文档版本	v1.0
发布日期	2020-08-10

HAL API

芯片开放社区
Open Chip Community

目录

相关结数据结构
timer_dev_t
timer_config_t
hal_timer_cb_t
使用示例
移植说明
注意事项

SPI

接口列表

接口详情

int32_t hal_spi_init(spi_dev_t ***spi)**

int32_t hal_spi_send(spi_dev_t ***spi, const uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_spi_recv(spi_dev_t ***spi, uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_spi_send_recv(spi_dev_t ***spi, uint8_t* tx_data, uint8_t *rx_data, uint16_t size, uint32_t timeout)**

int32_t hal_spi_finalize(spi_dev_t ***spi)**

相关宏定义

相关结数据结构

spi_dev_t

spi_config_t

使用示例

移植说明

注意事项

I2C

接口列表

接口详情

int32_t hal_i2c_init(i2c_dev_t ***i2c)**

int32_t hal_i2c_master_send(i2c_dev_t ***i2c, uint16_t dev_addr, const uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_master_recv(i2c_dev_t ***i2c, uint16_t dev_addr, uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_slave_send(i2c_dev_t ***i2c, const uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_slave_recv(i2c_dev_t ***i2c, uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_mem_write(i2c_dev_t ***i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, const uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_mem_read(i2c_dev_t ***i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, uint8_t* data, uint16_t size, uint32_t timeout)**

int32_t hal_i2c_finalize(i2c_dev_t ***i2c)**

相关宏定义

相关结数据结构

i2c_dev_t

i2c_config_t

使用示例

移植说明

ADC

接口列表

接口详情

int32_t hal_adc_init(adc_dev_t *adc)

int32_t hal_adc_value_get(adc_dev_t *adc, void* output, uint32_t timeout)

int32_t hal_adc_finalize(adc_dev_t *adc)

相关结数据结构

adc_dev_t

adc_config_t

使用示例

移植说明

注意事项

RTC

接口列表

使用说明

管脚映射

Hal使用前需要drv_pinmux_config()函数配置管脚功能。

如drv_pinmux_config(GPIO_P14,ADCC)配置14管脚为ADC功能。

GPIO

接口列表

函数名称	功能描述
hal_gpio_init	初始化指定GPIO管脚
hal_gpio_output_high	使指定GPIO输出高电平
hal_gpio_output_low	使指定GPIO输出低电平
hal_gpio_output_toggle	使指定GPIO输出翻转
hal_gpio_input_get	获取指定GPIO管脚的输入值
hal_gpio_enable_irq	使能指定GPIO的中断模式，挂载中断服务函数
hal_gpio_disable_irq	关闭指定GPIO的中断
hal_gpio_clear_irq	清除指定GPIO的中断状态
hal_gpio_finalize	关闭指定GPIO

接口详情

int32_t hal_gpio_init(gpio_dev_t *gpio)

描述	初始化指定GPIO管脚
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示GPIO初始化成功，非0表示失败

int32_t hal_gpio_output_high(gpio_dev_t *gpio)

描述	使指定GPIO输出高电平
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示GPIO输出高电平成功，非0表示失败

int32_t hal_gpio_output_low(gpio_dev_t *gpio)

描述	使指定GPIO输出低电平
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示GPIO输出低电平成功，非0表示失败

int32_t hal_gpio_output_toggle(gpio_dev_t *gpio)

描述	使指定GPIO输出翻转
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

int32_t hal_gpio_input_get(gpio_dev_t *gpio, uint32_t* value)

描述	获取指定GPIO管脚的输入值
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
	value：存储输入值的数据指针
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

int32_t hal_gpio_enable_irq(gpio_dev_t *gpio, gpio_irq_trigger_t trigger, gpio_irq_handler_t handler, void *arg)

描述	使能指定GPIO的中断模式，挂载中断服务函数
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
	trigger：中断的触发模式，上升沿、下降沿还是都触发
	gpio_irq_handler_t：中断服务函数指针，中断触发后将执行指向的函数
	arg：中断服务函数的入参
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

int32_t hal_gpio_disable_irq(gpio_dev_t *gpio)

描述	关闭指定GPIO的中断
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

int32_t hal_gpio_clear_irq(gpio_dev_t *gpio)

描述	清除指定GPIO的中断状态
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

int32_t hal_gpio_finalize(gpio_dev_t *gpio)

描述	关闭指定GPIO
参数	gpio：GPIO设备描述，定义需要初始化的GPIO管脚的相关特性
返回值	类型：int 返回成功或失败, 返回0表示timer创建成功，非0表示失败。

相关结数据结构

gpio_dev_t

```
typedef struct {
    uint8_t       port;    /* gpio逻辑端口号 */
    gpio_config_t config;  /* gpio配置信息 */
    void         *priv;    /* 私有数据 */
} gpio_dev_t;
```

gpio_config_t

```
typedef enum {
    ANALOG_MODE,                /* 管脚用作功能引脚，如用于pwm输出，uart的输入引脚 */
    IRQ_MODE,                   /* 中断模式，配置为中断源 */
    INPUT_PULL_UP,              /* 输入模式，内部包含一个上拉电阻 */
    INPUT_PULL_DOWN,            /* 输入模式，内部包含一个下拉电阻 */
    INPUT_HIGH_IMPEDANCE,       /* 输入模式，内部为高阻模式 */
    OUTPUT_PUSH_PULL,           /* 输出模式，普通模式 */
    OUTPUT_OPEN_DRAIN_NO_PULL, /* 输出模式，输出高电平时，内部为高阻状态 */
    OUTPUT_OPEN_DRAIN_PULL_UP, /* 输出模式，输出高电平时，被内部电阻拉高 */
} gpio_config_t;
```

gpio_irq_trigger_t

```
typedef enum {
    IRQ_TRIGGER_RISING_EDGE  = 0x1, /* 上升沿触发 */
    IRQ_TRIGGER_FALLING_EDGE = 0x2, /* 下降沿触发 */
    IRQ_TRIGGER_BOTH_EDGES   = IRQ_TRIGGER_RISING_EDGE |
IRQ_TRIGGER_FALLING_EDGE,                                   /* 上升沿下降沿均触发
*/
} gpio_irq_trigger_t;
```

gpio_irq_handler_t

```
typedef void (*gpio_irq_handler_t)(void *arg);
```

使用示例

GPIO作为输出

```
#include <hal/soc/gpio.h>

#define GPIO_LED_IO 18

/* define dev */
gpio_dev_t led;
```

```
int application_start(int argc, char *argv[])
{
    int ret = -1;

    drv_pinmux_config(GPIO_LED_IO, PIN_FUNC_GPIO);
    /* gpio port config */
    led.port = GPIO_LED_IO;

    /* set as output mode */
    led.config = OUTPUT_PUSH_PULL;

    /* configure GPIO with the given settings */
    ret = hal_gpio_init(&led);
    if (ret != 0) {
        printf("gpio init error !\n");
    }

    /* output high */
    hal_gpio_output_high(&led);

    /* output low */
    hal_gpio_output_low(&led);

    /* toggle the LED every 1s */
    while(1) {

        /* toggle output */
        hal_gpio_output_toggle(&led);

        /* sleep 1000ms */
        aos_msleep(1000);
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

GPIO作为中断输入

```
#include <hal/soc/gpio.h>

#define GPIO_BUTTON_IO 5

/* define dev */
gpio_dev_t button1;

/* pressed flag */
int button1_pressed = 0;

void button1_handler(void *arg)
{
    button1_pressed = 1;
}

int application_start(int argc, char *argv[])
```

```
{
    int ret = -1;

    drv_pinmux_config(GPIO_BUTTON_IO, PIN_FUNC_GPIO);

    /* input pin config */
    button1.port = GPIO_BUTTON_IO;

    /* set as interrupt mode */
    button1.config = INPUT_PULL_UP;

    /* configure GPIO with the given settings */
    ret = hal_gpio_init(&button1);
    if (ret != 0) {
        printf("gpio init error !\n");
    }

    /* gpio interrupt config */
    ret = hal_gpio_enable_irq(&button1, IRQ_TRIGGER_FALLING_EDGE,
                              button1_handler, NULL);
    if (ret != 0) {
        printf("gpio irq enable error !\n");
    }

    /* if button is pressed, print "button 1 is pressed !" */
    while(1) {
        if (button1_pressed == 1) {
            button1_pressed = 0;
            printf("button 1 is pressed !\n");
        }

        /* sleep 100ms */
        aos_msleep(100);
    };
}
```

当button被按下后，串口会打印"button 1 is pressed !"

移植说明

新建hal_gpio_xxmcu.c和hal_gpio_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_gpio_xxmcu.c中实现所需要的hal函数，hal_gpio_xxmcu.h中放置相关宏定义。

注意事项

(1).GPIO不支持双边沿中断触发。

(2).TG7100B的IO均支持唤醒，但只有部分IO支持中断(P0 ~ P3, P9 ~ P10, P14 ~ P17)。

(3).烧录固件时，P24、P25不能上拉，否则无法进入烧录模式。

(4).IO设置为唤醒功能，并触发唤醒时，无法同时触发中断。可以通过在唤醒处理函数pm_after_sleep_action()里，读取IO电平来判断IO的操作。

示例：

```
#define WAKEUP_IO_PIN P3

int pm_after_sleep_action()
{
    ....
    if (!phy_gpio_read(WAKEUP_IO_PIN)) {
        ....
    }
    ....
}
```

(5).低功耗模式下，IO的配置无效。如果需要保持IO的电平，可以不调用HAL接口，直接通过下列接口设置芯片内部上拉、下拉来保持电平。但需要注意，此时IO的驱动能力较弱，如需驱动LED，可增加三极管驱动。

示例：

```
#define LED_IO_PIN P3

phy_gpio_pull_set(LED_IO_PIN, PULL_DOWN); //设置PIN脚下拉，下拉电阻100K OHM
phy_gpio_pull_set(LED_IO_PIN, STRONG_PULL_UP); //设置PIN脚为强上拉 上拉电阻 10K OHM
phy_gpio_pull_set(LED_IO_PIN, WEAK_PULL_UP); //设置PIN脚为弱上拉, 上拉电阻 150K OHM
```

(6).P0和P33休眠后复用配置信息将不保存，IO配置为输出，其他IO配置为输入，具体参考TG7100B管脚使用说明。P1 ~ P3休眠时可以保持输入上拉状态，P0不保持。

UART

接口列表

函数名称	功能描述
hal_uart_init	初始化指定UART
hal_uart_send	从指定的UART发送数据
hal_uart_recv	从指定的UART接收数据
hal_uart_recv_II	从指定的UART接收数据2
hal_uart_finalize	关闭指定UART

接口详情

int32_t hal_uart_init(uart_dev_t *uart)

描述	初始化指定UART
参数	uart：UART设备描述，定义需要初始化的UART参数
返回值	返回成功或失败, 返回0表示UART初始化成功，非0表示失败

int32_t hal_uart_send(uart_dev_t *uart, const void* data, uint32_t size, uint32_t timeout)

描述	从指定的UART发送数据
参数	uart：UART设备描述，定义需要初始化的UART参数
	data：指向要发送数据的数据指针
	size：要发送的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示UART数据发送成功，非0表示失败

int32_t hal_uart_recv(uart_dev_t *uart, void* data, uint32_t expect_size, uint32_t timeout)

描述	从指定的UART接收数据
参数	uart：UART设备描述，定义需要初始化的UART参数
	data：指向接收缓冲区的数据指针
	expect_size：期望接收的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收expect_size个数据，非0表示失败

int32_t hal_uart_recv_II(uart_dev_t *uart, void* data, uint32_t expect_size, uint32_t *recv_size, uint32_t timeout)

描述	从指定的UART接收数据2
参数	uart：UART设备描述，定义需要初始化的UART参数
	data：指向接收缓冲区的数据指针
	expect_size：期望接收的数据字节数
	recv_size：实际接收数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收expect_size个数据，非0表示失败

int32_t hal_uart_finalize(uart_dev_t *uart)

描述	关闭指定UART
参数	uart：UART设备描述，定义需要初始化的UART参数
返回值	类型：int 返回成功或失败, 返回0表示UART关闭成功，非0表示失败。

相关结数据结构

uart_dev_t

```
typedef struct {
    uint8_t       port;   /* uart port */
    uart_config_t config; /* uart config */
    void         *priv;   /* priv data */
} uart_dev_t;
```

uart_config_t

```
typedef struct {
    uint32_t                baud_rate;
    hal_uart_data_width_t   data_width;
    hal_uart_parity_t       parity;
    hal_uart_stop_bits_t    stop_bits;
    hal_uart_flow_control_t flow_control;
    hal_uart_mode_t         mode;
} uart_config_t;
```

hal_uart_data_width_t

```
typedef enum {
    DATA_WIDTH_5BIT,
    DATA_WIDTH_6BIT,
    DATA_WIDTH_7BIT,
    DATA_WIDTH_8BIT,
    DATA_WIDTH_9BIT
} hal_uart_data_width_t;
```

hal_uart_parity_t

```
typedef enum {
    NO_PARITY,
    ODD_PARITY,
    EVEN_PARITY
} hal_uart_parity_t;
```

hal_uart_stop_bits_t

```
typedef enum {
    STOP_BITS_1,
    STOP_BITS_2
} hal_uart_stop_bits_t;
```

hal_uart_flow_control_t

```
typedef enum {
    FLOW_CONTROL_DISABLED,
    FLOW_CONTROL_CTS,
    FLOW_CONTROL_RTS,
    FLOW_CONTROL_CTS_RTS
} hal_uart_flow_control_t;
```

hal_uart_mode_t

```
typedef enum {
    MODE_TX,
    MODE_RX,
    MODE_TX_RX
} hal_uart_mode_t;
```

使用示例

```c
#include <hal/soc/uart.h>

#define UART1_PORT_NUM  1
#define UART_BUF_SIZE   10
#define UART_TX_TIMEOUT 10
#define UART_RX_TIMEOUT 10

/* define dev */
uart_dev_t uart1;

/* data buffer */
char uart_data_buf[UART_BUF_SIZE];

int application_start(int argc, char *argv[])
{
    int count   = 0;
    int ret     = -1;
    int i       = 0;
    int rx_size = 0;

    /* uart port set */
    uart1.port = UART1_PORT_NUM;

    /* uart attr config */
    uart1.config.baud_rate    = 115200;
    uart1.config.data_width   = DATA_WIDTH_8BIT;
    uart1.config.parity       = NO_PARITY;
    uart1.config.stop_bits    = STOP_BITS_1;
    uart1.config.flow_control = FLOW_CONTROL_DISABLED;
    uart1.config.mode         = MODE_TX_RX;

    /* init uart1 with the given settings */
    ret = hal_uart_init(&uart1);
    if (ret != 0) {
        printf("uart1 init error !\n");
    }

    /* init the tx buffer */
    for (i = 0; i < UART_BUF_SIZE; i++) {
        uart_data_buf[i] = i + 1;
    }

    /* send 0,1,2,3,4,5,6,7,8,9 by uart1 */
    ret = hal_uart_send(&uart1, uart_data_buf, UART_BUF_SIZE, UART_TX_TIMEOUT);
    if (ret == 0) {
        printf("uart1 data send succeed !\n");
    }

    /* scan uart1 every 100ms, if data received send it back */
    while(1) {
        ret = hal_uart_recv_II(&uart1, uart_data_buf, UART_BUF_SIZE,
                               &rx_size, UART_RX_TIMEOUT);
        if ((ret == 0) && (rx_size == UART_BUF_SIZE)) {
            printf("uart1 data received succeed !\n");

            ret = hal_uart_send(&uart1, uart_data_buf, rx_size,
UART_TX_TIMEOUT);
```

```
            if (ret == 0) {
                printf("uart1 data send succeed !\n");
            }
        }

        /* sleep 100ms */
        aos_msleep(100);
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_uart_xxmcu.c和hal_uart_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_uart_xxmcu.c中实现所需要的hal函数，hal_uart_xxmcu.h中放置相关宏定义。

注意事项

(1).TG7101模组的串口定义为P9 (TX) P10(RX)， 不建议更改为其他引脚，以免引起异常问题。如果使用TG7100B芯片，需要在RX脚外接10K上拉电阻，具体可以查看硬件参考设计。

(2).复用其他管脚为UART TX/RX功能时

- 设置RX引脚弱上拉
- P9 P10 复用为GPIO
- 指定引脚复用为UART TX/RX功能

配置其他管脚为UART功能的示例代码参考：

```
board\tg7100b\init\board_init.c
#define UART_RX_PIN P15
#define UART_TX_PIN P1
void hal_rfphy_init(void)
{
    ...
    //========== UART RX Pull up,设置RX引脚上拉
    phy_gpio_pull_set(UART_RX_PIN, WEAK_PULL_UP);
    ...
}

void __attribute__((weak)) board_init(void)
{
    //复用默认管脚为GPIO功能
    drv_pinmux_config(P9, PIN_FUNC_GPIO);
    drv_pinmux_config(P10, PIN_FUNC_GPIO);

    //指定管脚复用为UART功能
    drv_pinmux_config(UART_TX_PIN, UART_TX);
    drv_pinmux_config(UART_RX_PIN, UART_RX);
}
```

- 产测组件的相应修改:

```
platform\mcu\tg7100b\modules\ble_dut\ble_dut_test.c
static void config_uart_pin(void)
{
    drv_pinmux_config(UART_TX_PIN, UART_TX);
    drv_pinmux_config(UART_RX_PIN, UART_RX);
}
```

TIMER

接口列表

函数名称	功能描述
hal_timer_init	初始化指定TIMER
hal_timer_start	启动指定的TIMER
hal_timer_stop	停止指定的TIMER
hal_timer_para_chg	改变指定TIMER的参数
hal_timer_finalize	关闭指定TIMER

接口详情

int32_t hal_timer_init(timer_dev_t *tim)

描述	初始化指定TIMER
参数	tim：TIMER设备描述，定义需要初始化的TIMER参数
返回值	返回成功或失败, 返回0表示TIMER初始化成功，非0表示失败

int32_t hal_timer_start(timer_dev_t *tim)

描述	启动指定的TIMER
参数	tim：TIMER设备描述
返回值	返回成功或失败, 返回0表示TIMER启动成功，非0表示失败

void hal_timer_stop(timer_dev_t *tim)

描述	停止指定的TIMER
参数	tim：TIMER设备描述
返回值	返回成功或失败, 返回0表示TIMER停止成功，非0表示失败

int32_t hal_timer_para_chg(timer_dev_t *tim, timer_config_t para)

描述	改变指定TIMER的参数
参数	tim：TIMER设备描述
	para：TIMER配置信息
返回值	返回成功或失败, 返回0表示TIMER参数改变成功，非0表示失败

int32_t hal_timer_finalize(timer_dev_t *tim)

描述	关闭指定TIMER
参数	tim：TIMER设备描述
返回值	返回成功或失败, 返回0表示TIMER关闭成功，非0表示失败

相关宏定义

```
#define TIMER_RELOAD_AUTO  1 /* timer reload automatic */
#define TIMER_RELOAD_MANU  2 /* timer reload manual */
```

相关结数据结构

timer_dev_t

```
typedef struct {
    int8_t         port;   /* timer port */
    timer_config_t config; /* timer config */
    void           *priv;  /* priv data */
} timer_dev_t;
```

timer_config_t

```
typedef struct {
    uint32_t       period; /* us */
    uint8_t        reload_mode;
    hal_timer_cb_t cb;
    void          *arg;
} timer_config_t;
```

hal_timer_cb_t

```
typedef void (*hal_timer_cb_t)(void *arg);
```

使用示例

```
#include <hal/soc/timer.h>

#define TIMER1_PORT_NUM 1

/* define dev */
timer_dev_t timer1;

void timer_handler(void *arg)
{
    static int timer_cnt = 0;

    printf("timer_handler: %d times !\n", timer_cnt++);
}

int application_start(int argc, char *argv[])
{
    int ret = -1;
    timer_config_t timer_cfg;
    static int count = 0;

    /* timer port set */
    timer1.port = TIMER1_PORT_NUM;

    /* timer attr config */
    timer1.config.period         = 1000000; /* 1s */
    timer1.config.reload_mode    = TIMER_RELOAD_AUTO;
    timer1.config.cb             = timer_handler;

    /* init timer1 with the given settings */
    ret = hal_timer_init(&timer1);
    if (ret != 0) {
        printf("timer1 init error !\n");
```

```
    }

    /* start timer1 */
    ret = hal_timer_start(&timer1);
    if (ret != 0) {
        printf("timer1 start error !\n");
    }

    while(1) {

        /* change the period to 2s */
        if (count == 5) {
            memset(&timer_cfg, 0, sizeof(timer_config_t));
            timer_cfg.period = 2000000;

            ret = hal_timer_para_chg(&timer1, timer_cfg);
            if (ret != 0) {
                printf("timer1 para change error !\n");
            }
        }

        /* stop and finalize timer1 */
        if (count == 20) {
            hal_timer_stop(&timer1);
            hal_timer_finalize(&timer1);
        }

        /* sleep 1000ms */
        aos_msleep(1000);
        count++;
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_timer_xxmcu.c和hal_timer_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_timer_xxmcu.c中实现所需要的hal函数，hal_timer_xxmcu.h中放置相关宏定义。

注意事项

1、Timer 计数寄存器为24bit,最大支持4.1s定时。

2、SDK V1.2.6版本中硬件Timer的初始化需要如下改动才能支持多个定时器同时工作。

```
timer_handle_t csi_timer_initialize(int32_t idx, timer_event_cb_t cb_event)
{
...
    if (g_timer_active_num == 0) {
        clk_gate_enable(MOD_TIMER);
        drv_irq_register(timer_priv->irq, handler);
        timer_deactive_control(addr);
        drv_irq_enable(timer_priv->irq);
    }

    g_timer_active_num++;
...
}
```

SPI

接口列表

函数名称	功能描述
hal_spi_init	初始化指定SPI端口
hal_spi_send	从指定的SPI端口发送数据
hal_spi_recv	从指定的SPI端口接收数据
hal_spi_send_recv	从指定的SPI端口发送并接收数据
hal_spi_finalize	关闭指定SPI端口

接口详情

int32_t hal_spi_init(spi_dev_t *spi)

描述	初始化指定SPI端口
参数	spi：SPI设备描述，定义需要初始化的SPI参数
返回值	返回成功或失败, 返回0表示SPI初始化成功，非0表示失败

int32_t hal_spi_send(spi_dev_t *spi, const uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的SPI端口发送数据
参数	spi：SPI设备描述
	data：指向要发送数据的数据指针
	size：要发送的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示SPI数据发送成功，非0表示失败

int32_t hal_spi_recv(spi_dev_t *spi, uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的SPI端口接收数据
参数	spi：SPI设备描述
	data：指向接收缓冲区的数据指针
	size：期望接收的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

int32_t hal_spi_send_recv(spi_dev_t *spi, uint8_t* tx_data, uint8_t *rx_data, uint16_t size, uint32_t timeout)

描述	从指定的SPI端口发送并接收数据
参数	spi：SPI设备描述
	tx_data：指向接收缓冲区的数据指针
	rx_data：指向发送缓冲区的数据指针
	size：要发送和接收数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功发送和接收size个数据，非0表示失败

int32_t hal_spi_finalize(spi_dev_t *spi)

描述	关闭指定SPI端口
参数	spi：SPI设备描述
返回值	类型：int 返回成功或失败, 返回0表示SPI关闭成功，非0表示失败。

相关宏定义

```
#define HAL_SPI_MODE_MASTER 1 /* spi communication is master mode */
#define HAL_SPI_MODE_SLAVE  2 /* spi communication is slave mode */
```

相关结数据结构

spi_dev_t

```
typedef struct {
    uint8_t      port;   /* spi port */
    spi_config_t config; /* spi config */
    void        *priv;   /* priv data */
} spi_dev_t;
```

spi_config_t

```
typedef struct {
    uint32_t mode; /* spi communication mode */
    uint32_t freq; /* communication frequency Hz */
} spi_config_t;
```

使用示例

```
#include <hal/soc/spi.h>

#define SPI1_PORT_NUM  1
#define SPI_BUF_SIZE   10
#define SPI_TX_TIMEOUT 10
#define SPI_RX_TIMEOUT 10

/* define dev */
spi_dev_t spi1;

/* data buffer */
```

```
char spi_data_buf[SPI_BUF_SIZE];

int application_start(int argc, char *argv[])
{
    int count   = 0;
    int ret     = -1;
    int i       = 0;
    int rx_size = 0;

    /* spi port set */
    spi1.port = SPI1_PORT_NUM;

    /* spi attr config */
    spi1.config.mode  = HAL_SPI_MODE_MASTER;
    spi1.config.freq = 30000000;

    /* init spi1 with the given settings */
    ret = hal_spi_init(&spi1);
    if (ret != 0) {
        printf("spi1 init error !\n");
    }

    /* init the tx buffer */
    for (i = 0; i < SPI_BUF_SIZE; i++) {
        spi_data_buf[i] = i + 1;
    }

    /* send 0,1,2,3,4,5,6,7,8,9 by spi1 */
    ret = hal_spi_send(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_TX_TIMEOUT);
    if (ret == 0) {
        printf("spi1 data send succeed !\n");
    }

    /* scan spi every 100ms to get the data */
    while(1) {
        ret = hal_spi_recv(&spi1, spi_data_buf, SPI_BUF_SIZE, SPI_RX_TIMEOUT);
        if (ret == 0) {
            printf("spi1 data received succeed !\n");
        }

        /* sleep 100ms */
        aos_msleep(100);
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_spi_xxmcu.c和hal_spi_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_spi_xxmcu.c中实现所需要的hal函数，hal_spi_xxmcu.h中放置相关宏定义。

注意事项

目前MCU主频为48M

(1).master发送最高的速率是12M。

(2).slave(receive only)， 支持最大速率8M。

(3).slave(receive and send) ,支持最大速率6M。

I2C

接口列表

函数名称	功能描述
hal_i2c_init	初始化指定I2C端口
hal_i2c_master_send	master模式下从指定的I2C端口发送数据
hal_i2c_master_recv	master模式下从指定的I2C端口接收数据
hal_i2c_slave_send	slave模式下从指定的I2C端口发送数据
hal_i2c_slave_recv	slave模式下从指定的I2C端口接收数据
hal_i2c_mem_write	mem模式下从指定的I2C端口发送数据
hal_i2c_mem_read	mem模式下从指定的I2C端口接收数据
hal_i2c_finalize	关闭指定I2C端口

接口详情

int32_t hal_i2c_init(i2c_dev_t *i2c)

描述	初始化指定I2C端口
参数	i2c：I2C设备描述，定义需要初始化的I2C参数
返回值	返回成功或失败, 返回0表示I2C初始化成功，非0表示失败

int32_t hal_i2c_master_send(i2c_dev_t *i2c, uint16_t dev_addr, const uint8_t* data, uint16_t size, uint32_t timeout)

描述	master模式下从指定的I2C端口发送数据
参数	i2c：I2C设备描述
	dev_addr：目标设备地址
	data：指向发送缓冲区的数据指针
	size：要发送的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示I2C数据发送成功，非0表示失败

int32_t hal_i2c_master_recv(i2c_dev_t *i2c, uint16_t dev_addr, uint8_t* data, uint16_t size, uint32_t timeout)

描述	master模式下从指定的I2C端口接收数据
参数	i2c：I2C设备描述
	dev_addr：目标设备地址
	data：指向接收缓冲区的数据指针
	size：期望接收的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

int32_t hal_i2c_slave_send(i2c_dev_t *i2c, const uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的I2C端口发送并接收数据
参数	i2c：I2C设备描述
	data：指向发送缓冲区的数据指针
	size：期望发送的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功发送size个数据，非0表示失败

int32_t hal_i2c_slave_recv(i2c_dev_t *i2c, uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的I2C端口接收数据
参数	i2c：I2C设备描述
	data：指向要接收数据的数据指针
	size：要接收的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

int32_t hal_i2c_mem_write(i2c_dev_t *i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, const uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的I2C端口接收数据
参数	i2c：I2C设备描述
	dev_addr：目标设备地址
	mem_addr：内部内存地址
	mem_addr_size：内部内存地址大小
	data：指向要发送数据的数据指针
	size：要发送的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功发送size个数据，非0表示失败

int32_t hal_i2c_mem_read(i2c_dev_t *i2c, uint16_t dev_addr, uint16_t mem_addr, uint16_t mem_addr_size, uint8_t* data, uint16_t size, uint32_t timeout)

描述	从指定的I2C端口发送并接收数据
参数	i2c：I2C设备描述
	dev_addr：目标设备地址
	mem_addr：内部内存地址
	mem_addr_size：内部内存地址大小
	data：指向接收缓冲区的数据指针
	size：要接收的数据字节数
	timeout：超时时间（单位ms），如果希望一直等待设置为HAL_WAIT_FOREVER
返回值	返回成功或失败, 返回0表示成功接收size个数据，非0表示失败

int32_t hal_i2c_finalize(i2c_dev_t *i2c)

描述	关闭指定I2C端口
参数	i2c：I2C设备描述
返回值	类型：int 返回成功或失败, 返回0表示I2C关闭成功，非0表示失败。

相关宏定义

```
#define I2C_MODE_MASTER 1 /* i2c communication is master mode */
#define I2C_MODE_SLAVE  2 /* i2c communication is slave mode */

#define I2C_MEM_ADDR_SIZE_8BIT  1 /* i2c menory address size 8bit */
#define I2C_MEM_ADDR_SIZE_16BIT 2 /* i2c menory address size 16bit */

/*
 * Specifies one of the standard I2C bus bit rates for I2C communication
 */
#define I2C_BUS_BIT_RATES_100K  100000
#define I2C_BUS_BIT_RATES_400K  400000
#define I2C_BUS_BIT_RATES_3400K 3400000

#define I2C_HAL_ADDRESS_WIDTH_7BIT  0
#define I2C_HAL_ADDRESS_WIDTH_10BIT 1
```

相关结数据结构

i2c_dev_t

```
typedef struct {
    uint8_t      port;   /* i2c port */
    i2c_config_t config; /* i2c config */
    void        *priv;   /* priv data */
} i2c_dev_t;
```

i2c_config_t

```
typedef struct {
    uint32_t address_width;
    uint32_t freq;
    uint8_t  mode;
    uint16_t dev_addr;
} i2c_config_t;
```

使用示例

```
#include <hal/soc/i2c.h>

#define I2C1_PORT_NUM  1
#define I2C_BUF_SIZE   10
#define I2C_TX_TIMEOUT 10
#define I2C_RX_TIMEOUT 10

#define I2C_DEV_ADDR       0x30f
#define I2C_DEV_ADDR_WIDTH I2C_HAL_ADDRESS_WIDTH_7BIT

/* define dev */
i2c_dev_t i2c1;

/* data buffer */
char i2c_data_buf[I2C_BUF_SIZE];

int application_start(int argc, char *argv[])
{
    int count   = 0;
    int ret     = -1;
    int i       = 0;
    int rx_size = 0;

    drv_pinmux_config(IIC_SCL_PIN, IIC1_SCL);
    drv_pinmux_config(IIC_SDA_PIN, IIC1_SDA);

    /* i2c port set */
    i2c1.port = I2C1_PORT_NUM;

    /* i2c attr config */
    i2c1.config.mode          = I2C_MODE_MASTER;
    i2c1.config.freq          = 30000000;
```

```
    i2c1.config.address_width = I2C_DEV_ADDR_WIDTH;
    i2c1.config.dev_addr      = I2C_DEV_ADDR;

    /* init i2c1 with the given settings */
    ret = hal_i2c_init(&i2c1);
    if (ret != 0) {
        printf("i2c1 init error !\n");
    }

    /* init the tx buffer */
    for (i = 0; i < I2C_BUF_SIZE; i++) {
        i2c_data_buf[i] = i + 1;
    }

    /* send 0,1,2,3,4,5,6,7,8,9 by i2c1 */
    ret = hal_i2c_master_send(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                              I2C_BUF_SIZE, I2C_TX_TIMEOUT);
    if (ret == 0) {
        printf("i2c1 data send succeed !\n");
    }

    ret = hal_i2c_master_recv(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                              I2C_BUF_SIZE, I2C_RX_TIMEOUT);
    if (ret == 0) {
        printf("i2c1 data received succeed !\n");
    }

    while(1) {
        printf("AliOS Things is working !\n");

        /* sleep 1000ms */
        aos_msleep(1000);
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_i2c_xxmcu.c和hal_i2c_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_i2c_xxmcu.c中实现所需要的hal函数，hal_i2c_xxmcu.h中放置相关宏定义。

ADC

接口列表

函数名称	功能描述
hal_adc_init	初始化指定ADC
hal_adc_value_get	获取ADC采样值
hal_adc_finalize	关闭指定ADC

接口详情

int32_t hal_adc_init(adc_dev_t *adc)

描述	初始化指定ADC
参数	adc：ADC设备描述
返回值	返回成功或失败, 返回0表示ADC初始化成功，非0表示失败

int32_t hal_adc_value_get(adc_dev_t *adc, void* output, uint32_t timeout)

描述	获取ADC采样值
参数	adc：ADC设备描述
	output：数据缓冲区
	timeout：超时时间
返回值	返回成功或失败, 返回0表示ADC时间获取成功，非0表示失败

int32_t hal_adc_finalize(adc_dev_t *adc)

描述	关闭指定ADC
参数	adc：ADC设备描述
返回值	返回成功或失败, 返回0表示ADC时间设定成功，非0表示失败

相关结数据结构

adc_dev_t

```
typedef struct {
    uint8_t      port;   /* adc port,port:2-7 to gpio:12 11 14 13 20 15 voice
channel*/
    adc_config_t config; /* adc config */
    void        *priv;   /* priv data */
} adc_dev_t;
```

adc_config_t

```
typedef struct {
    uint32_t sampling_cycle;  /* sampling period in number of ADC clock cycles
*/
} adc_config_t;
```

使用示例

定义P20管脚采集外部电压值

```
#include <hal/soc/adc.h>
#include "adc.h"
#include "gpio.h"

/* define dev */
adc_dev_t adc1;

int application_start(int argc, char *argv[])
{
    int ret   = -1;
    int value = 0;

    adc_config_t adc_cfg;

    drv_pinmux_config(P20, ADCC);

    /* adc port set */
    adc1.port = hal_adc_pin2channel(P20);

    /* set sampling_cycle */
    adc1.config.sampling_cycle = 100;

    /* init adc1 with the given settings */
    ret = hal_adc_init(&adc1);
    if (ret != 0) {
        printf("adc1 init error !\n");
    }

    /* get adc value */
    ret = hal_adc_value_get(&adc1, &value, HAL_WAIT_FOREVER);
    if (ret != 0) {
```

```
        printf("adc1 vaule get error !\n");
    }

    /* finalize adc1 */
    hal_adc_finalize(&adc1);

    while(1) {
        /* sleep 500ms */
        aos_msleep(500);
    };
}
```

注：port为逻辑频道号，其与物理端口号的对应关系参考platform\mcu\tg7100b\csi\csi_driver\phyplus\common\include\adc.h中ADC Channel的定义adc_CH_t

移植说明

新建hal_adc_xxmcu.c和hal_adc_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_adc_xxmcu.c中实现所需要的hal函数，hal_adc_xxmcu.h中放置相关宏定义。

注意事项

(1).ADC参考电压为1V，量程为0-3.3V。

(2).ADC误差范围+-2%。

(3).ADC电压采样值单位为mV

(4).ADC差分使用示例

```
配置channel时需要配置positive channel,如ADC_CH1P，ADC_CH2P，ADC_CH3P
const static GPIO_Pin_e s_pinmap[ADC_CH_NUM] = {
  GPIO_DUMMY, //ADC_CH0 =0,
  GPIO_DUMMY, //ADC_CH1 =1,
  P12, //ADC_CH1P =2,  ADC_CH1DIFF = 2,
  P11, //ADC_CH1N =3,
  P14, //ADC_CH2P =4,  ADC_CH2DIFF = 4,
  P13, //ADC_CH2N =5,
  P20, //ADC_CH3P =6,  ADC_CH3DIFF = 6,
  P15, //ADC_CH3N =7,
  GPIO_DUMMY,  //ADC_CH_VOICE =8,
};

adc_Cfg_t cfg = {
      .is_continue_mode = FALSE,
      .is_differential_mode = TRUE,
      .is_high_resolution = FALSE,
      .is_auto_mode = FALSE,
};

int test_diff_adc(void)
{
```

```
    int i = 0;
    int ret = 0;

    phy_adc_init();

    adc_CH_t channel = ADC_CH3P;
    GPIO_Pin_e pin = s_pinmap[channel];

    ret = phy_adc_config_channel(channel, cfg, NULL);
    if(ret < 0){
        return 0;
    }
    phy_adc_start_int_dis();

    return adc_poilling(ADC_CH3P);
}
```

(5).TG7100B芯片内部集成了分压电路，支持内部电压采集功能。

内部电压采集功能的配置，需要修改platform\mcu\tg7100b\hal\adc.c中

```
int32_t hal_adc_value_get(adc_dev_t *adc, void *output, uint32_t timeout)
{
    ...
    //enable_link_internal_voltage配置为1，打开内部电压采集功能
    sconfig.enable_link_internal_voltage = ADC_INTERNAL_CAP_ENABLE;
    ...
}
```

(6).使用外部分压电路做ADC采集时，为了保证精度，需保证IO的输入电压在1V左右。具体设计参考电路可以查看硬件参考设计文档。

RTC

接口列表

函数名称	功能描述
hal_rtc_init	初始化指定RTC
hal_rtc_get_time	获取指定RTC时间
hal_rtc_set_time	设置指定RTC时间
hal_rtc_finalize	关闭指定RTC

接口详情

int32_t hal_rtc_init(rtc_dev_t *rtc)

描述	初始化指定RTC
参数	rtc：RTC设备描述
	time：存储时间的缓冲区
返回值	返回成功或失败, 返回0表示RTC初始化成功，非0表示失败

int32_t hal_rtc_get_time(rtc_dev_t *rtc*, *rtc_time_t* time)

描述	获取指定RTC时间
参数	rtc：RTC设备描述
	time：要设定的时间
返回值	返回成功或失败, 返回0表示RTC时间获取成功，非0表示失败

int32_t hal_rtc_set_time(rtc_dev_t *rtc*, *const rtc_time_t* time)

描述	设置指定RTC时间
参数	rtc：RTC设备描述
返回值	返回成功或失败, 返回0表示RTC时间设定成功，非0表示失败

int32_t hal_rtc_finalize(rtc_dev_t *rtc)

描述	关闭指定RTC
参数	rtc：RTC设备描述
返回值	返回成功或失败, 返回0表示RTC关闭成功，非0表示失败

相关结数据结构

```
#define HAL_RTC_FORMAT_DEC 1
#define HAL_RTC_FORMAT_BCD 2
```

相关结数据结构

rtc_dev_t

```
typedef struct {
    uint8_t      port;   /* rtc port */
    rtc_config_t config; /* rtc config */
    void        *priv;   /* priv data */
} rtc_dev_t;
```

rtc_config_t

```
typedef struct {
    uint8_t  format; /* time formart DEC or BCD */
} rtc_config_t;
```

rtc_time_t

```
typedef struct {
    uint8_t sec;     /* DEC format:value range from 0 to 59, BCD format:value
range from 0x00 to 0x59 */
    uint8_t min;     /* DEC format:value range from 0 to 59, BCD format:value
range from 0x00 to 0x59 */
    uint8_t hr;      /* DEC format:value range from 0 to 23, BCD format:value
range from 0x00 to 0x23 */
    uint8_t weekday; /* DEC format:value range from 1 to  7, BCD format:value
range from 0x01 to 0x07 */
    uint8_t date;    /* DEC format:value range from 1 to 31, BCD format:value
range from 0x01 to 0x31 */
    uint8_t month;   /* DEC format:value range from 1 to 12, BCD format:value
range from 0x01 to 0x12 */
    uint8_t year;    /* DEC format:value range from 0 to 99, BCD format:value
range from 0x00 to 0x99 */
} rtc_time_t;
```

使用示例

```
#include <hal/soc/rtc.h>

#define RTC1_PORT_NUM 0

/* define dev */
rtc_dev_t rtc1;

int application_start(int argc, char *argv[])
{
```

```
    int ret = -1;

    rtc_config_t rtc_cfg;
    rtc_time_t   time_buf;

    /* rtc port set */
    rtc1.port = RTC1_PORT_NUM;

    /* set to DEC format */
    rtc1.config.format = HAL_RTC_FORMAT_DEC;

    /* init rtc1 with the given settings */
    ret = hal_rtc_init(&rtc1);
    if (ret != 0) {
        printf("rtc1 init error !\n");
    }

    time_buf.sec     = 0;
    time_buf.min     = 0;
    time_buf.hr      = 0;
    time_buf.weekday = 2;
    time_buf.date    = 1;
    time_buf.month   = 1;
    time_buf.year    = 19;

    /* set rtc1 time to 2019/1/1,00:00:00 */
    ret = hal_rtc_set_time(&rtc1, &time_buf);
    if (ret != 0) {
        printf("rtc1 set time error !\n");
    }

    memset(&time_buf, 0, sizeof(rtc_time_t));

    while(1) {
        /* sleep 5000ms */
        aos_msleep(5000);

        /* get rtc current time */
        ret = hal_rtc_get_time(&rtc1, &time_buf);
        if (ret != 0) {
            printf("rtc1 get time error !\n");
        }
        printf("rtc1 get time %d:%d:%d %d:%d:%d!\n", time_buf.year,
                                                     time_buf.month,
                                                     time_buf.date,
                                                     time_buf.hr,
                                                     time_buf.min,
                                                     time_buf.sec);
    };

    /* finalize rtc1 */
    hal_rtc_finalize(&rtc1);
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

PWM

接口列表

函数名称	功能描述
hal_pwm_init	初始化指定PWM
hal_pwm_start	开始输出指定PWM
hal_pwm_stop	停止输出指定PWM
hal_pwm_para_chg	修改指定PWM参数
hal_pwm_finalize	关闭指定PWM

接口详情

int32_t hal_pwm_init(pwm_dev_t *pwm)

描述	初始化指定PWM
参数	pwm：PWM设备描述，定义需要初始化的PWM参数
返回值	返回成功或失败, 返回0表示PWM初始化成功，非0表示失败

int32_t hal_pwm_start(pwm_dev_t *pwm)

描述	开始输出指定PWM
参数	pwm：PWM设备描述
返回值	返回成功或失败, 返回0表示PWM开始输出成功，非0表示失败

int32_t hal_pwm_stop(pwm_dev_t *pwm)

描述	停止输出指定PWM
参数	pwm：PWM设备描述
返回值	返回成功或失败, 返回0表示PWM停止输出成功，非0表示失败

int32_t hal_pwm_para_chg(pwm_dev_t *pwm, pwm_config_t para)

描述	修改指定PWM参数
参数	pwm：PWM设备描述
	para：新配置参数
返回值	返回成功或失败, 返回0表示PWM参数修改成功，非0表示失败

int32_t hal_pwm_finalize(pwm_dev_t *pwm)

描述	关闭指定PWM
参数	pwm：PWM设备描述
返回值	返回成功或失败, 返回0表示PWM关闭成功，非0表示失败

相关结数据结构

pwm_dev_t

```
typedef struct {
    uint8_t      port;   /* pwm port */
    pwm_config_t config; /* pwm config */
    void        *priv;   /* priv data */
} pwm_dev_t;
```

pwm_config_t

```
typedef struct {
    uint32_t duty_cycle; /* the pwm duty_cycle */
    uint32_t freq;       /* the pwm freq */
} pwm_config_t;
```

使用示例

```
#include <hal/soc/pwm.h>
```

```c
#define PWM1_PORT_NUM 1

/* define dev */
pwm_dev_t pwm1;

int application_start(int argc, char *argv[])
{
    int ret = -1;
    pwm_config_t pwm_cfg;
    static int count = 0;

    drv_pinmux_config(PWM_PIN, PWM1);

    /* pwm port set */
    pwm1.port = PWM1_PORT_NUM;

    /* pwm attr config */
    pwm1.config.duty_cycle = 1000; /* 1000us */
    pwm1.config.freq       = 500; /* 500hz 2000us */

    /* init pwm1 with the given settings */
    ret = hal_pwm_init(&pwm1);
    if (ret != 0) {
        printf("pwm1 init error !\n");
    }

    /* start pwm1 */
    ret = hal_pwm_start(&pwm1);
    if (ret != 0) {
        printf("pwm1 start error !\n");
    }

    while(1) {

        /* change the duty cycle to 25% */
        if (count == 5) {
            memset(&pwm_cfg, 0, sizeof(pwm_config_t));
            pwm_cfg.duty_cycle = 500;/* 500us */
            pwm_cfg.freq = 500;/* 2000us */
            ret = hal_pwm_para_chg(&pwm1, pwm_cfg);
            if (ret != 0) {
                printf("pwm1 para change error !\n");
            }
        }

        /* stop and finalize pwm1 */
        if (count == 20) {
            hal_pwm_stop(&pwm1);
            hal_pwm_finalize(&pwm1);
        }

        /* sleep 1000ms */
        aos_msleep(1000);
        count++;
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

> 新建hal_pwm_xxmcu.c和hal_pwm_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_pwm_xxmcu.c中实现所需要的hal函数，hal_pwm_xxmcu.h中放置相关宏定义。

注意事项

(1).PWM最大周期不能超过4095us。也就是频率不能低于245Hz。

(2).hal_pwm_stop停止输出电平为高。如需保持电平建议配置GPIO输出控制。

(3).duty_cycle设置为0%时会有毛刺，需要用gpio拉低代替设置0。

(4).duty_cycle设置为100%时会有毛刺，需要用gpio拉高代替。

FLASH

接口列表

函数名称	功能描述
hal_flash_get_info	获取指定区域的FLASH信息
hal_flash_erase	擦除FLASH的指定区域
hal_flash_write	写FLASH的指定区域
hal_flash_erase_write	先擦除再写FLASH的指定区域
hal_flash_read	读FLASH的指定区域
hal_flash_enable_secure	使能加密FLASH的指定区域(暂不支持)
hal_flash_dis_secure	关闭加密FLASH的指定区域(暂不支持)
hal_flash_addr2offset	将物理地址转换为分区号和偏移

接口详情

hal_logic_partition_t *hal_flash_get_info(hal_partition_t in_partition)

描述	获取指定区域的FLASH信息
参数	in_partition：FLASH分区
返回值	成功则返回分区信息，否则返回NULL

int32_t hal_flash_erase(hal_partition_t in_partition, uint32_t off_set, uint32_t size)

描述	擦除FLASH的指定区域
参数	in_partition：FLASH分区
	off_set：偏移量
	size：要擦除的字节数
返回值	返回成功或失败, 返回0表示擦除成功，非0表示失败

int32_t hal_flash_write(hal_partition_t in_partition, uint32_t *off_set, const void* in_buf, uint32_t in_buf_len)

描述	写FLASH的指定区域
参数	in_partition：FLASH分区
	off_set：偏移量
	in_buf：指向要写入数据的指针
	in_buf_len：要写入的字节数
返回值	返回成功或失败, 返回0表示写入成功，非0表示失败

int32_t hal_flash_erase_write(hal_partition_t in_partition, uint32_t *off_set, const void* in_buf, uint32_t in_buf_len)

描述	先擦除再写FLASH的指定区域
参数	in_partition：FLASH分区
	off_set：偏移量
	in_buf：指向要写入数据的指针
	in_buf_len：要写入的字节数
返回值	返回成功或失败, 返回0表示TIMER参数改变成功，非0表示失败

int32_t hal_flash_read(hal_partition_t in_partition, uint32_t *off_set, void* out_buf, uint32_t in_buf_len)

描述	关闭指定TIMER
参数	in_partition：FLASH分区
	off_set：偏移量
	out_buf：数据缓冲区地址
	in_buf_len：要写入的字节数
返回值	返回成功或失败, 返回0表示TIMER关闭成功，非0表示失败

int32_t hal_flash_enable_secure(hal_partition_t partition, uint32_t off_set, uint32_t size)

描述	使能加密FLASH的指定区域
参数	partition：FLASH分区
	off_set：偏移量
	size：使能区域字节数
返回值	返回成功或失败, 返回0表示使能成功，非0表示失败

int32_t hal_flash_dis_secure(hal_partition_t partition, uint32_t off_set, uint32_t size)

描述	关闭加密FLASH的指定区域
参数	partition：FLASH分区
	off_set：偏移量
	size：使能区域字节数
返回值	返回成功或失败, 返回0表示关闭成功，非0表示失败

int32_t hal_flash_addr2offset(hal_partition_t *in_partition, uint32_t* off_set, uint32_t addr)

描述	将物理地址转换为分区号和偏移
参数	in_partition：FLASH分区
	off_set：偏移量
	addr：要转换的物理地址
返回值	返回成功或失败, 返回0表示转换成功，非0表示失败。

相关宏定义

```
#define PAR_OPT_READ_POS  ( 0 )
#define PAR_OPT_WRITE_POS ( 1 )

#define PAR_OPT_READ_MASK  ( 0x1u << PAR_OPT_READ_POS )
#define PAR_OPT_WRITE_MASK ( 0x1u << PAR_OPT_WRITE_POS )

#define PAR_OPT_READ_DIS  ( 0x0u << PAR_OPT_READ_POS )
#define PAR_OPT_READ_EN   ( 0x1u << PAR_OPT_READ_POS )
#define PAR_OPT_WRITE_DIS ( 0x0u << PAR_OPT_WRITE_POS )
#define PAR_OPT_WRITE_EN  ( 0x1u << PAR_OPT_WRITE_POS )
```

相关结数据结构

hal_logic_partition_t

```
typedef struct {
    hal_flash_t partition_owner;
    const char *partition_description;
    uint32_t    partition_start_addr;
    uint32_t    partition_length;
    uint32_t    partition_options;
} hal_logic_partition_t;
```

hal_flash_t

```
typedef enum {
    HAL_FLASH_EMBEDDED,
    HAL_FLASH_SPI,
    HAL_FLASH_QSPI,
    HAL_FLASH_MAX,
    HAL_FLASH_NONE,
} hal_flash_t;
```

hal_partition_t

```
typedef enum {
    HAL_PARTITION_ERROR = -1,
    HAL_PARTITION_BOOTLOADER,
    HAL_PARTITION_APPLICATION,
    HAL_PARTITION_ATE,
    HAL_PARTITION_OTA_TEMP,
    HAL_PARTITION_RF_FIRMWARE,
    HAL_PARTITION_PARAMETER_1,
    HAL_PARTITION_PARAMETER_2,
    HAL_PARTITION_PARAMETER_3,
    HAL_PARTITION_PARAMETER_4,
    HAL_PARTITION_BT_FIRMWARE,
    HAL_PARTITION_SPIFFS,
    HAL_PARTITION_CUSTOM_1,
    HAL_PARTITION_CUSTOM_2,
    HAL_PARTITION_RECOVERY,
    HAL_PARTITION_RECOVERY_BACK_PARA,
    HAL_PARTITION_MAX,
    HAL_PARTITION_NONE,
} hal_partition_t;
```

WDG

接口列表

函数名称	功能描述
hal_wdg_init	初始化指定看门狗
hal_wdg_reload	重载指定看门狗，喂狗
hal_wdg_finalize	关闭指定看门狗

接口详情

int32_t hal_wdg_init(wdg_dev_t *wdg)

描述	初始化指定看门狗
参数	wdg：看门狗设备描述
返回值	返回成功或失败, 返回0表示看门狗初始化成功，非0表示失败

void hal_wdg_reload(wdg_dev_t *wdg)

描述	重载指定看门狗，喂狗
参数	wdg：看门狗设备描述
返回值	返回成功或失败, 返回0表示看门狗重载成功，非0表示失败

int32_t hal_wdg_finalize(wdg_dev_t *wdg)

描述	关闭指定看门狗
参数	wdg：看门狗设备描述
返回值	返回成功或失败, 返回0表示看门狗关闭成功，非0表示失败

相关结数据结构

wdg_dev_t

```
typedef struct {
    uint8_t      port;   /* wdg port */
    wdg_config_t config; /* wdg config */
    void        *priv;   /* priv data */
} wdg_dev_t;
```

wdg_config_t

```
typedef struct {
    uint32_t timeout; /* Watchdag timeout ms */
} wdg_config_t;
```

使用示例

```
#include <hal/soc/wdg.h>
#include <drv_wdt.h>

int32_t csi_wdt_config_feed(int32_t idx, uint8_t auto_feed);
#define WDG1_PORT_NUM 0
```

```c
/* define dev */
wdg_dev_t wdg1;

int application_start(int argc, char *argv[])
{
    int ret = -1;
    wdg_config_t wdg_cfg;
    static int count = 0;

    /* wdg port set */
    wdg1.port = WDG1_PORT_NUM;

    /* set reload time to 2000ms */
    wdg1.config.timeout = 2000; /* 2000ms */

    /* init wdg1 with the given settings */
    ret = hal_wdg_init(&wdg1);
    if (ret != 0) {
        printf("wdg1 init error !\n");
    }

    /* config to auto feed mode with wdg1 */
    ret = csi_wdt_config_feed(wdg1.port, WDT_FEED_MANU);
    if (ret != 0) {
        printf("wdg auto feed config error !\n");
    }

    while(1) {
        /* clear wdg about every 500ms */
        hal_wdg_reload(&wdg1);

        /* finalize wdg1 */
        if (count == 20) {
            hal_wdg_finalize(&wdg1);
        }

        /* sleep 500ms */
        aos_msleep(500);
        count++;
    };
}
```

注：port为逻辑端口号，其与物理端口号的对应关系见具体的对接实现

移植说明

新建hal_wdg_xxmcu.c和hal_wdg_xxmcu.h的文件，并将这两个文件放到platform/mcu/xxmcu/hal目录下。在hal_wdg_xxmcu.c中实现所需要的hal函数，hal_wdg_xxmcu.h中放置相关宏定义。

注意事项：

(1).定时时间固定2s,不可设。

(2).可支持自动喂狗和手动喂狗两种模式，驱动实现中默认为手动喂狗，需要用户自定义喂狗逻辑；自动喂狗，驱动实现会在中断处理函数中自动喂狗。

```
csi_wdt_config_feed(wdg1.port, WDT_FEED_AUTO); //设置为自动喂狗
csi_wdt_config_feed(wdg1.port, WDT_FEED_MANU); //设置为手动喂狗
```

(3).低功耗功能使能后，需要在唤醒时重新启动看门狗。

(4).自动喂狗示例

```
light_ctl示例和node_ctl示例中有自动喂狗示例的配置，
参考light_ctl.mk和node_ctl.mk文件，打开配置即可。

genie_wdt_config = 1
```