，AES 加密算法可以以更快的速率对大量的数据信息进行加密，其加密流程如图所示，加解密钥匙一样。



加解密流程示意图

AES 加速模块实现数据分组长度为 128 比特，密钥长度为 128/192/256 比特的 AES 加密或解密功能。AES 加速模块作为 AHB 的一个从设备，可以实现数据以及秘钥的高效访问，加/解密均为软件通过 AHB 总线读写相应的寄存器实现，加密完成后通过中断上报或者软件查询，其主要功能和特性为：

- 支持同一个KEY加解密数据包时，软件只需在开始配置一次密钥，而不是在每个128-bit 数据加解密时，都要重新配置KEY；如果加密和解密的密钥相同，也不需要重新配置。但在AES128、AES192或AES256之间切换时，必须重新配置完整的密钥；
- AES 加解密支持电子密码本模式（Electronic codebook，ECB）以及密码分组链接模式（Cipher Block Chaining，CBC）两种模式，CBC 模式需要配置初始化向量(Initialization Vector, IV)。
- 支持加解密数据、密钥以及 IV 值的字节序可配置；
- 支持两种启动方式：软件启动模式，软件配置控制寄存器的启动信号开始硬件加解密工作；硬件启动模式，当硬件启动生效时，每配置完 128 比特数据就启动开始加解密工作。

7.2 AES 使用方法

AES 加解密计算实现为线程安全，所有避免在中断处理中使用 AES 相关 API 接口。

7.2.1 初始化 AES

初始化 AES 使用中要用到的系统资源。如果在 menuconfig 时配置了 AES，那么系统启动时会自动调用 AES 初始化，所以使用 AES 前无需额外调用该接口。

函数原型：

```
int drv_aes_init(void)
```

参数说明：

输入参数	说明
无	
返回	——
0	执行成功
其它	执行失败

7.2.2 加锁 AES

在使用之前，需要给 AES 加锁，防止有其它任务同时操作 AES，导致 AES 硬件使用出现问题。因此，使用 AES 前必须先调用该接口。

函数原型：

```
int drv_aes_lock(void)
```

参数说明：

输入参数	说明
无	

返回	——
0	执行成功
其它	执行失败

7.2.3 解锁 AES

在使用完成后，需要给 AES 解锁，这样其它任务才可以使用 AES。

函数原型：

```
int drv_aes_unlock(void)
```

参数说明：

输入参数	说明
无	
返回	——
0	执行成功
其它	执行失败

7.2.4 设置在 ECB 模式下加/解密

此接口用于设置 AES 在 ECB 方式下进行加密或者解密，同时设置相应的密钥及位数。如果参数不变的话，调用该接口配置一次即可，无需每次加/解密前都调用一次。

函数原型：

```
int drv_aes_ecb_setkey(const unsigned char *key, unsigned int keybits, int mode)
```

参数说明：

输入参数	说明
key	存放密钥的地址，采用 unsigned char 的指针类型
keybits	密钥的长度，选择 128，192 或 256 比特
mode	AES 的模式，选择加密方式或解密方式

返回	——
0	执行成功
其它	执行失败

keybits 参数可取值：

```
/*  密钥长度选择  */
#define DRV_AES_KEYBITS_128          128
#define DRV_AES_KEYBITS_192          192
#define DRV_AES_KEYBITS_256          256
```

mode 参数可取值：

```
/* AES 模式选择  */
#define AES_REG_CTRL_ENCRYPTION  (1<<0)     //选择加密模式
#define AES_REG_CTRL_DECRYPTION  (0<<0)     //选择解密模式
```

7.2.5 设置在 CBC 方式下加/解密

此接口用于设置 AES 在 CDC 方式下进行加密或者解密，同时设置相应的密钥及位数。如果参数不变的话，调用该接口配置一次即可，后面可以连续进行加/解密（硬件会自动根据本次的计算结果更新 iv 参数）。

函数原型：

```
int drv_aes_cbc_setkey(const unsigned char *key, unsigned int keybits, int mode, const unsigned char *iv)
```

参数说明：

输入参数	说明
key	存放密钥的地址，采用 unsigned char 的指针类型
keybits	密钥的长度，选择 128，192 或 256 比特
mode	AES 的模式，选择加密方式或解密方式
iv	存放初始向量的地址，采用 unsigned char 的指针类型

返回	——
0	执行成功
其它	执行失败

keybits 参数可取值：

```
/*  密钥长度选择  */
#define DRV_AES_KEYBITS_128          128
#define DRV_AES_KEYBITS_192          192
#define DRV_AES_KEYBITS_256          256
```

mode 参数可取值：

```
/* AES 模式选择  */
#define AES_REG_CTRL_ENCRYPTION  (1<<0)     //选择加密模式
#define AES_REG_CTRL_DECRYPTION  (0<<0)     //选择解密模式
```

7.2.6 AES 加密/解密

此接口用于在加密或解密时，输入要加密/解密的消息后，经过硬件自动加解密，并输出加解密后的数据。（如果完成了加/解密计算，需要解锁 AES。）

函数原型：

void drv_aes_crypt(const unsigned char input[16], unsigned char output[16])

输入参数	说明
input[16]	存放待加密/解密的数据
output[16]	存放加密/解密完成后的数据
返回	——
无	

7.3 AES 使用示例

示例代码的主要步骤如下所示：

1. AES 加锁

2. 设置 AES 模块的硬件配置，包括控制寄存器和 IV 寄存器
3. 进行 AES 加密或解密计算
4. AES 解锁

参考代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "cli.h"
#include "oshal.h"
#include "chip_pinmux.h"
#include "aes.h"

/**Definition of encryption and decryption types*/
#define AES_ECB    0
#define AES_CBC    1

/**The test key length is 128 bits*/
static const unsigned char test_aes_key_128[16] =
{
        0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0xaf, 0x7f,
        0x67, 0x98
};

/**The test key length is 192 bits*/
static const unsigned char test_aes_key_192[24] =
{
        0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0xaf, 0x7f,
        0x67, 0x98,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
};

/**The test key length is 256 bits*/
```

```
static const unsigned char test_aes_key_256[32] =
{
        0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0xaf, 0x7f,
        0x67, 0x98,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
        0x0f, 0x00
};

/**Test data 1*/
static const unsigned char test_aes_plain_text_1[16] =
{
        0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54,
        0x32, 0x10
};

/**Test data 2*/
static const unsigned char test_aes_plain_text_2[16] =
{
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
        0x0e, 0x0f
};

/**Encrypted initialization vector*/
static const unsigned char test_aes_iv_encrypt[16] =
{
        0xEB, 0xDF, 0x2B, 0x26, 0xD0, 0xC8, 0x54, 0x18, 0x29, 0x14, 0x7E, 0x82, 0x49, 0xB5,
        0x43, 0x82
};



/**In ECB mode,the key length is 128 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_128[16] =
{
```

```
        0xff, 0x0b, 0x84, 0x4a, 0x08, 0x53, 0xbf, 0x7c, 0x69, 0x34, 0xab, 0x43, 0x64, 0x14,
        0x8f, 0xb9
};

/**In ECB mode,the key length is 192 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_192[16] =
{
        0x54, 0xE2, 0x49, 0x7B, 0x07, 0xC6, 0xA5, 0x3E, 0x55, 0x17, 0xA8, 0xAA, 0xAD,
        0x49, 0x2E, 0x39
};

/**In ECB mode,the key length is 256 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_256[16] =
{
        0xD2, 0xD0, 0xC2, 0xB2, 0x17, 0xB5, 0x9F, 0xF1, 0xBB, 0xD1, 0x24, 0x18, 0xD3,
        0x7B, 0x0E, 0x9B
};

/**In ECB mode,the key length is 128 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_ecb_128[16] =
{
        0xEB, 0xDF, 0x2B, 0x26, 0xD0, 0xC8, 0x54, 0x18, 0x29, 0x14, 0x7E, 0x82, 0x49, 0xB5,
        0x43, 0x82
};

/**In ECB mode,the key length is 192 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_ecb_192[16] =
{
        0xC5, 0x47, 0x7F, 0x79, 0x92, 0xA7, 0x3A, 0x60, 0x98, 0xFD, 0x76, 0xF0, 0x75, 0x9D,
        0x0E, 0xEB
};

/**In ECB mode,the key length is 256 bits, the encrypted data corresponding to test data 2*/
```

```
static const unsigned char test_aes_cipher_text_2_ecb_256[16] =
{
        0xCB, 0xE2, 0x26, 0xB4, 0x7B, 0x38, 0x9D, 0x84, 0x86, 0x62, 0x7C, 0x02, 0x7D, 0x71,
        0x45, 0x59
};



/**In CBC mode,the key length is 128 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_128[16] =
{
        0xA0, 0x5C, 0x1B, 0xEE, 0xD6, 0x0E, 0x11, 0xA9, 0x2F, 0xD0, 0xBF, 0x52, 0xF3,
        0xD2, 0xC8, 0x14
};

/**In CBC mode,the key length is 192 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_192[16] =
{
        0x99, 0x9C, 0x35, 0xAA, 0x76, 0x3E, 0xB1, 0xE5, 0x4F, 0xF8, 0xCE, 0x98, 0xA2,
        0xEB, 0x6C, 0x0B
};

/**In CBC mode,the key length is 256 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_256[16] =
{
        0xAF, 0xBB, 0x5D, 0x4F, 0x4E, 0x5D, 0x8B, 0x5D, 0x59, 0x9D, 0x6F, 0xE2, 0x5B,
        0x0F, 0xF5, 0xBA
};

/**In CBC mode,the key length is 128 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_128[16] =
{
        0x8D, 0xD6, 0x98, 0x09, 0xBF, 0xE5, 0xAC, 0x09, 0xBE, 0x0B, 0x96, 0xE4, 0xF4,
        0x69, 0x3B, 0x33
```

```
};

/**In CBC mode,the key length is 192 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_192[16] =
{
        0x27, 0xE6, 0x9E, 0x28, 0x1D, 0x5C, 0xDD, 0xA9, 0x93, 0xD6, 0x35, 0x92, 0x12,
        0x1A, 0xC0, 0xEC
};

/**In CBC mode,the key length is 256 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_256[16] =
{
        0x38, 0x55, 0xAD, 0xF7, 0x12, 0x2F, 0x7A, 0xC8, 0xBA, 0xAE, 0x9B, 0xD0, 0x59,
        0x4F, 0x77, 0xE1
};



/**The test key length is 192 bits*/
static const unsigned char test_aes_key_192[24] =
{
        0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0xaf, 0x7f,
        0x67, 0x98,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
};

/**The test key length is 256 bits*/
static const unsigned char test_aes_key_256[32] =
{
        0x0f, 0x15, 0x71, 0xc9, 0x47, 0xd9, 0xe8, 0x59, 0x0c, 0xb7, 0xad, 0xd6, 0xaf, 0x7f,
        0x67, 0x98,
        0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
        0x0f, 0x00
};
```

```
/**Test data 1*/
static const unsigned char test_aes_plain_text_1[16] =
{
        0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54,
        0x32, 0x10
};

/**Test data 2*/
static const unsigned char test_aes_plain_text_2[16] =
{
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
        0x0e, 0x0f
};

/**Encrypted initialization vector*/
static const unsigned char test_aes_iv_encrypt[16] =
{
        0xEB, 0xDF, 0x2B, 0x26, 0xD0, 0xC8, 0x54, 0x18, 0x29, 0x14, 0x7E, 0x82, 0x49, 0xB5,
        0x43, 0x82
};


/**In ECB mode,the key length is 128 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_128[16] =
{
        0xff, 0x0b, 0x84, 0x4a, 0x08, 0x53, 0xbf, 0x7c, 0x69, 0x34, 0xab, 0x43, 0x64, 0x14,
        0x8f, 0xb9
};

/**In ECB mode,the key length is 192 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_192[16] =
{
```

```
        0x54, 0xE2, 0x49, 0x7B, 0x07, 0xC6, 0xA5, 0x3E, 0x55, 0x17, 0xA8, 0xAA, 0xAD,
        0x49, 0x2E, 0x39
};

/**In ECB mode,the key length is 256 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_ecb_256[16] =
{
        0xD2, 0xD0, 0xC2, 0xB2, 0x17, 0xB5, 0x9F, 0xF1, 0xBB, 0xD1, 0x24, 0x18, 0xD3,
        0x7B, 0x0E, 0x9B
};



/**In ECB mode,the key length is 128 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_ecb_128[16] =
{
        0xEB, 0xDF, 0x2B, 0x26, 0xD0, 0xC8, 0x54, 0x18, 0x29, 0x14, 0x7E, 0x82, 0x49, 0xB5,
        0x43, 0x82
};

/**In ECB mode,the key length is 192 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_ecb_192[16] =
{
        0xC5, 0x47, 0x7F, 0x79, 0x92, 0xA7, 0x3A, 0x60, 0x98, 0xFD, 0x76, 0xF0, 0x75, 0x9D,
        0x0E, 0xEB
};

/**In ECB mode,the key length is 256 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_ecb_256[16] =
{
        0xCB, 0xE2, 0x26, 0xB4, 0x7B, 0x38, 0x9D, 0x84, 0x86, 0x62, 0x7C, 0x02, 0x7D, 0x71,
        0x45, 0x59
};
```

```
/**In CBC mode,the key length is 128 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_128[16] =
{
        0xA0, 0x5C, 0x1B, 0xEE, 0xD6, 0x0E, 0x11, 0xA9, 0x2F, 0xD0, 0xBF, 0x52, 0xF3,
        0xD2, 0xC8, 0x14
};

/**In CBC mode,the key length is 192 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_192[16] =
{
        0x99, 0x9C, 0x35, 0xAA, 0x76, 0x3E, 0xB1, 0xE5, 0x4F, 0xF8, 0xCE, 0x98, 0xA2,
        0xEB, 0x6C, 0x0B
};

/**In CBC mode,the key length is 256 bits, the encrypted data corresponding to test data 1*/
static const unsigned char test_aes_cipher_text_1_cbc_256[16] =
{
        0xAF, 0xBB, 0x5D, 0x4F, 0x4E, 0x5D, 0x8B, 0x5D, 0x59, 0x9D, 0x6F, 0xE2, 0x5B,
        0x0F, 0xF5, 0xBA
};


/**In CBC mode,the key length is 128 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_128[16] =
{
        0x8D, 0xD6, 0x98, 0x09, 0xBF, 0xE5, 0xAC, 0x09, 0xBE, 0x0B, 0x96, 0xE4, 0xF4,
        0x69, 0x3B, 0x33
};

/**In CBC mode,the key length is 192 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_192[16] =
{
```

```
        0x27, 0xE6, 0x9E, 0x28, 0x1D, 0x5C, 0xDD, 0xA9, 0x93, 0xD6, 0x35, 0x92, 0x12,
        0x1A, 0xC0, 0xEC
};

/**In CBC mode,the key length is 256 bits, the encrypted data corresponding to test data 2*/
static const unsigned char test_aes_cipher_text_2_cbc_256[16] =
{
        0x38, 0x55, 0xAD, 0xF7, 0x12, 0x2F, 0x7A, 0xC8, 0xBA, 0xAE, 0x9B, 0xD0, 0x59,
        0x4F, 0x77, 0xE1
};

   static int aes_encryption(cmd_tbl_t *t, int argc, char *argv[])
{
        unsigned char cbuf[16];
unsigned int type = strtoul(argv[1], NULL, 0);
        unsigned int mode = strtoul(argv[2], NULL, 0);
        drv_aes_lock();
          switch (mode)
        {
              case 128:
                    drv_aes_ecb_setkey(test_aes_key_128,              DRV_AES_KEYBITS_128,
        DRV_AES_MODE_ENC);

                    drv_aes_crypt(test_aes_plain_text_1, cbuf);
                    if (memcmp(test_aes_cipher_text_1_ecb_128, cbuf, 16) == 0)
                    {
                          cli_printf(">> aes_ecb_encrypt keybit_128 data-1 pass!\r\n");
                    }
                    else
                    {
                          cli_printf(">> aes_ecb_encrypt keybit_128 data-1 failed!\r\n");
                    }
```

```
                drv_aes_crypt(test_aes_plain_text_2, cbuf);
                if (memcmp(test_aes_cipher_text_2_ecb_128, cbuf, 16) == 0)
                {
                    cli_printf(">> aes_ecb_encrypt keybit_128 data-2 pass!\r\n\r\n");
                }
                else
                {
                    cli_printf(">> aes_ecb_encrypt keybit_128 data-2 failed!\r\n\r\n");
                }

                break;

            case 192:
                drv_aes_ecb_setkey(test_aes_key_192,          DRV_AES_KEYBITS_192,
DRV_AES_MODE_ENC);
                drv_aes_crypt(test_aes_plain_text_1, cbuf);
 if (memcmp(test_aes_cipher_text_1_ecb_192, cbuf, 16) == 0)
 {
    cli_printf(">> aes_ecb_encrypt keybit_192 data-1 pass!\r\n");
 }
 else
 {
    cli_printf(">> aes_ecb_encrypt keybit_192 data-1 failed!\r\n");
 }

                drv_aes_crypt(test_aes_plain_text_2, cbuf);
                if (memcmp(test_aes_cipher_text_2_ecb_192, cbuf, 16) == 0)
                {
                    cli_printf(">> aes_ecb_encrypt keybit_192 data-2 pass!\r\n\r\n");
                }
                else
```

```
            {
                cli_printf(">> aes_ecb_encrypt keybit_192 data-2 failed!\r\n\r\n");
            }

            break;

        case 256:
            drv_aes_ecb_setkey(test_aes_key_256,          DRV_AES_KEYBITS_256,
DRV_AES_MODE_ENC);
            drv_aes_crypt(test_aes_plain_text_1, cbuf);
 if (memcmp(test_aes_cipher_text_1_ecb_256, cbuf, 16) == 0)
 {
    cli_printf(">> aes_ecb_encrypt keybit_256 data-1 pass!\r\n");
 }
 else
 {
    cli_printf(">> aes_ecb_encrypt keybit_256 data-1 failed!\r\n");
 }

 drv_aes_crypt(test_aes_plain_text_2, cbuf);
 if (memcmp(test_aes_cipher_text_2_ecb_256, cbuf, 16) == 0)
 {
    cli_printf(">> aes_ecb_encrypt keybit_256 data-2 pass!\r\n\r\n");
 }
 else
 {
    cli_printf(">> aes_ecb_encrypt keybit_256 data-2 failed!\r\n\r\n");
 }

            break;

        default:
```

```
                    break;
}
  }
```

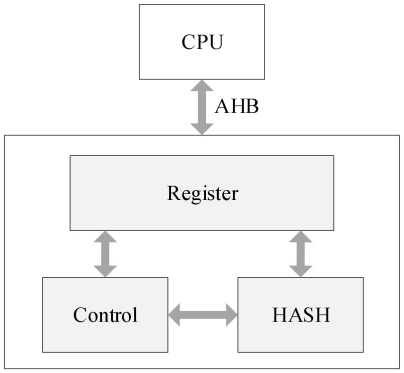
8 HASH

8.1 HASH 简介

HASH 算法是把任意长度的输入，通过散列算法，变换成固定长度的输出，是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。

HASH 可用于许多操作，包括身份验证和数字签名。主要用于对整个消息的完整性进行校验。对于任意长度的消息，SHA-256 都会产生一个 256bit 长的哈希值，SHA2-512 产生一个 512bit 长的哈希值，称作消息摘要。一般来说，为了加快计算速度，签名或者验签都是对摘要进行，而不是对原始数据进行。

HASH 算法框图如图所示，通过 AHB 总线将所要运算的数据写入对应的地址空间，再配置寄存器触发 HASH 进行运算。CPU 可以通过轮询或者是中断的方式来查询运算的工作情况。



HASH模块框图

在 HASH 算法中，必须把原始消息转换成位字符串，同时消息需要进行补位，以使其长度在对 512 取模以后的余数是 448，即补位后的消息长度 mod 512 为 448。

即使长度已经满足对 512 取模后余数是 448，补位也要进行。补位是这样进行的：先补一个 1，然后再补 0，直到长度满足对 512 取模后余数是 448。总而言之，补位是至少补 1 位，最多补 512 位。当硬件自动补位时，仅支持消息为字的整数倍。

8.2 HASH 使用方法

8.2.1 初始化 HASH

初始化 HASH 使用中要用到的系统资源。如果在 menuconfig 时配置了 HASH，那么系统启动时会自动调用 HASH 初始化，所以使用 HASH 前无需额外调用该接口。

函数原型：

```
int drv_hash_init(void)
```

参数说明：

输入参数	说明
无	
返回	——
0	执行成功
其它	执行失败

8.2.2 计算 SHA256 值

函数原型：

```
int drv_hash_sha256_ret(const unsigned char *input, unsigned int ilen, unsigned char output[32])
```

参数说明：

输入参数	说明
intput	存放输入数据的指针，采用 unsigned char 的指针类型
ilen	输入数据的长度
output[32]	存放输出数据的数组
返回	——

0	执行成功
其它	执行失败

8.2.3 计算 SHA512 值

函数原型：

```
int drv_hash_sha512_ret(const unsigned char *input, unsigned int ilen, unsigned char output[64])
```

参数说明：

输入参数	说明
intput	存放输入数据的指针，采用 unsigned char 的指针类型
ilen	输入数据的长度
output[64]	存放输出数据的数组
返回	——
0	执行成功
其它	执行失败

8.3 HASH 使用示例

示例代码的主要步骤如下所示：

计算 SHA256 值/计算 SHA512 值

参考代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "cli.h"
#include "oshal.h"
#include "chip_pinmux.h"
#include "hash.h"

#define HASH_256_SUM_LEN        32
#define HASH_512_SUM_LEN        64
```

```
static const unsigned char test_hash_data_1[] = "12345678";
static const unsigned char test_hash_data_2[] = "abcdefghijklmnopqrstuvwxyz00";
static const unsigned char test_hash_data_3[] = "12345678abcdefghijklmnopqrstuvwxyz"
        "12345678abcdefghijklmnopqrstuvwxyz12345678abcdefghijklmnopqrstuvwxyz1234567
        8abcdefghijklmnopqrstuvwxyz";

static const unsigned char test_sha256_sum_1[] =
{
        0xef, 0x79, 0x7c, 0x81, 0x18, 0xf0, 0x2d, 0xfb,
        0x64, 0x96, 0x07, 0xdd, 0x5d, 0x3f, 0x8c, 0x76,
        0x23, 0x04, 0x8c, 0x9c, 0x06, 0x3d, 0x53, 0x2c,
        0xc9, 0x5c, 0x5e, 0xd7, 0xa8, 0x98, 0xa6, 0x4f
};

static const unsigned char test_sha256_sum_2[] =
{
        0x5b, 0xb5, 0x32, 0x55, 0x07, 0xaf, 0x99, 0x12,
        0x81, 0x84, 0x84, 0x7b, 0xaf, 0xe0, 0x2a, 0x30,
        0xa1, 0x91, 0xae, 0xda, 0xa8, 0xba, 0xb8, 0x56,
        0x98, 0x07, 0x5e, 0xa6, 0x8e, 0x7b, 0x63, 0x2a
};

static const unsigned char test_sha256_sum_3[] =
{
        0x61, 0xa8, 0x9d, 0x8c, 0x04, 0x3d, 0xc9, 0xd9,
        0x54, 0xfc, 0x86, 0x56, 0xee, 0xe2, 0xbc, 0x42,
        0xd8, 0x5b, 0xb5, 0x31, 0xfe, 0x69, 0x4e, 0x6c,
        0xbd, 0x48, 0x23, 0xcb, 0x51, 0x05, 0x71, 0x31
};

static const unsigned char test_sha512_sum_1[] =
```

```
{
        0xfa, 0x58, 0x5d, 0x89, 0xc8, 0x51, 0xdd, 0x33,
        0x8a, 0x70, 0xdc, 0xf5, 0x35, 0xaa, 0x2a, 0x92,
        0xfe, 0xe7, 0x83, 0x6d, 0xd6, 0xaf, 0xf1, 0x22,
        0x65, 0x83, 0xe8, 0x8e, 0x09, 0x96, 0x29, 0x3f,
        0x16, 0xbc, 0x00, 0x9c, 0x65, 0x28, 0x26, 0xe0,
        0xfc, 0x5c, 0x70, 0x66, 0x95, 0xa0, 0x3c, 0xdd,
        0xce, 0x37, 0x2f, 0x13, 0x9e, 0xff, 0x4d, 0x13,
        0x95, 0x9d, 0xa6, 0xf1, 0xf5, 0xd3, 0xea, 0xbe
};

static const unsigned char test_sha512_sum_2[] =
{
        0xd2, 0xa4, 0xe3, 0x4a, 0x9b, 0x3c, 0xe2, 0xd0,
        0xa4, 0x0b, 0x0f, 0x51, 0x7d, 0x26, 0xcc, 0x82,
        0x7e, 0x91, 0x6c, 0x40, 0x45, 0x42, 0x21, 0xaa,
        0xaa, 0xfe, 0x55, 0x63, 0x49, 0x15, 0x5a, 0x6b,
        0xe3, 0x54, 0x02, 0xf9, 0xae, 0x9f, 0xc5, 0x94,
        0xf6, 0x39, 0xd3, 0x8c, 0xc2, 0x95, 0x22, 0xe5,
        0x7e, 0xb4, 0xde, 0x0c, 0xe9, 0xf2, 0x50, 0xc3,
        0x28, 0x31, 0xf4, 0x15, 0x16, 0xea, 0xf5, 0xce
};

static const unsigned char test_sha512_sum_3[] =
{
        0x78, 0x75, 0xbb, 0x5a, 0x55, 0x23, 0xe8, 0xdc,
        0x4c, 0xa1, 0xe9, 0x5b, 0x90, 0x5a, 0x7a, 0x72,
        0x3c, 0x20, 0x39, 0x40, 0x7e, 0xcc, 0x85, 0x00,
        0xa0, 0xda, 0xc3, 0xe3, 0x3e, 0xe7, 0x9e, 0x93,
        0x98, 0xde, 0x10, 0x9b, 0x1c, 0xbe, 0x6e, 0xb0,
        0x77, 0xfc, 0xbf, 0x84, 0xaa, 0x8b, 0xd7, 0xe7,
        0x03, 0x1e, 0x1b, 0x74, 0xd9, 0x97, 0x57, 0xd0,
```

```
        0xcd, 0xf2, 0x79, 0x46, 0xb4, 0x37, 0x6f, 0xa3
};



static int sha256_test(cmd_tbl_t *t, int argc, char *argv[])
{
unsigned int flag = strtoul(argv[1], NULL, 0);
        unsigned char output[HASH_512_SUM_LEN];
        unsigned int data_len1, data_len2, data_len3;

        switch (flag)
        {
                case 1:
                data_len1 = sizeof(test_hash_data_1) - 1;
                drv_hash_sha256_ret(test_hash_data_1, data_len1, output);

         if (memcmp(test_sha256_sum_1, output, HASH_256_SUM_LEN) == 0)
         {
          cli_printf(">> sha256_test data-1 pass!\r\n");
          }
         else
         {
                cli_printf(">> sha256_test data-1 failed!\r\n");
         }
                break;

         case 2:
                data_len2 = sizeof(test_hash_data_2) - 1;
                drv_hash_sha256_ret(test_hash_data_2, data_len2, output);

                if (memcmp(test_sha256_sum_2, output, HASH_256_SUM_LEN) == 0)
                {
```

```
                cli_printf(">> sha256_test data-2 pass!\r\n");
            }
            else
            {
            cli_printf(">> sha256_test data-2 failed!\r\n");
            }
            break;

        case 3:
            data_len3 = sizeof(test_hash_data_3) - 1;
            drv_hash_sha256_ret(test_hash_data_3, data_len3, output);

        if (memcmp(test_sha256_sum_3, output, HASH_256_SUM_LEN) == 0)
            {
                cli_printf(">> sha256_test data-3 pass!\r\n");
            }
            else
            {
                cli_printf(">> sha256_test data-3 failed!\r\n");
             }
            break;

        default:
            break;
        }

       return 0;
}
CLI_CMD(sha256, sha256_test, "test_data_num", "test data and size");
```

9 RNG

9.1 RNG 简介

随机数是密码学算法的基础，是现代加密体系中最重要的部分之一。几乎所有的密码学算法都需要使用随机数，因此，高质量的随机数在信息安全系统中的作用举足轻重。

随机数可以通过硬件来生成，也可以通过软件来生成。通过硬件生成的随机数列，是根据传感器收集的热量、声音的变化等事实上无法预测和重现的自然现象信息来生成的。像这样的硬件设备就称为随机数生成器，随机数发生器所产生的数据，后面的数与前面的数毫无关系。

一般来说，随机数发生器有真随机数发生器（TRNG，True Random Number Generator）、伪随机数发生器（PRNG，Pseudo Random Number Generator）。

TRNG 生成的是真正的随机数，但是真正的随机数的来源难以得到。来源有物理的噪声等。

PRNG 是通过某一种确定性算法来生成随机数。

9.2 RNG 使用方法

9.2.1 获取真随机数

函数原型：

```
unsigned int drv_trng_get(void)
```

参数说明：

输入参数	说明
无	
返回	——
unsigned int	随机数发生器产生 32 位的真随机数

9.2.2 获取伪随机数

函数原型：

```
unsigned int drv_prng_get(void)
```

参数说明：

输入参数	说明
无	
返回	——
unsigned int	随机数发生器产生 32 位的伪随机数

2.3 RNG 使用示例

参考代码如下：

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "cli.h"
#include "oshal.h"
#include "chip_pinmux.h"
#include "trng.h"

static int trng_out(cmd_tbl_t *t, int argc, char *argv[])
{
        unsigned int data;
        data = drv_trng_get();
        cli_printf("trng = 0x%x\n", data);

        return CMD_RET_SUCCESS;
}

CLI_CMD(trng_read, trng_out, "trng_out", "trng_read_out");
static int prng_out(cmd_tbl_t *t, int argc, char *argv[])
{
        unsigned int data;
        data = drv_prng_get();
```

```
        cli_printf("prng = 0x%x\n", data);


        return CMD_RET_SUCCESS;
}


CLI_CMD(prng_read, prng_out, "prng_out", "prng_read_out");
```

第四篇 PIT

10 PWM

10.1 PWM 简介

PWM(Pulse Width Modulation，脉冲宽度调制) 是一种对模拟信号电平进行数字编码的方法，通过不同频率的脉冲使用方波的占空比用来对一个具体模拟信号的电平进行编码，使输出端得到一系列幅值相等的脉冲，用这些脉冲来代替所需要波形的设备。

两个 PIT 理论上可以提供最多 8 个 PWM 通路，不过实际上 6600 只把 6 个通路的 PWM 信号引到芯片外面来（PIT0 的 4 路和 PIT1 前两路），所以软件只能最多操作 6 路 PWM。

10.2 PWM 使用方法

10.2.1 打开 PWM 设备

打开 PWM 设备，将 PIT 通道配置成 PWM 模式。

函数原型：

```
int drv_pwm_open(unsigned int chn_pwm)
```

参数说明：

输入参数	说明
chn_pwm	指定打开 PWM 的通路值。
返回	——
0	打开成功
-1	打开

chn_pwm 参数可取值：

```
/** PWM CHANNEL NUMBER */
#define DRV_PWM_CHN0                DRV_PIT_CHN_0
```

```
#define DRV_PWM_CHN1        DRV_PIT_CHN_1
#define DRV_PWM_CHN2        DRV_PIT_CHN_2
#define DRV_PWM_CHN3        DRV_PIT_CHN_3
#define DRV_PWM_CHN4        DRV_PIT_CHN_4
#define DRV_PWM_CHN5        DRV_PIT_CHN_5
```

10.2.2 配置 PWM 通道值、频率、占空比

配置 PWM 通道值、频率、占空比，得到理想的 PWM 波形。

函数原型：

```
int drv_pwm_config(unsigned int chn_pwm, unsigned int freq, unsigned int duty_ratio)
```

参数说明：

输入参数	说明
chn_pwm	指定配置 PWM 的通路值
freq	期望 PWM 的频率
duty_ratio	PWM 占空比
返回	——
0	配置成功
-1	配置失败

chn_pwm 参数可取值：

```
/** PWM CHANNEL NUMBER */
#define DRV_PWM_CHN0        DRV_PIT_CHN_0
#define DRV_PWM_CHN1        DRV_PIT_CHN_1
#define DRV_PWM_CHN2        DRV_PIT_CHN_2
#define DRV_PWM_CHN3        DRV_PIT_CHN_3
#define DRV_PWM_CHN4        DRV_PIT_CHN_4
#define DRV_PWM_CHN5        DRV_PIT_CHN_5
```

freq 参数可取值：

可取值为 1~24000000Hz

duty_ratio 参数可取值：

可取值为 0~1000

10.2.3 启动 PWM 设备

启动相应的 PWM 通道。

函数原型：

```
int drv_pwm_start(unsigned int chn_pwm)
```

参数说明：

输入参数	说明
chn_pwm	指定启动 PWM 的通路值
返回	——
0	启动成功
-1	启动失败

chn_pwm 参数可取值：

```
/** PWM CHANNEL NUMBER */
#define DRV_PWM_CHN0        DRV_PIT_CHN_0
#define DRV_PWM_CHN1        DRV_PIT_CHN_1
#define DRV_PWM_CHN2        DRV_PIT_CHN_2
#define DRV_PWM_CHN3        DRV_PIT_CHN_3
#define DRV_PWM_CHN4        DRV_PIT_CHN_4
#define DRV_PWM_CHN5        DRV_PIT_CHN_5
```

10.2.4 获取 PWM 通路工作状态

函数原型：

```
unsigned int drv_pwm_get_status(void)
```

参数说明：

输入参数	说明

无	
返回	——
0	当前 PWM 通路未在工作状态
1	当前 PWM 通路在工作状态

10.2.5 关闭 PWM 设备

关闭相应的 PWM 通道。

函数原型：

```
int drv_pwm_stop(unsigned int chn_pwm)
```

参数说明：

输入参数	说明
chn_pwm	指定关闭 PWM 的通路值
返回	——
0	关闭成功
-1	关闭失败

chn_pwm 参数可取值：

```
/** PWM CHANNEL NUMBER */
#define DRV_PWM_CHN0            DRV_PIT_CHN_0
#define DRV_PWM_CHN1            DRV_PIT_CHN_1
#define DRV_PWM_CHN2            DRV_PIT_CHN_2
#define DRV_PWM_CHN3            DRV_PIT_CHN_3
#define DRV_PWM_CHN4            DRV_PIT_CHN_4
#define DRV_PWM_CHN5            DRV_PIT_CHN_5
```

10.3 PWM 使用示例

示例代码的主要步骤如下所示：

1. 打开 PWM 设备，将 PIT 通道 3 配置成 PWM 模式
2. 设置 PWM_CHN3 频率和脉冲宽度默认值

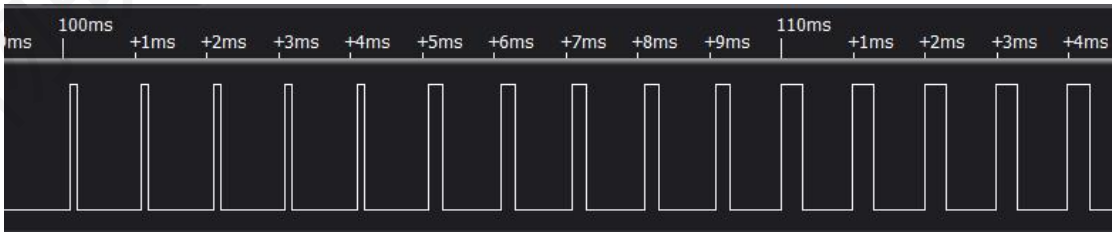
3. 启动 PWM_CHN3
4. 使用 while 循环，每 5ms 修改一次 PWM 脉冲宽度

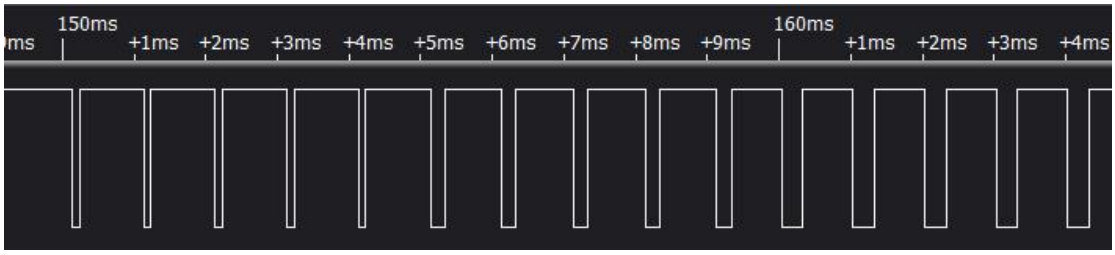
参考代码如下：

```
/*
 * 程序清单：这是一个 PWM 设备使用例程
 * 命令调用格式：pwmtest
 * 程序功能：每 5 毫秒修改一次 PWM 脉冲宽度。
*/
#include "chip_memmap.h"
#include "chip_clk_ctrl.h"
#include "arch_irq.h"
#include "pwm.h"
#include "pit.h"

static int pwm_led_sample(cmd_tbl_t *t, int argc, char *argv[])
{
    unsigned int    period,    pulse, dir;
    period = 1000;          /* 频率为 1000hz, 周期为 1ms      */
    dir = 1;                /* PWM 脉冲宽度值的增减方向 */
    pulse = 0;              /* PWM 占空比 */

    drv_pwm_open(DRV_PWM_CHN3);

/* 设置 PWM 频率和脉冲宽度默认值 */
    drv_pwm_config(DRV_PWM_CHN3, period, pulse);

    /* 使能设备 */
    drv_pwm_get_status();
drv_pwm_start(DRV_PWM_CHN3);

    while (1)
    {
```

```
            drv_pit_delay(120000);/*  延时时间 5ms */
            if (dir)
            {
                pulse += 100;               /*  脉宽从 0 值开始每次增加 0.1ms */
            }
            else
            {
                pulse -= 100;               /*  脉宽从最大值开始每次减少 0.1ms */
            }
            if (pulse >= 1000)
            {
                dir = 0;
            }
            if (0 == pulse)
            {
                dir = 1;
            }
            /*  设置 PWM 周期和脉冲宽度  */
            drv_pwm_config(DRV_PWM_CHN3, period, pulse);
        }
        return 0;
}
CLI_CMD(pwmtest, pwm_led_sample, "pwm led sample", "pwmtest");
```

测试结果：





11 TIMER

11.1 TIMER 简介

TIMER 定时器是 PIT 的另一个功能，TIMER 实际上是计数器，它通过累计已知时间间隔的个数来计算时间。被累计的时间间隔若是系统时钟，计数器就变成了定时器。

可以提供 4 个 TIMER 通路。

11.2 TIMER 使用方法

11.2.1 控制 TIMER 设备

控制的 TIMER 具体事项包括开启、关闭 TIMER，配置 TIMER 的模式，配置 TIMER 的周期、频率，配置 TIMER 的中断回调函数等。

函数原型：

```
int drv_timer_ioctrl(unsigned int chn_timer, int event, void * arg);
```

参数说明：

输入参数	说明
chn_timer	指定打开 TIMER 的通路值。
event	控制事件的类型
arg	控制参数，具体含义根据 event 来定
返回	——
0	执行成功
其它	执行失败

chn_timer 参数可取值：

```
/** PIT TIMER NUMBER */
#define DRV_TIMER0          DRV_PIT_CHN_0
#define DRV_TIMER1          DRV_PIT_CHN_1
#define DRV_TIMER2          DRV_PIT_CHN_2
#define DRV_TIMER3          DRV_PIT_CHN_3
#define DRV_TIMER_MAX       4
```

event 参数可取值

```
p_timer_dev->isr[chn].reload = *((unsigned int *)arg)
```

设置 TIMER 的模式，相应 arg 参数则为指向定时模式的指针

11.2.2 打开 TIMER 设备

打开 TIMER 设备，将 PIT 通道配置成 TIMER 模式。

函数原型：

```
int drv_timer_open(unsigned int chn_timer)
```

参数说明：

输入参数	说明
chn_timer	指定打开 TIMER 的通路值。
返回	——
0	打开成功
其它	打开失败

chn_timer 参数可取值：

```
/** PIT TIMER NUMBER */
#define DRV_TIMER0          DRV_PIT_CHN_0
#define DRV_TIMER1          DRV_PIT_CHN_1
#define DRV_TIMER2          DRV_PIT_CHN_2
#define DRV_TIMER3          DRV_PIT_CHN_3
```

```
#define DRV_TIMER_MAX       4
```

11.2.3 关闭 TIMER 设备

关闭 TIMER 通路，清除、关闭 TIMER 定时器的中断。

函数原型：

```
int drv_timer_close(unsigned int chn_timer)
```

参数说明：

输入参数	说明
chn_timer	指定关闭 TIMER 的通路值。
返回	——
0	关闭成功
其它	关闭失败

chn_timer 参数可取值：

```
/** PIT TIMER NUMBER */
#define DRV_TIMER0          DRV_PIT_CHN_0
#define DRV_TIMER1          DRV_PIT_CHN_1
#define DRV_TIMER2          DRV_PIT_CHN_2
#define DRV_TIMER3          DRV_PIT_CHN_3
#define DRV_TIMER_MAX       4
```

11.3 TIMER 使用示例

示例代码的主要步骤如下所示：

1. 打开 TIMER 通路 0
2. 注册 TIMER 中断回调函数
3. 设置产生 1s 定时中断时间的周期值

4.启动 TIMER 通路 0

参考代码如下：

```
/*
 * 程序清单：这是一个 TIMER 设备使用例程
 * 命令调用格式：timertest
 * 程序功能：产生定时时间为 1 s 的周期性中断。
*/
#include "chip_memmap.h"
#include "chip_clk_ctrl.h"
#include "arch_irq.h"
#include "timer.h"
#include "pit.h"

static os_sem_handle_t timer_process;

int hal_timer_isr(void *data)
{
  os_sem_post(timer_process);
}

void hal_timer_process()
{
   while(1)
{
os_sem_wait(timer_process, 0xFFFFFFFF);
cli_printf("isr_income\r\n");
   }
}
static int timer_led_sample(cmd_tbl_t *t, int argc, char *argv[])
{
      unsigned int mode, counter;
```

```
    T_TIMER_ISR_CALLBACK    m_callback;
    m_callback.timer_callback = (&hal_timer_isr);
m_callback.timer_data = 0;

    drv_timer_open(DRV_PIT_CHN_0);
drv_timer_ioctrl(DRV_PIT_CHN_0,        DRV_TIMER_CTRL_REG_CALLBACK,        (void
        *)&m_callback);
mode = 1;
drv_timer_ioctrl(DRV_PIT_CHN_0, DRV_TIMER_CTRL_SET_MODE, &mode);

counter = 1000000; //单位 us
drv_timer_ioctrl(DRV_PIT_CHN_0, DRV_TIMER_CTRL_SET_PERIOD, &counter);
drv_timer_ioctrl(DRV_PIT_CHN_0, DRV_TIMER_CTRL_START, 0);
    os_task_create( "timer_process",    4, 2048, (main_t)hal_timer_process, NULL);
timer_process = os_sem_create(32, 0);
return 0;
}

CLI_CMD(timertest, timer_led_sample, "timer sample", "timertest");
```

测试结果:

```
FlashID :0x154285[2021-02-05 11:09:57.252]
EasyFlash V4.0.0 is initialize success.[2021-02-05 11:09:57.252]
timertest[2021-02-05 11:10:02.876]
[0]minsys:OK[2021-02-05 11:10:02.876]
isr_income[2021-02-05 11:10:03.876][2021-02-05 11:10:03.876]
isr_income[2021-02-05 11:10:04.876][2021-02-05 11:10:04.876]
isr_income[2021-02-05 11:10:05.876][2021-02-05 11:10:05.876]
isr_income[2021-02-05 11:10:06.875][2021-02-05 11:10:06.875]
isr_income[2021-02-05 11:10:07.875][2021-02-05 11:10:07.875]
isr_income[2021-02-05 11:10:08.875][2021-02-05 11:10:08.875]
isr_income[2021-02-05 11:10:09.875][2021-02-05 11:10:09.875]
isr_income[2021-02-05 11:10:10.874][2021-02-05 11:10:10.874]
isr_income[2021-02-05 11:10:11.874][2021-02-05 11:10:11.874]
isr_income[2021-02-05 11:10:12.889][2021-02-05 11:10:12.889]
isr_income[2021-02-05 11:10:13.874][2021-02-05 11:10:13.874]
isr_income[2021-02-05 11:10:14.889][2021-02-05 11:10:14.889]
```

第五篇 RTC

12 RTC 简介

RTC 是一个低功率实时时钟，它为系统提供精确的时间基准。用于设置系统时钟，提供报警器、唤醒或周期性的定时器。

13 RTC 使用方法

应用程序可以通过下述接口访问 RTC 硬件：

函数	描述
drv_rtc_init()	RTC 设备初始化
drv_rtc_set_time ()	设置 RTC 时间
drv_rtc_get_time ()	读取 RTC 时间
drv_rtc_set_alarm ()	设置 RTC 闹钟
drv_rtc_get_alarm()	读取 RTC 闹钟
drv_rtc_isr_register()	注册 RTC 中断并使能
drv_rtc_isr_unregister()	注销 RTC 中断

13.1 打开 RTC 设备

使用 RTC 前，需要打开 RTC 设备。在打开 RTC 设备时，接口函数进行设备的初始化。

函数原型：

int drv_rtc_init ()

参数说明：

输入参数	说明
无	开启 RTC，启动计时

返回	——
0	设备打开成功
其它	设备打开失败

13.2 读取 RTC 时间

使用 RTC 前，需要打开 RTC 设备。通过以下接口读取 rtc 时间

函数原型：

```
int drv_rtc_get_time(struct rtc_time *time)
```

参数说明：

输入参数	说明
*rtc_time	rtc_time 时间结构体指针
返回	——
0	设备获取成功
其它	设备获取失败

rtc_time 结构体原型

```
struct rtc_time
{
    unsigned int day;
    unsigned int hour;
    unsigned int min;
    unsigned int sec;
    unsigned int cnt_32k;    //note：1(sec)=32768 (cnt_32k)
};
```

13.3 写入 RTC 时间

使用 RTC 前，需要打开 RTC 设备。通过以下接口写入 rtc 时间

函数原型：

int drv_rtc_set_time(struct rtc_time *time)

参数说明：

输入参数	说明
*rtc_time	rtc_time 时间结构体指针，其中 cnt_32k 只读，不可设置；当设置完时间，cnt_32k 会自动清零。
返回	——
0	设备写入成功
其它	设备写入失败

13.4 设置 RTC 闹钟

使用 RTC 前，需要打开 RTC 设备。通过以下接口设置 rtc 闹钟

函数原型：

int drv_rtc_set_alarm(struct rtc_time *time)

参数说明：

输入参数	说明
*rtc_time	rtc_time 时间结构体指针，仅支持配置闹钟的时分秒和 cnt32k
返回	——
0	闹钟设置成功
其它	闹钟设置失败

13.5 读取 RTC 闹钟

使用 RTC 前，需要打开 RTC 设备。通过以下接口读取 rtc 闹钟

函数原型：

int drv_rtc_get_alarm(struct rtc_time *time)

参数说明：

输入参数	说明
*rtc_time	rtc_time 时间结构体指针，仅支持读取闹钟的时分秒和 cnt32k。
返回	——
0	闹钟读取成功
其它	闹钟读取失败

13.6 注册 RTC 中断

使用 RTC 前，需要打开 RTC 设备。通过以下接口注册 rtc 中断

函数原型：

int drv_rtc_isr_register(RTC_TYPEtype,void_fn cb)

参数说明：

输入参数	说明
type	中断类型，详细 RTC 中断类型见 type 枚举定义
cb	中断回调函数
返回	——
0	中断注册成功
其它	中断注册失败

type 枚举定义原型

```
typedef enum _RTC_TYPE{
    DRV_RTC_NORMAL=0,    //正常模式
    DRV_RTC_ALARM,       //闹钟中断
    DRV_RESERVER0,       //保留
    DRV_RTC_DAY=3,       //天中断
    DRV_RTC_HOUR,      //小时中断
    DRV_RTC_MIN,         //分钟中断
    DRV_RTC_SEC,         //秒中断
    DRV_RTC_HSEC,        //保留
    DRV_RTC_ISR_MAX,
```

```
}RTC_TYPE;
```

由于 RTC 支持中断配置，对应常用的中断类型有 day、hour、min、sec 和 alarm 中断，根据实际情况选择对应的模式。

13.7 注销 RTC 中断

使用 RTC 前，需要打开 RTC 设备。通过以下接口注销 rtc 中断

函数原型：

int drv_rtc_isr_unregister(RTC_TYPEtype)

参数说明：

输入参数	说明
type	中断类型，详细中断类型见 2.6 节 type 枚举定义
返回	——
0	中断注销成功
其它	中断注销失败

14 RTC 使用示例

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

#include "cli.h"
#include "oshal.h"
#include "rtc.h"
#include "chip_pinmux.h"
void callback_day()
{
        //ISR callback function,when interrupt generate ,this function will run.
}
void callback_hour()
```

```
{
        //ISR callback function,when interrupt generate ,this function will run.
}
void callback_min()
{
         //ISR callback function,when interrupt generate ,this function will run.
}
void callback_sec()
{
         //ISR callback function,when interrupt generate ,this function will run.
}
void callback_alarm()
{
        //ISR callback function,when interrupt generate ,this function will run.
}


static rtc_test_example(cmd_tbl_t *t, int argc, char *argv[])
{
        struct rtc_time tm = {0};
        /** DAY interrupt mode initialization example*/
        drv_rtc_init();
        //drv_rtc_isr_register(DRV_RTC_DAY,callback_day);        //support multi interrupt
        register
        //drv_rtc_isr_register(DRV_RTC_HOUR,callback_hour);
        //drv_rtc_isr_register(DRV_RTC_MIN,callback_min);
        drv_rtc_isr_register(DRV_RTC_SEC,callback_sec);
        tm.day     = 0;                                /** range 0-32767*/
        tm.hour    = 0;                                /** range 0-24 */
        tm.min     = 0;                                /** range 0-59 */
        tm.sec     = 0;                                /** range 0-59 */
        tm.cnt_32k= 0;                                 /** range 0-32767 ,note: 1sec = 32768cnt*/
        drv_rtc_set_time(&tm);
```

```
        drv_rtc_get_time(&tm);
        cli_printf("day:hour:min:sec:cnt  %d:%d:%d:%d:%d\n",tm.day,tm.hour,tm.min,tm.sec,tm.
        cnt_32k);

        //drv_rtc_isr_unregister(DRV_RTC_ALARM);/**  注销 RTC ALARM  中断*/
        //drv_rtc_isr_unregister(DRV_RTC_DAY);    /**  注销 RTC DAY  中断*/
        //drv_rtc_isr_unregister(DRV_RTC_HOUR); /**  注销 RTC HOUR  中断*/
        //drv_rtc_isr_unregister(DRV_RTC_MIN);    /**  注销 RTC MIN  中断*/
        //drv_rtc_isr_unregister(DRV_RTC_SEC);    /**  注销 RTC SEC  中断*/

        return CMD_RET_SUCCESS;
}
CLI_CMD(rtc, rtc_test_example, "open rtc function", "open rtc");
static rtc_alarm_example(cmd_tbl_t *t, int argc, char *argv[])
{
        struct rtc_time tm = {0};
        /** DAY interrupt mode initialization example*/
        drv_rtc_init();

        /** ALARM interrupt mode initial example
        *       note: alarm only support set hour-min-sec-cnt_32k,not support set day
        */
        drv_rtc_isr_register(DRV_RTC_ALARM,callback_alarm);
        drv_rtc_get_time(&tm);
        cli_printf("[time]
        hour:min:sec:cnt %d:%d:%d:%d\n",tm.hour,tm.min,tm.sec,tm.cnt_32k);
        tm.sec += 5;
        tm.sec = tm.sec%60;

        tm.min    +=tm.sec/60;
        tm.min    +=tm.min%60;
```

```
        tm.hour +=tm.min/60;
        tm.hour +=tm.hour%24;
        /** set alarm at hour:min:sec:snt_32k=0:0:10:0 ;when arrive will goto callback()*/
        drv_rtc_set_alarm(&tm);
        cli_printf("[set                                                    alarm]
        hour:min:sec:cnt %d:%d:%d:%d\n",tm.hour,tm.min,tm.sec,tm.cnt_32k);
        //tm.hour                   /** range 0-24 */
        //tm.min                    /** range 0-59 */
        //tm.sec                    /** range 0-59 */
        //tm.cnt_32k                /** range 0-32767 ,note: 1sec = 32768cnt*/
        drv_rtc_get_alarm(&tm);
        cli_printf("get                                                    alarm
        hour:min:sec:cnt %d:%d:%d:%d\n",tm.hour,tm.min,tm.sec,tm.cnt_32k);
        /** ALARM interrupt mode initial example*/
        //drv_rtc_isr_unregister(DRV_RTC_ALARM);     /**  注销 RTC ALARM  中断*/

        return CMD_RET_SUCCESS;
}

CLI_CMD(rtc_alarm, rtc_alarm_example, "open rtc function,set alarm", "set alarm");
```

第六篇 GPIO

15 GPIO 简介

GPIO，全称 General-PurposeInput/Output（通用输入输出），是一种软件运行期间能够动态配置和控制的通用引脚，也是一颗芯片（MCU）必须具备的最基本外设功能。ECR6600 芯片总共有 25 个可以配置为 GPIO 功能的 pin 脚。所有的 GPIO 在上电后的初始状态都是输入模式，可以通过软件设为上拉或下拉，也可以设置为中断脚。GPIO 通常有三种状态：高电平、低电平和高阻态。高阻态就是断开状态或浮空态。因此上拉和下拉的一个作用就是为了防止输入端悬空，使其有确定的状态。减弱外部电流对芯片的产生的干扰。

每个 GPIO 口除了通用输入输出功能以外，还可能有其它复用功能，具体情况可以参考 ECR6600 相关资料 TR6600_def_pinlist_ballmap.xls 文件进行配置。

16 GPIO 使用方法

应用程序可以通过下述接口访问 GPIO 接口：

函数	描述
drv_gpio_init ()	GPIO 初始化
drv_gpio_write ()	GPIO 输出电平值
drv_gpio_read ()	读 GPIO 当前电平值
drv_gpio_ioctrl ()	控制 GPIO 接口

16.1 初始化 GPIO 接口

函数原型：

```
int drv_gpio_init(void)
```

参数说明：

输入参数	说明

void	无
返回	——
0	GPIO 初始化成功
-1	GPIO 初始化失败

GPIO 初始化函数中主要完成中断服务程序的注册，以及打开中断的功能。并置初始化成功的 flag 为 1。

接口使用示例

在 chip_startup 函数中进行初始化操作即可。

```
#if  defined(CONFIG_GPIO)
      drv_gpio_init();
#endif  //CONFIG_GPIO
```

16.2 GPIO 输出电平值

函数原型：

```
int  drv_gpio_write(int  gpio_num,  GPIO_LEVEL_VALUE  gpio_level)
```

参数说明：

输入参数	说明
gpio_num	要输出电平值(gpio_level)的 GPIO
gpio_level	要输出 GPIO(gpio_num)的电平值
返回	——
0	操作成功
-2	入参错误
-3	初始化错误

gpio_num 的取值

```
/* gpio 编号取值 */
/*  GPIO NUM */
#define GPIO_NUM_0          0x00
#define GPIO_NUM_1          0x01
#define GPIO_NUM_2          0x02
#define GPIO_NUM_3          0x03
#define GPIO_NUM_4          0x04
#define GPIO_NUM_5          0x05
#define GPIO_NUM_6          0x06
#define GPIO_NUM_7          0x07
#define GPIO_NUM_8          0x08
#define GPIO_NUM_9          0x09
#define GPIO_NUM_10             0x0A
#define GPIO_NUM_11             0x0B
#define GPIO_NUM_12             0x0C
#define GPIO_NUM_13             0x0D
#define GPIO_NUM_14             0x0E
#define GPIO_NUM_15             0x0F
#define GPIO_NUM_16             0x10
#define GPIO_NUM_17             0x11
#define GPIO_NUM_18             0x12
#define GPIO_NUM_19             0x13
#define GPIO_NUM_20             0x14
#define GPIO_NUM_21             0x15
#define GPIO_NUM_22             0x16
#define GPIO_NUM_23             0x17
#define GPIO_NUM_24             0x18
#define GPIO_NUM_25             0x19
```

gpio_ level 的取值

```
/* gpio 电平取值 */
typedef enum
{
    LEVEL_LOW   = 0x00,
    LEVEL_HIGH      = 0x01,
} GPIO_LEVEL_VALUE;
```

接口使用示例

```
static int cmd_GPIO_write(cmd_tbl_t *t, int argc, char *argv[])
{
    unsigned int ret = 0;
    unsigned int value = 0;
    unsigned int gpio_num   = strtoul(argv[1], NULL, 0);

    if (strcmp(argv[2], "high") == 0){
        value = 1;
    }
    else if (strcmp(argv[2], "low") == 0){
        value = 0;
    }
    //must set first
    if(set_gpio_function(gpio_num)){
        return CMD_RET_FAILURE;}
    ret = drv_gpio_write(gpio_num,value);
    if(ret != 0){
        return CMD_RET_FAILURE;}
```

```
    drv_gpio_ioctrl(gpio_num,DRV_GPIO_CTRL_SET_DIRCTION,DRV_GPIO_ARG_DIR_OUT);

    return CMD_RET_SUCCESS;
}

CLI_CMD(gpio_write, cmd_GPIO_write, "GPIO write ", "gpio_write <num> <'high'/'low'>");
```

！注意事项
使对应的 GPIO 输出电平值之前，需要先将 ECR6600 对应的 pin 脚复用为 GPIO 功能，其次向对应 GPIO 写入要输出的电平值，最后设置对应 GPIO 的方向为输出方向，设置完成后方可成功设置 gpio_num 输出电平为 gpio_level。

16.3 读 GPIO 当前电平值

函数原型：

```
int drv_gpio_read(int gpio_num)
```

参数说明：

输入参数	说明
gpio_num	要读取电平值的 GPIO
返回	——
val	读取到的 GPIO(gpio_num)的电平值，0：high level 1：low level
-2	入参错误
-3	初始化错误

gpio_num 的取值

```
/* gpio 编号取值 */
/*   GPIO NUM */
#define GPIO_NUM_0          0x00
#define GPIO_NUM_1          0x01
#define GPIO_NUM_2          0x02
#define GPIO_NUM_3          0x03
#define GPIO_NUM_4          0x04
#define GPIO_NUM_5          0x05
#define GPIO_NUM_6          0x06
#define GPIO_NUM_7          0x07
#define GPIO_NUM_8          0x08
#define GPIO_NUM_9          0x09
#define GPIO_NUM_10             0x0A
#define GPIO_NUM_11             0x0B
#define GPIO_NUM_12             0x0C
#define GPIO_NUM_13             0x0D
#define GPIO_NUM_14             0x0E
#define GPIO_NUM_15             0x0F
#define GPIO_NUM_16             0x10
#define GPIO_NUM_17             0x11
#define GPIO_NUM_18             0x12
#define GPIO_NUM_19             0x13
#define GPIO_NUM_20             0x14
#define GPIO_NUM_21             0x15
#define GPIO_NUM_22             0x16
#define GPIO_NUM_23             0x17
#define GPIO_NUM_24             0x18
#define GPIO_NUM_25             0x19
```

接口使用示例

```
static int cmd_GPIO_read(cmd_tbl_t *t, int argc, char *argv[])
{
    unsigned int ret = 0;
    unsigned int gpio_num  = strtoul(argv[1], NULL, 0);


    //must set first
    if(set_gpio_function(gpio_num)){
        return CMD_RET_FAILURE;}

    ret = drv_gpio_read(gpio_num);
    cli_printf("gpio[%d] value = %x\n",gpio_num,ret);
    if(ret != 0){
        return CMD_RET_FAILURE;}

    return CMD_RET_SUCCESS;
}

CLI_CMD(gpio_read, cmd_GPIO_read,  "GPIO read ",  "gpio_read <num>");
```

！注意事项
读不同 GPIO 当前电平值之前，需要先将 ECR6600 对应的 pin 脚复用为 GPIO 功能，其次读取对应 GPIO 当前电平值即可。

16.4 控制 GPIO 接口

函数原型：

int drv_gpio_ioctrl(int gpio_num, int event, int arg)

参数说明：

输入参数	说明
gpio_num	要控制的 GPIO

event	GPIO 控件事件类型
arg	控制参数，具体含义根据事件而定
返回	——
0	操作成功
-2	入参错误
-3	初始化错误

gpio_num 的取值

```
/* gpio 编号取值 */
/*  GPIO NUM */
#define GPIO_NUM_0          0x00
#define GPIO_NUM_1          0x01
#define GPIO_NUM_2          0x02
#define GPIO_NUM_3          0x03
#define GPIO_NUM_4          0x04
#define GPIO_NUM_5          0x05
#define GPIO_NUM_6          0x06
#define GPIO_NUM_7          0x07
#define GPIO_NUM_8          0x08
#define GPIO_NUM_9          0x09
#define GPIO_NUM_10             0x0A
#define GPIO_NUM_11             0x0B
#define GPIO_NUM_12             0x0C
#define GPIO_NUM_13             0x0D
#define GPIO_NUM_14             0x0E
#define GPIO_NUM_15             0x0F
#define GPIO_NUM_16             0x10
#define GPIO_NUM_17             0x11
#define GPIO_NUM_18             0x12
```

```
#define  GPIO_NUM_19          0x13
#define  GPIO_NUM_20          0x14
#define  GPIO_NUM_21          0x15
#define  GPIO_NUM_22          0x16
#define  GPIO_NUM_23          0x17
#define  GPIO_NUM_24          0x18
#define  GPIO_NUM_25          0x19
```

event 的取值

```
/*  event 取值  */
#define  DRV_GPIO_CTRL_REGISTER_ISR        0
#define  DRV_GPIO_CTRL_INTR_MODE           1
#define  DRV_GPIO_CTRL_INTR_ENABLE         2
#define  DRV_GPIO_CTRL_INTR_DISABLE        3
#define  DRV_GPIO_CTRL_SET_DIRCTION        4
#define  DRV_GPIO_CTRL_PULL_TYPE           5
#define  DRV_GPIO_CTRL_SET_DEBOUNCE        6
#define  DRV_GPIO_CTRL_CLOSE_DEBOUNCE      7
```

arg 的取值

```
/*  arg 取值  */
DRV_GPIO_CTRL_REGISTER_ISR  模式 arg 的取值：
（T_GPIO_ISR_CALLBACK  *）结构体指针类型的参数


DRV_GPIO_CTRL_INTR_MODE 模式 arg 的取值：
#define  DRV_GPIO_ARG_INTR_MODE_NOP          0
#define  DRV_GPIO_ARG_INTR_MODE_HIGH         2
#define  DRV_GPIO_ARG_INTR_MODE_LOW              3
#define  DRV_GPIO_ARG_INTR_MODE_N_EDGE       5
```

```
#define  DRV_GPIO_ARG_INTR_MODE_P_EDGE          6
#define  DRV_GPIO_ARG_INTR_MODE_DUAL_EDGE       7

DRV_GPIO_CTRL_SET_DIRCTION 模式 arg 的取值：
#define  DRV_GPIO_ARG_DIR_IN                    0
#define  DRV_GPIO_ARG_DIR_OUT                   1

DRV_GPIO_CTRL_PULL_TYPE 模式 arg 的取值：
#define  DRV_GPIO_ARG_PULL_UP                   0
#define  DRV_GPIO_ARG_PULL_DOWN                 1

DRV_GPIO_CTRL_SET_DEBOUNCE 模式 arg 的取值：
Bounce_time:  ns  gpio 的防抖时间
```

17 GPIO 使用示例

17.1 GPIO 对下降沿的响应

示例代码的主要步骤如下所示：

1. 先进行 GPIO 的初始化操作；
2. 进行 pin 脚复用，将 GPIO20 所对应的 pin 脚设置为 GPIO 功能；
3. 设置 GPIO20 中断的模式为下降沿触发模式；
4. 设置中断服务程序中调用的 callback 函数；
5. 打开中断使能。

参考代码如下：

```
void  hal_ir_init(void)
{
        unsigned  int  argv  =  0;
        unsigned  int  data  =  0;
        T_GPIO_ISR_CALLBACK  p_callback;
```

```
    p_callback.gpio_callback =(&hal_ir_callback);
    p_callback.gpio_data = &data;

    drv_gpio_init();
    PIN_FUNC_SET(IO_MUX_GPIO20, FUNC_GPIO20_GPIO20);
    drv_gpio_ioctrl(GPIO_NUM_20,DRV_GPIO_CTRL_INTR_MODE,DRV_GPIO_ARG_
    INTR_MODE_N_EDGE);     //interrupt by negative edge
    drv_gpio_ioctrl(GPIO_NUM_20,DRV_GPIO_CTRL_REGISTER_ISR,(int)&p_callbac
    k);
    drv_gpio_ioctrl(GPIO_NUM_20,DRV_GPIO_CTRL_INTR_ENABLE,argv);
    os_task_create( "ir_data_process",  4,2048, (main_t)hal_ir_data_process, NULL);
    ir_data_process = os_sem_create(32, 0);
}
```

！注意事项
由于 GPIO 初始化默认是输入方向，所以使用时未做输入输出方向的设置。

第七篇 EFUSE

18 EFUSE 简介

EFUSE，是一次性可编程存储器，在芯片出场之前会被写入信息，可用于存储 MEM repair 的存储修复数据，也可用于存储芯片的信息：如芯片可使用电源电压，芯片的版本号，生产日期。在生产好 die 后，会进行 ATE 测试，将芯片的信息写到 efuse 中去。

Efuse_controller 模块通过 APB 接口交互， 完成 Efuse IP 的读写。 该模块主要特征如下：

支持 32 比特、 和单比特的 Program(PGM)操作；

32-bit 读操作；

支持时序可配置。

19 EFUSE 使用方法

应用程序可以通过下述接口访问 EFUSE 接口：

函数	描述
drv_efuse_init ()	EFUSE 初始化
drv_efuse_write ()	写 EFUSE 接口
drv_efuse_read ()	读 EFUSE 接口

19.1 初始化 EFUSE 接口

函数原型：

```
void drv_efuse_init(void)
```

参数说明：

输入参数	说明

void	无
返回	——
void	无

EFUSE 初始化函数中主要完成打开 EFUSE 模块时钟以及根据不同的 APB 频率选择对应的初始化函数进行时序配置操作。

接口使用示例

在 chip_startup 函数中进行初始化操作即可。

```
#if  defined(CONFIG_EFUSE)
      drv_efuse_init();
#endif  //CONFIG_EFUSE
```

19.2 写 EFUSE 接口

函数原型：

int drv_efuse_write(unsigned int addr, unsigned int value, unsigned int mask)

参数说明：

输入参数	说明
addr	要将数据写入 efuse 的地址，范围：0x00-0x7c。
value	要写入 efuse 指定地址的数据
mask	控制写入 32 位数据中哪些 bit 可以写入。以 bit 为单位 0：可以写入，1：不可写入
返回	——
0	操作成功
-2	参数错误

通过 APB 一次写 32-bit 或者通过 0x24 地址 mask 部分 bit，向地址 0x200 至 0x27C 地址直接写入待烧写的数据即可。需要注意的是函数中默认写入地址从 0x200 开始，所以 addr 入参的取值范围为 0x00-0x7c。

接口使用示例

直接调用写 efuse 接口，并传入入参即可，addr 入参需要注意的是，由于 APB 一次写 32-bit，所以实际操作以 4 字节对齐的地址开始写操作。

```
static int cmd_efuse_write(cmd_tbl_t *t, int argc, char *argv[])
{
    unsigned int addr  = strtoul(argv[1], NULL, 0);
    unsigned int value = strtoul(argv[2], NULL, 0);
    unsigned int mask  = strtoul(argv[3], NULL, 0);
    unsigned int ret;

    ret = drv_efuse_write(addr, value, mask);
    if(ret != 0){
        return CMD_RET_FAILURE;}

    return CMD_RET_SUCCESS;
}

CLI_CMD(efuse_write, cmd_efuse_write, "efuse write ", "efuse_write <addr> <value> <mask>");
```

19.3 读 EFUSE 接口

函数原型：

void drv_efuse_read(unsigned int addr,unsigned int * value, unsigned int length)

参数说明：

输入参数	说明
addr	要在 efuse 中读取数据的地址。范围：0x00-0x7c，四字节对齐。
* value	存放从 efuse 内存中的指定地址读取指定长度数据的指针
length	从 efuse 内存中指定地址读取的指定长度。
返回	——

0	操作成功
-2	参数错误

可读取 EFUSE 数据的内存总共有两块，分别是 0x100-0x17c 以及 0x300-0x37c。两块内存存储信息基本一致。不同之处在于，当向 0x200 地址写入数据后，0x300 地址及时同步到写入的信息，可以立刻进行读取操作。0x100 地址需要 reload 操作之后才同步写入的信息，并使配置生效。而且 0x100-0x120 地址的数据不对外显示。0x300-0x37c 内存全部可以显示。

接口使用示例

直接调用读 efuse 接口，并传入入参即可，addr 入参需要注意的是，由于 APB 一次读 32-bit，所以实际操作以 4 字节对齐的地址开始读操作。

```
static int cmd_efuse_read(cmd_tbl_t *t, int argc, char *argv[])
{
    unsigned int addr = strtoul(argv[1], NULL, 0);
    unsigned int length = strtoul(argv[2], NULL, 0);
    unsigned char value[100] = {0};
    int ret,i = 0;

    ret = drv_efuse_read(addr, value, length);

    if(ret != 0){
        return CMD_RET_FAILURE;}

    for(i=0;i<length;i++)
    {
        cli_printf("data[%d] = %x\n",i,value[i]);
    }
    return CMD_RET_SUCCESS;
}

CLI_CMD(efuse_read, cmd_efuse_read, "efuse read ", "efuse_read <addr> <length>");
```