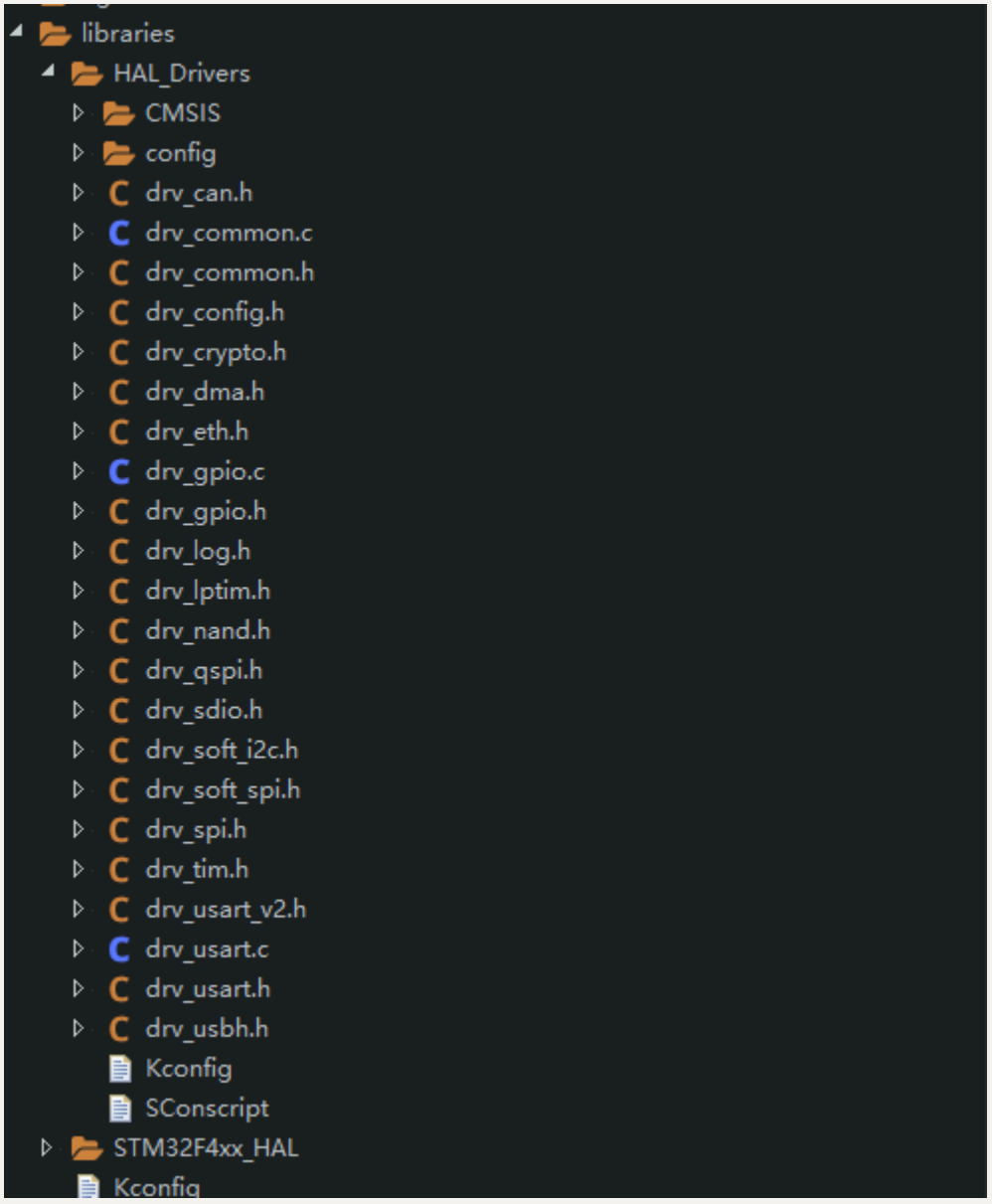
1.使能引脚

1.RT将stm32中的引脚使能封装成自己独有的函数GET_PIN

```
#if defined(SOC_SERIES_STM32MP1)
#define GET_PIN(PORTx,PIN) (GPIO##PORTx == GPIOZ) ? (176 + PIN) :
((rt_base_t)((16 * ( ((rt_base_t)__STM32_PORT(PORTx) -
(rt_base_t)GPIOA_BASE)/(0x1000UL) )) + PIN))
#else
#define GET_PIN(PORTx,PIN) (rt_base_t)((16 * (
((rt_base_t)__STM32_PORT(PORTx) - (rt_base_t)GPIOA_BASE)/(0x0400UL) ))
+ PIN)
#endif
```

注:该类gpio口的封装全部在drv_gpio.c中



2.在实际使用中常通过define的形式引出需要的引脚

```
#define PIN_BEEP        GET_PIN(B, 0)      // PA1:  BEEP         -->
BEEP (PB1)
```

3.配置引脚模式的函数

```
void rt_pin_mode(rt_base_t pin, rt_base_t mode)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);
    _hw_pin.ops->pin_mode(&_hw_pin.parent, pin, mode);
}
```

1.在pin.c和pin.h中,有关于pin设置的结构体和函数

这里列出函数中关于pin_mode 的一些定义:

```
#define PIN_LOW                 0x00  // 定义 PIN 的低电平状态。0x00 表示低电平
#define PIN_HIGH                0x01  // 定义 PIN 的高电平状态。0x01 表示高电平

#define PIN_MODE_OUTPUT         0x00  // 定义 PIN 的输出模式。0x00 表示该引脚被配置为输出模式。
#define PIN_MODE_INPUT          0x01  // 定义 PIN 的输入模式。0x01 表示该引脚被配置为普通输入模式。
#define PIN_MODE_INPUT_PULLUP   0x02  // 定义 PIN 的输入模式并启用上拉电阻。0x02 表示输入模式并启用内置上拉电阻。
#define PIN_MODE_INPUT_PULLDOWN 0x03  // 定义 PIN 的输入模式并启用下拉电阻。0x03 表示输入模式并启用内置下拉电阻。
#define PIN_MODE_OUTPUT_OD      0x04  // 定义 PIN 的开漏输出模式。0x04 表示该引脚被配置为开漏输出模式（Open Drain）。

#define PIN_IRQ_MODE_RISING             0x00  // 定义 PIN 的中断触发模式为上升沿触发。0x00 表示该引脚的中断触发在信号由低到高的变化时发生。
#define PIN_IRQ_MODE_FALLING            0x01  // 定义 PIN 的中断触发模式为下降沿触发。0x01 表示该引脚的中断触发在信号由高到低的变化时发生。
#define PIN_IRQ_MODE_RISING_FALLING     0x02  // 定义 PIN 的中断触发模式为上升沿和下降沿触发。0x02 表示该引脚的中断触发在信号的上升沿或下降沿时都可以发生。
#define PIN_IRQ_MODE_HIGH_LEVEL         0x03  // 定义 PIN 的中断触发模式为高电平触发。0x03 表示该引脚的中断触发在信号为高电平时发生。
```

```
#define PIN_IRQ_MODE_LOW_LEVEL          0x04   // 定义 PIN 的中断触发模式为低电平触发。0x04 表示该引脚的中断触发在信号为低电平时发生。

#define PIN_IRQ_DISABLE                 0x00   // 定义中断禁用模式。0x00 表示禁用该引脚的中断功能。
#define PIN_IRQ_ENABLE                  0x01   // 定义中断启用模式。0x01 表示启用该引脚的中断功能。

#define PIN_IRQ_PIN_NONE                -1     // 定义无效的引脚编号。-1 表示该引脚不与任何中断相关联，或者是未配置的引脚。
```

2.使用例程:

```
#define PIN_BEEP        GET_PIN(B, 0)      // PA1:  BEEP         --> BEEP (PB1)
rt_pin_mode(PIN_BEEP, PIN_MODE_OUTPUT);
```

4. 关于引脚状态的函数

```
void rt_pin_write(rt_base_t pin, rt_base_t value)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);
    _hw_pin.ops->pin_write(&_hw_pin.parent, pin, value);
}

int rt_pin_read(rt_base_t pin)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);
    return _hw_pin.ops->pin_read(&_hw_pin.parent, pin);
}


rt_base_t rt_pin_get(const char *name)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);

    if (name[0] != 'P' && name[0] != 'p')
    {
```

```
        return -RT_EINVAL;
    }
    if (_hw_pin.ops->pin_get == RT_NULL)
    {
        return -RT_ENOSYS;
    }
    return _hw_pin.ops->pin_get(name);
}
```

注:其中rt_pin_write和rt_pin_read都是很简单的设置(读取)引脚电平的解释,这里不过多解释

```
rt_pin_write(PIN_BEEP,PIN_HIGH);
```

主要解释一下,rt_pin_get函数的作用:

- `rt_pin_get` 是用来获取引脚句柄的函数，通过引脚的名称来识别和获取引脚对象。
- 它的返回值是该引脚的句柄，成功时返回有效句柄，失败时返回负数错误码。
- 获取引脚句柄后，可以进一步进行如读取、写入等操作。
- 通过引脚句柄操作引脚时，需要注意引脚名称的正确性及硬件支持。

这个函数为开发提供了更灵活的硬件抽象，允许开发者根据名字操作硬件引脚。

这里给出一个示例:

```
#include <rtthread.h>
#include <board.h>

#define PIN_NAME "P0"  // 假设我们要操作的引脚名称为 "P0"

void example_rt_pin_get_usage(void)
{
    rt_base_t pin_handle;

    // 获取引脚对象
    pin_handle = rt_pin_get(PIN_NAME);

    // 判断返回值是否有效
    if (pin_handle < 0)
```

```
    {
        // 如果返回值小于0，表示出错
        rt_kprintf("Failed to get pin: %d\n", pin_handle);
        return;
    }

    // 如果返回值有效，执行其他操作，例如读取引脚的电平状态
    int pin_value = rt_pin_read(pin_handle);//通过读取到的句柄来控制引脚

    // 打印读取到的引脚电平
    if (pin_value == PIN_HIGH)
    {
        rt_kprintf("Pin %s is HIGH\n", PIN_NAME);
    }
    else
    {
        rt_kprintf("Pin %s is LOW\n", PIN_NAME);
    }
}

int main(void)
{
    // 调用函数执行引脚操作
    example_rt_pin_get_usage();

    return 0;
}
```

2. 延时函数 与 日志输出函数

---

1. 延时函数

```
rt_err_t rt_thread_mdelay(rt_int32_t ms)
{
    rt_tick_t tick;
```

```
    tick = rt_tick_from_millisecond(ms);

    return rt_thread_sleep(tick);

}
```

rt_thread_mdelay`函数解析

`rt_thread_mdelay` 是 RT-Thread 操作系统中的一个函数，用于使当前线程延时指定的毫秒数。该函数内部通过将毫秒数转换为系统时钟滴答数（tick）来进行延时。

函数原型：

```
rt_err_t rt_thread_mdelay(rt_int32_t ms);
```

参数：

- `ms`：要延时的毫秒数。

返回值：

- 返回值为 `rt_err_t` 类型，表示操作的结果。通常情况下，返回 `RT_EOK` 表示操作成功。

重点:之前使用Keil开发时,常使用hal_delay来满足简单的延时需求,但换用RT-Thread操作系统后,使用更高级的延时函数,既能实现简单的延时功能,也能做到多个线程间的切换

`hal_delay` 与 `rt_thread_mdelay` 的区别

`hal_delay` 通常是硬件抽象层（HAL）中的延时函数，它是一个基于忙等待的延时方法，而 `rt_thread_mdelay` 是 RT-Thread 操作系统中的延时函数，通常基于线程调度机制。

1. 延时机制：

- `hal_delay`：
    - 基于忙等待（busy-waiting）来实现延时。
    - 在执行 `hal_delay` 时，CPU 会一直处于忙碌状态，执行一个空循环来浪费时间，直到指定的时间过去。
    - 由于没有线程调度，`hal_delay` 在延时过程中会一直占用 CPU 资源，其他线程或任务无法获得 CPU 执行时间。
- `rt_thread_mdelay`：
    - 使用 RT-Thread 的线程调度机制来实现延时。
    - 在执行 `rt_thread_mdelay` 时，当前线程会被挂起，直到延时时间（转换后的 tick）到期。
    - 延时过程中，RT-Thread 会让出 CPU 资源给其他线程或任务。这样，其他线程可以并行执行，而当前线程在延时过程中不占用 CPU 资源。

2. 效率与资源占用：

- `hal_delay`：
    - 会消耗 CPU 时间，不允许其他线程在延时期间执行。适用于需要精确控制时间的低级硬件延时，但在多任务环境中，它是效率较低的方式。
    - 一般只在没有操作系统或者简单的裸机系统中使用，或者在不需要任务调度的场景下使用。
- `rt_thread_mdelay`：
    - 通过操作系统的调度机制实现延时，可以让 CPU 资源在延时期间被其他线程利用。适用于多任务操作系统，延时过程中不会浪费 CPU 时间。

3. 使用场景：

- `hal_delay`：
    - 用于硬件编程或者裸机系统，通常需要与硬件紧密交互，延时期间不关心其他任务的执行。
    - 适用于需要精确延时的情况，如一些特定硬件操作、外设的时序控制等。
- `rt_thread_mdelay`：

- 用于 RT-Thread 操作系统中，适用于多任务并发的情况。延时过程中，RT-Thread 操作系统能够调度其他线程执行，提高系统的并发性能。
- 更适合于多任务系统，或者需要系统调度的应用场景。

4. 精度：

- `hal_delay`：
  - 延时的精度通常取决于内部实现，可能会受到 CPU 时钟频率和延时函数的实现方式的影响。精度较差时会导致不精确的延时。
  - 在一些简单的裸机系统中，通常使用非常粗略的延时。
- `rt_thread_mdelay`：
  - 延时的精度受系统时钟（tick）的影响。如果系统时钟的精度较高（如 1ms），那么延时的精度也会相对较高。

总结

- `hal_delay` 是基于忙等待的延时方式，占用 CPU 时间，并且适用于没有操作系统或简单的嵌入式系统。
- `rt_thread_mdelay` 是基于操作系统调度机制的延时方式，使用系统时钟滴答来控制延时，不占用 CPU 时间，更适合在多任务操作系统（如 RT-Thread）中使用。

在多任务操作系统中，建议使用 `rt_thread_mdelay`，因为它不会阻塞其他任务的执行，而 `hal_delay` 适用于较为简单的系统

2.日志输出函数

```
#if (DBG_LEVEL >= DBG_LOG)
#define LOG_D(fmt, ...)      dbg_log_line("D", 0, fmt, ##__VA_ARGS__)
#else
#define LOG_D(...)
#endif

#if (DBG_LEVEL >= DBG_INFO)
#define LOG_I(fmt, ...)      dbg_log_line("I", 32, fmt, ##__VA_ARGS__)
#else
#define LOG_I(...)
```

```
#endif

#if (DBG_LEVEL >= DBG_WARNING)
#define LOG_W(fmt, ...)      dbg_log_line("W", 33, fmt, ##__VA_ARGS__)
#else
#define LOG_W(...)
#endif

#if (DBG_LEVEL >= DBG_ERROR)
#define LOG_E(fmt, ...)      dbg_log_line("E", 31, fmt, ##__VA_ARGS__)
#else
#define LOG_E(...)
#endif
```

DBG_LEVEL：这是一个预定义的调试级别，它决定了哪些日志级别的宏被激活。不同的级别可能包括 `DBG_LOG`（调试日志）、`DBG_INFO`（信息日志）、`DBG_WARNING`（警告日志）和 `DBG_ERROR`（错误日志）。

dbg_log_line：这是一个假设的函数，负责打印日志。它接受一个日志级别（如"D"、"I"、"W"、"E"），一个颜色代码和一个格式化的日志信息。

条件编译：通过 `#if` 语句，根据设置的 `DBG_LEVEL` 决定是否启用相应的日志宏。如果日志级别大于等于当前宏所对应的级别，则启用该日志宏；否则，宏什么都不做

日志输出影响:

日志输出等级（Log Level）对日志的输出有着非常重要的影响，它决定了在不同的开发和运行环境下，哪些信息会被记录和显示。通过设置不同的日志等级，开发人员可以控制输出的日志内容的详细程度，避免在生产环境中输出过多的调试信息，同时也能确保在调试阶段获得足够的日志信息来定位问题。

以下是常见的日志等级及其影响：

1. DEBUG（调试日志）

- **级别最低**：`DEBUG`是日志级别中的最低等级，通常用于开发和调试阶段。

- **输出内容**：该级别的日志包含了最详细的信息，包括程序执行的每一个细节、变量的值、函数的调用等。它对于开发人员在调试时非常有用，但如果在生产环境中启用，可能会产生大量的日志信息，导致性能下降。
- **影响**：通常只在开发和调试时启用。它有助于深入理解程序的执行过程，但会增加程序的日志输出量。

示例：

```
LOG_D("Variable x = %d", x);
```

2. INFO（信息日志）

- **一般性信息**：`INFO`日志用来记录程序的常规操作信息，通常包括系统的启动、关键模块的初始化、成功执行的操作等。它的目的是提供程序的运行情况，但不涉及详细的调试信息。
- **输出内容**：该级别的日志包含了对程序运行有意义的信息，但不包括调试级别的细节。
- **影响**：`INFO`日志在生产环境中通常启用，能够提供系统运行的概览而不会过多干扰。

示例：

```
LOG_I("System initialization complete.");
```

3. WARNING（警告日志）

- **潜在问题提示**：`WARNING`日志用于记录潜在的、不影响程序运行的异常或可疑情况。例如，系统可能已经遇到了一个问题，但它不会导致程序崩溃或立即停止运行。
- **输出内容**：该级别的日志用于提醒开发者或用户系统中存在某些需要关注的情况。虽然这些情况不致命，但仍可能影响系统的稳定性或性能。
- **影响**：`WARNING`日志通常在生产环境中启用，可以用来追踪系统中潜在的故障或性能问题。

示例：

```
LOG_W("Low memory detected, performance may be affected.");
```

4. ERROR（错误日志）

- **程序异常**：`ERROR`日志用于记录程序运行中的错误和异常情况，通常这些问题会导致程序的某些功能无法正常工作，或者程序可能无法继续运行。
- **输出内容**：该级别的日志包含了严重的错误信息，比如数据库连接失败、文件读取失败、关键功能崩溃等。
- **影响**：`ERROR`日志通常在生产环境中启用，因为它有助于快速识别并修复系统中的严重问题。

示例：

```
LOG_E("File %s not found, cannot continue execution.", filename);
```

5. CRITICAL（严重错误日志）

- **系统崩溃**：某些日志系统还会提供一个更高等级的日志，比如`CRITICAL`，用于记录那些会导致程序完全崩溃的致命错误。
- **输出内容**：该级别的日志通常用于记录程序中最严重的错误，可能会导致程序停止运行或崩溃。
- **影响**：`CRITICAL`级别的日志通常在生产环境中只会在出现非常严重的系统故障时输出，通常会引发报警或系统中断。

示例：

```
LOG_C("System failure! Unrecoverable error encountered.");
```

总结：

> *日志等级决定了输出日志的内容和频率，它直接影响日志的详细程度、系统性能和存储占用。通过合理设置日志等级，开发者可以在不同的环境（如开发、测试和生产环境）中调整日志输出，既保证了足够的信息用于调试和监控，又避免了不必要的性能开销。在生产环境中，通常会设置较高的日志等级（如`ERROR`、`WARNING`），而在开发阶段，则启用较低的日志等级（如`DEBUG`）以获取更多的调试信息。*

6.定义日志输出等级

```
// 定义调试标签为"main"。调试标签是用于标识日志输出的来源模块或部分，
// 例如在多模块的程序中，可以根据标签来区分日志来自哪个模块。
// 这里的"main"通常表示程序的主模块或入口函数。
#define DBG_TAG "main"

// 定义调试级别为DBG_LOG。
// DBG_LVL指定了调试的级别，它会决定日志的输出范围。
// DBG_LOG是一个常见的调试级别，表示“调试”级别的日志。
// 通过设置DBG_LVL为DBG_LOG，表示调试过程中将输出调试级别的日志。
#define DBG_LVL DBG_LOG

// 包含调试库的头文件，rtdbg.h是RT-Thread实时操作系统中的调试功能库。
// 这个库提供了丰富的调试功能，包括日志输出、调试信息输出、断言检查等。
// 通过包含这个头文件，可以使用RT-Thread的调试API，如`rt_kprintf`、`rtdbg`等来输出日志信息。
#include <rtdbg.h>
```

3.利用上述函数实现led灯闪烁功能

```
#include <rtthread.h>
#include <rtdevice.h>
#include <board.h>

#define DBG_TAG "main"
#define DBG_LVL DBG_LOG
#include <rtdbg.h>

/* 配置 LED 灯引脚 */
#define PIN_LED_B               GET_PIN(F, 11)      // PF11 :  LED_B
     --> LED
#define PIN_LED_R               GET_PIN(F, 12)      // PF12 :  LED_R
     --> LED
```

```c
/* 定义 LED 亮灭电平 */
#define LED_ON  (0)
#define LED_OFF (1)

/* 定义 8 组 LED 闪灯表，其顺序为 R B */
static const rt_uint8_t _blink_tab[][2] =
{
    {LED_ON, LED_ON},
    {LED_OFF, LED_ON},
    {LED_ON, LED_OFF},
    {LED_ON, LED_ON},
    {LED_OFF, LED_OFF},
    {LED_ON, LED_OFF},
    {LED_OFF, LED_ON},
    {LED_OFF, LED_OFF},
};

int main(void)
{
    unsigned int count = 0;
    unsigned int group_num = sizeof(_blink_tab)/sizeof(_blink_tab[0]);
    unsigned int group_current;

    /* 设置 RGB 灯引脚为输出模式 */
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
    rt_pin_mode(PIN_LED_B, PIN_MODE_OUTPUT);
    rt_pin_write(PIN_LED_R, LED_OFF);
    rt_pin_write(PIN_LED_B, LED_OFF);

    do
    {
        /* 获得组编号 */
        group_current = count % group_num;

        /* 控制 RGB 灯 */
        rt_pin_write(PIN_LED_R, _blink_tab[group_current][0]);
        rt_pin_write(PIN_LED_B, _blink_tab[group_current][1]);

        /* 输出 LOG 信息 */
```

```
        LOG_D("group: %d | red led [%-3.3s] | | blue led [%-3.3s]",
            group_current,
            _blink_tab[group_current][0] == LED_ON ? "ON" : "OFF",
            _blink_tab[group_current][1] == LED_ON ? "ON" : "OFF");

        count++;

        /* 延时一段时间 */
        rt_thread_mdelay(500);
    }while(count>0);
    return 0;
}
```