1

FR309x 系统编程说明书

Version:v1.0

Date: 2023.12

FREQCHIP
富芮坤

修订版本

版本	日期	更新内容
V1.0	2023.12.19	首版

目录

1. 系统概述

1.1. 概述

在开始 FR309x 的 trunkey 框架编程开发之前，首先需要对实时操作系统（RTOS）有一些基础的了解，因为该框架采用了 FreeRTOS 作为其实时操作系统。下面将介绍一些 RTOS 开发的基础知识，这将有助于初步了解并能够更轻松地着手进行开发。

1.2. 裸机与 RTOS 区别

1.2.1. 裸机概述

在了解 RTOS 之前，我们先回顾一下逻辑的编程框架。裸机编程采用的是死循环系统，一般也被称为前后台系统。在这种模型中，应用程序被设计成一个无限循环，循环中调用相应的函数来完成各种操作，这一部分可以被视为后台行为，或者称之为任务级。同时中断服务程序负责处理异步事件，这一部分可以被视为前台行为，也叫做中断级。在裸机开发中，前台行为与后台行为之间通过中断来实现异步事件的处理。这死循环的结构使得系统能够简单而高效地响应事件，同时执行应用程序的核心逻辑。因此，后台（任务级）负责处理顺序性的任务，而前台（中断级）则负责处理异步和实时性要求较高的事件，两者相互协作构成了整个逻辑系统的基本框架，如下图所示。

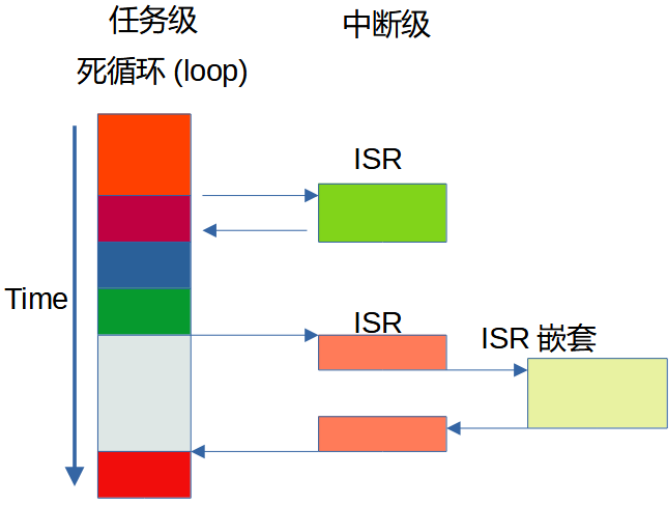

图 1-1 编程系统基本框架图

对于前后台系统的编程思路主要有以下两种方式：

1) 查询方式

在一些简单的应用中，处理器可以通过查询数据或消息是否就绪，一旦就绪就进行处理，然后再等待，如此循环。这对于简单的任务来说是一种简单易处理的方式。然而，在大多数情况下，我们需要处理多个接口的数据或消息，这就要求进行多次处理。下面的流程图展示了这个过程：

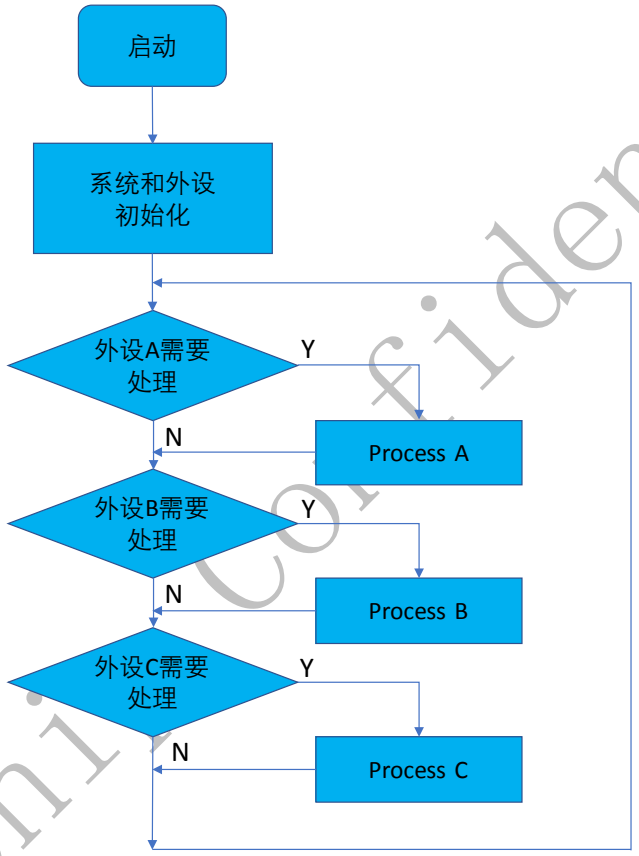

图 1-2 编程流程图

2) 中断方式

3) 在处理那些查询方式难以有效执行紧急任务的情况下，采用中断方式能够有效地解决这个问题。中断方式通过允许系统在出现紧急事件时立即中断当前任务的执行，转而处理紧急事件，从而避免了等待时间过长的问题。下面是中断方式的简单流程图：

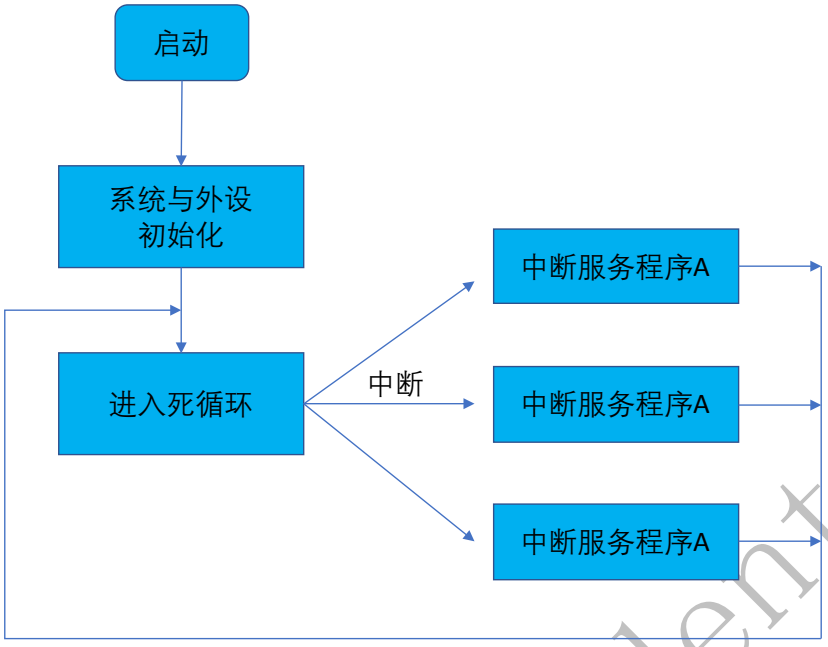

图 1-3 中断处理流程图

1.2.2. RTOS 概述

多任务系统的核心在于任务的调度和管理，通过引入实时操作系统（RTOS）来处理各个任务的执行。相比裸机方式，多任务系统的优势在于：

任务调度与管理：RTOS 负责有效地调度各个任务，使用不同的调度算法确保任务能够按照优先级和时间要求进行执行。

中断处理：中断处理在 RTOS 中得到更好的组织，通过任务和中断服务例程的协同工作，解决了裸机方式中 ISR 复杂性和嵌套带来的不可预测性问题。

数据共享：多任务系统通过更为安全和可控的方式实现任务间的数据共享，避免了全局共享变量可能带来的数据不一致性问题。

时间同步：RTOS 可以更灵活地处理系统计时器，适应多种不同的周期时间需求，同时对超过任务周期的耗时函数进行有效拆分，减小软件开销，提高应用程序可理解性。

系统扩展：多任务系统使得应用程序更为模块化，易于扩展。对于变更的影响更为可控，减少了可能产生不可预测副作用的风险。

RTOS 的优势：通过引入 RTOS，解决了裸机方式中存在的问题，提高了系统的可维护性、稳定性和实时性。

同时在多任务系统中，任务可以并发执行，通过 RTOS 的调度器决定哪个任务在何时执行，从而更好地满足复杂工程的需求。

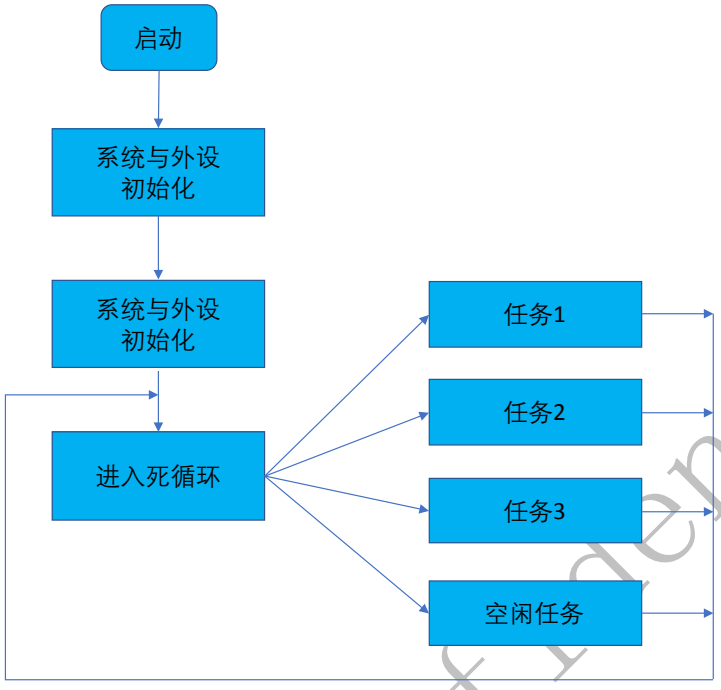

图 1-4 RTOS 系统任务处理流程图

在多任务系统 RTOS 的实现中，调度器扮演着关键的角色，其主要职责是利用相应的调度算法来决定当前需要执行的任务。正如上图所示，一旦任务被创建并完成了操作系统的初始化，通过调度器就能够在某一时刻确定应当运行的任务，如任务 1、任务 2、任务 3 以及空闲任务等，从而实现多任务系统的运行。

需要注意的是，在同一时刻，系统总是只能有一个任务在执行，也就是只有一个任务处于运行态。

调度器的决策使得看起来所有任务都在同时运行。为了更好地说明这一点，可以通过详细的运行例子来阐述这些任务的运行过程，执行流程图如下。

运行条件：

1. 每个任务都赋予了一个优先级
2. 每个任务都可以存在于一个或多个状态
3. 在任何时候都只有一个任务可以处于运行状态
4. 调度器总是在所有处于就绪态的任务中选择具有最高优先级的任务来执行
5. 优先级抢占式调度”我们这里主要讲这种方式

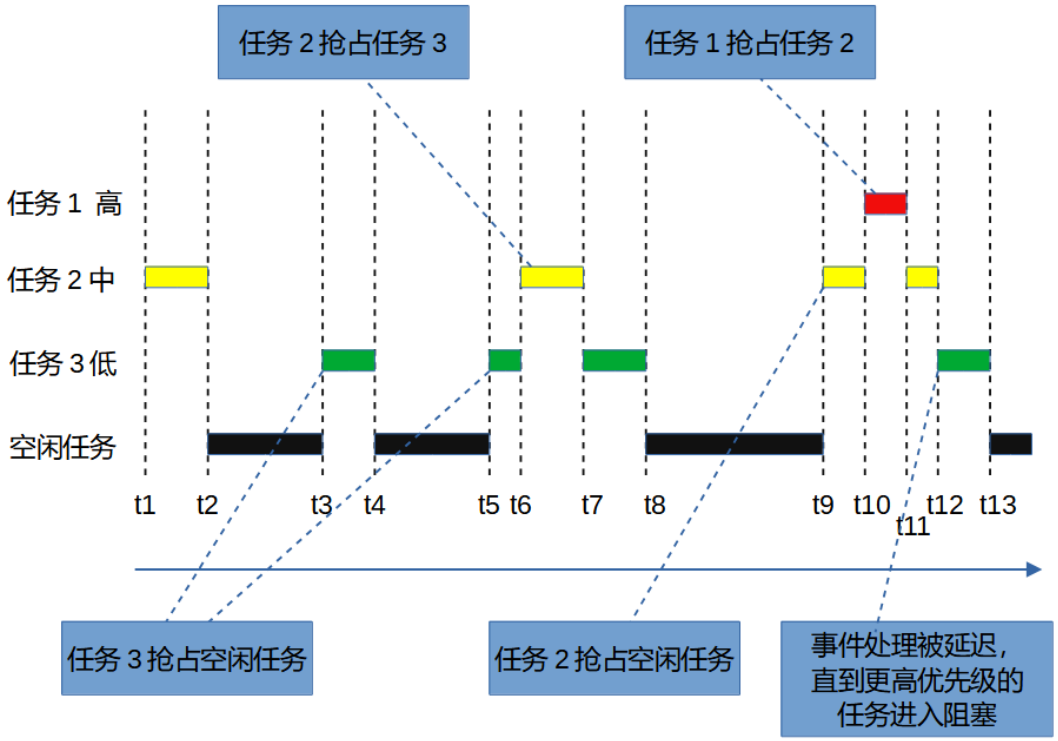

图 1-5 执行流程图

a) **空闲任务**

空闲任务被赋予最低的优先级，在更高优先级任务处于就绪态时，空闲任务会被抢占，如在图中的 t3、t5 和 t9 时刻。这确保了系统在有更高优先级任务需要执行时，空闲任务不会占用处理器资源。

b) **任务 3**

任务 3 是一个普通事件任务，其优先级相对较低，但高于空闲任务。大部分时间，任务 3 处于阻塞态，等待其关心的事件。事件的发生可以立即唤醒任务，从而使得任务 3 在 t3、t5 和 t9 至 t12 之间的某个时刻得以执行。在 t3 和 t5 时刻，由于任务 3 优先级最高，事件可以被立即处理。然而 t9 至 t12 时刻的事件只有在任务 1 和任务 2 进入阻塞态后才能得到处理。

c) **任务 2**

任务 2 是一个周期性任务，其优先级高于任务 3 但低于任务 1。根据周期间隔，任务 2 预期在 t1、t6 和 t9 时刻执行。在 t6 时刻，任务 3 正在运行，但由于任务 2 优先级较高，它抢占了任务 3 并立即执行。任务 2 在完成处理后返回阻塞态，任务 3 重新进入运行态，并在 t8 时刻进入阻塞状态。

d) 任务 1

任务 1 是一个事件驱动任务，拥有最高优先级，可以抢占系统中的任何其他任务。在图中，任务 1 的事件只在 t10 时刻发生，此时它抢占了任务 2。直到 t11 时刻，任务 1 再次进入阻塞态，任务 2 才有机会继续执行。

e) 任务优先级

从上图中可以观察到，任务的优先级分配如何根本性地影响应用程序的行为。一般而言，具有完成硬实时功能的任务被分配较高的优先级，而完成软实时功能的任务则被分配较低的优先级。然而，还需要考虑其他因素，如执行时间和处理器利用率，以确保应用程序不会超过硬实时需求的限制。因此，任务的优先级选择需要综合考虑任务的实时性要求、执行时间、以及系统的可调度性。单调速率调度等方法提供了一种有用的方式，但在实际应用中，需要根据具体情况综合考虑不同因素，以确保系统能够满足各种任务的需求。

f) FreeRTOS 常用 api 介绍

参考官方开发者文档学习相关的 api 使用，链接：FreeRTOSFeatures - FreeRTOS https://www.freertos.org/features.html

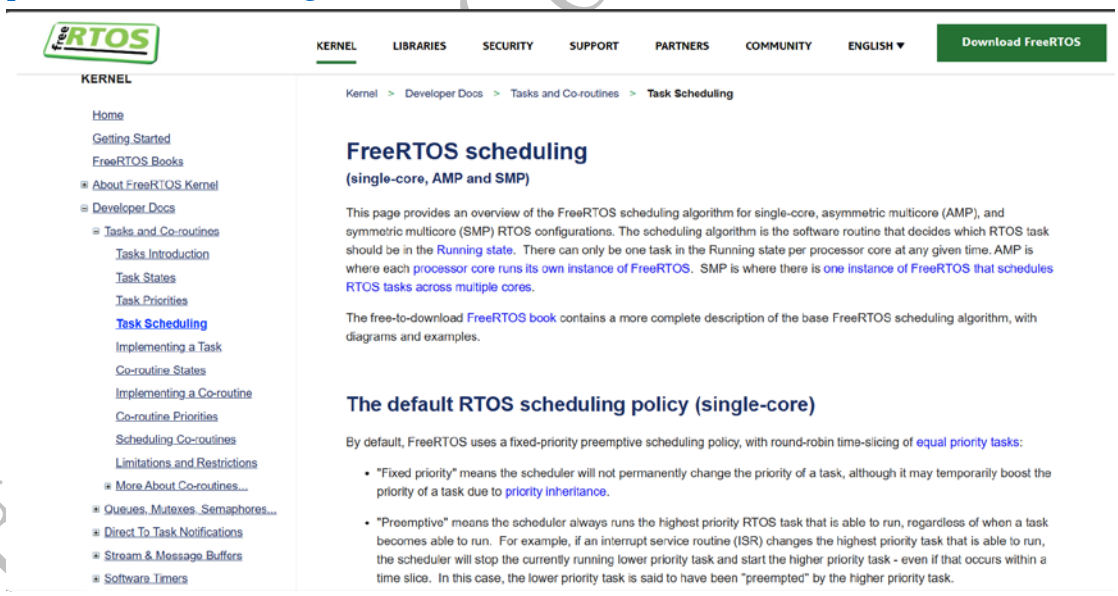

图 1-6 官方开发者文档资料

2. 系统任务管理

通过以上介绍对 RTOS 有个基本的认识，现在我们介绍目前整个 trunkey 框架中 RTOS 相关任务，目前框架中创建了 5 个任务，分别是 app_task，rpmsg,btdm_stack，lvgl，monitor 任务，用户不要随意修改这几个任务的相关的配置，如优先级任务栈大小等。

1) app_task 任务

任务中负责初始化一些外设，循环中任务进入阻塞态等待其通知值，然后处理应用层的事件，比如外设中断服务函数中一些标志和数据通过事件传递到应用任务中去处理，不要在中断里面去处理耗时长的操作，当然用户外加其他的应用的事件都可以加到这个任务中去处理。

```
static void app_task(void *arg)
{
    struct app_task_event *event;

    printf("app_task\r\n");

#if ENABLE_DSP == 1…
#else
    event = app_task_event_alloc(APP_TASK_EVENT_RPMSG_INITED, 0, true);
    app_task_event_post(event, false);
#endif
    device_pa_init();
    device_charge_init();

#if BOARD_SEL == BOARD_EVB_FR5090
    button_init(BUTTON_PIN_NUM|SOS_KEY_PIN_NUM|KEY1_PIN_NUM);
#elif BOARD_SEL == BOARD_EVB_FR3092E
    button_init(BUTTON_PIN_NUM|SOS_KEY_PIN_NUM);
#elif BOARD_SEL == BOARD_EVB_FR3092E_CM
    button_init(BUTTON_PIN_NUM|SOS_KEY_PIN_NUM|KEY1_PIN_NUM);
#else
#error "choose correct board"
#endif

    device_pmu_io_init();

    dev_motor_init();

    while(1) {
        ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
        app_task_event_handler();
    }
}

void app_task_init(void)
{
    xTaskCreate(app_task, "app", APP_TASK_STACK_SIZE, NULL, APP_TASK_PRIORITY, &app_task_handle);
}
```

图 2-1 app_task 函数示例

事件处理函数里面实现具体逻辑，例如 at 指令处理，rpmsg 初始化，host 初始化等事件，函数进来首先挂起调度器，从事件链表里面获取事件值，然后恢复调度器（这里主要为了防止从链表中取数据过程中有其他高优先级任务抢占又往链表中插入数据，破坏链表中数据），再判断当前事件类型，读取事件参数值，调用对应的函数处理。

```c
static void app_task_event_handler(void)
{
    struct app_task_event *event = NULL;
    vTaskSuspendAll();
    event = (struct app_task_event *)co_list_pop_front(&event_list);
    xTaskResumeAll();
    //printf("%s %x\r\n",__func__, (uint32_t)&event);

    if(event) {
        switch(event->event_type) {
            case APP_TASK_EVENT_AT_CMD:
                app_at_cmd_recv_handler(event->param, event->param_len);
                break;
            case APP_TASK_EVENT_RPMSG_INITED:
#if BTDM_STACK_ENABLE == 1
                app_btdm_init();
                break;
#endif
            case APP_TASK_EVENT_HOST_INITED:
                app_lvgl_init();
//                break;
#if BTDM_STACK_ENABLE == 1
            case APP_TASK_EVENT_LVGL_INITED:
                app_btdm_start();
#endif
                break;

            case APP_TASK_EVENT_BTN_TOGGLE:
                {
                    button_toggle_handler(*(uint32_t *)event->param);
                }
                break;

            case APP_TASK_EVENT_BTN_OUTPUT:
                {
                    button_event_handler(event->param);
                }
```

图 2-2 事件处理函数示例

当我们要往应用任务里面发送事件通知时，通过 app_task_event_post 接口进行发送，该函数已经做了临界保护，任务中和中断服务函数里面都可以调用，调用函数首先将事件值插入到链表中，然后向 app_task 发送任务通知，appt 任务将会从阻塞态切换到运行态执行相关处理。

拿我们编码器的 io 状态处理为例，在 pmu 中断服务函数里面将当前的状态，通过 app_task_event_post 发送到应用任务中去处理。

```
void app_task_event_post(struct app_task_event *event, bool high)
{
    uint32_t old_basepri;

    if(xPortIsInsideInterrupt()) {
        old_basepri = taskENTER_CRITICAL_FROM_ISR();
        if(high) {
            co_list_push_front(&event_list, &event->hdr);
        }
        else {
            co_list_push_back(&event_list, &event->hdr);
        }
        taskEXIT_CRITICAL_FROM_ISR(old_basepri);
        vTaskNotifyGiveFromISR(app_task_handle, NULL);
    }
    else {
        taskENTER_CRITICAL();
        if(high) {
            co_list_push_front(&event_list, &event->hdr);
        }
        else {
            co_list_push_back(&event_list, &event->hdr);
        }
        taskEXIT_CRITICAL();
        xTaskNotifyGive(app_task_handle);
    }
}
```

图 2-3 app_task_event_post 函数示例

```
void PMU_GPIO_PMU_IRQHandler(void)
{
    uint16_t data = ool_read16(PMU_REG_PIN_DATA);
    uint16_t result = ool_read16(PMU_REG_PIN_XOR_RESULT);
    /* update last value with latest data */
    ool_write16(PMU_REG_PIN_LAST_V, data);
    /* clear last XOR result */
    ool_write16(PMU_REG_PIN_XOR_CLR, result);
//    printf("PMU IO: 0x%04x, 0x%04x\r\n", data, result);
    if (data & PMU_PIN_9) {
        system_prevent_sleep_clear(SYSTEM_PREVENT_SLEEP_TYPE_HCI_RX);
    }
    else {
        system_prevent_sleep_set(SYSTEM_PREVENT_SLEEP_TYPE_HCI_RX);
    }
    if (result & (ENCONDED_A_PIN_NUM | ENCONDED_B_PIN_NUM)) {
        uint8_t encode_bit_status;
        if (!(data & ENCONDED_A_PIN_NUM))···
        else···

        if (!(data & ENCONDED_B_PIN_NUM))···
        else···

        encode_toggle_detected(encode_bit_status);
    }

    if (result & (BUTTON_PIN_NUM | SOS_KEY_PIN_NUM | KEY1_PIN_NUM)) {···
}
```

图 2-4 PMU_GPIO_PMU_IRQHandler 函数示例

```
void encode_toggle_detected(uint8_t curr_encode)
{
    //printf("encode %d \n",curr_encode);
    if(encode_bit_status != curr_encode)
    {
        encode_bit_status = curr_encode;
        encode_push_key(curr_encode);
        struct app_task_event *event;
        event = app_task_event_alloc(APP_TASK_EVENT_ENCODE_TOGGLE, sizeof(uint8_t), false);
        if(event)
        {
            memcpy(event->param, (void *)&curr_encode, sizeof(uint8_t));
            event->param_len = sizeof(uint8_t);
            app_task_event_post(event, false);
        }

    }
}
```

图 2-5 encode_toggle_detected 函数示例

2) rpmsg 任务：

负责处理与 dsp 通信相关的任务，用户无需关心，不用修改这个任务相关配置。

3) btdm_stack 任务：

蓝牙协议栈相关处理任务，用户无需关心内部逻辑，默认不用修改任务配置。

4) lvgl 任务：

gui 处理任务，lvgl 的界面的渲染和显示相关等，默认不用修改任务相关配置。

5) monitor 任务：

该任务主要处理监视 RTOS 相关任务的运行状态打印输出，调试时可以打开相关的 log 看任务状态,同时也负责看门狗的喂狗，RTC 时间刷新等。

6) 任务间通信

当我们 app 应用任务/蓝牙任务接收数据中的数据要传递给 gui 任务显示时，需要通过这个 gui_task_msg_send 接口进行发送，这个函数接口内部发送的数据会通过 FreeRTOS 的消息队列进行发送，gui 任务中会接收当前消息进行处理以及更新到显示界面，消息类型需要自己添加定义。

注意：调用这个接口一定要等到 gui_task 初始化完运行后才能使用。

```
int gui_task_msg_send(uint16_t msg_type,
                      void *header,
                      uint16_t header_length,
                      uint8_t *payload,
                      uint16_t payload_length,
                      ipc_tx_callback callback)
{
    gui_task_msg_t queue_event;
    queue_event.msg_type = msg_type;
    queue_event.param_len = header_length+payload_length;
    memcpy(queue_event.param, header, header_length);
    memcpy(queue_event.param+header_length, payload, payload_length);
    //if (portNVIC_INT_CTRL_REG & 0xff)
    if(xPortIsInsideInterrupt())
    {
        BaseType_t xTaskWokenByPost = pdFALSE;
        if (xQueueSendFromISR(gui_queue_handle, &queue_event, &xTaskWokenByPost) ==  errQUEUE_FULL )
        {
            return -1;
        }
        else
        {
            portYIELD_FROM_ISR(xTaskWokenByPost);
        }
    }
    else
    {
        if (xQueueSend(gui_queue_handle, &queue_event, ( TickType_t ) 0) != pdPASS)
        {
            return -1;
        }
    }
    return 0;
}
```

图 2-6 gui_task_msg_send 函数示例

```c
//gui task
static void gui_task(void *arg)
{
    gui_task_msg_t queue_event;
    void lfs_custom_init(void)
    lfs_custom_init();
    //lfs_custom_test_write_file();
    dev_rtc_time_init();
    lvgl_init();
    // Create  queue for gui task
    gui_queue_handle = xQueueCreate(GUI_TASK_QUEUE_LENGTH, sizeof(gui_task_msg_t));
    //battery detect
    adc_vbat_start_detect();
    setup_gui_running();
    while(1) {
        vTaskDelay(1);
        lv_timer_handler();
        rtc_running();
        encode_key_release();

        if (xQueueReceive(gui_queue_handle, &queue_event, 0) == pdPASS)
        {
            gui_task_queue_callback(&queue_event);
        }

        gui_task_auto_suspend();

    }
}
```

图 2-7 gui_task 函数示例

gui_task_queue_callback 中根据消息类型将对应参数传递给 lvgl 相关界面的 obj 对象。这里通过也是通过事件方式发送。会触发相应的回调函数里面去处理具体逻辑。

```c
void gui_task_queue_callback(gui_task_msg_t * event)
{
    switch(event->msg_type)
    {
        case BUTTON_KEY_EVT:
            {…
            break;
        case ENCODE_KEY_EVT:
            {…
            break;
        case BUTTON_KEY1_EVT:
            {…
            break;
        case BUTTON_KEY2_EVT:
            {                …
            break;
        case MESSAGE_IN_EVT:
            {
                if(lv_obj_is_valid(prj_parent_cont))
                {
                    lv_event_send(prj_parent_cont, LV_EVENT_ANCS_MSG_IN, NULL);
                }
            }
            break;
        case PHONE_CALL_IN_EVT:
            {
                uint8_t call_mode = user_get_call_mode();
                if(lv_obj_is_valid(prj_parent_cont))
                {
                    if(call_mode!=5)
                    {
                        lv_event_send(prj_parent_cont, LV_EVENT_CALL_IN, NULL);
                    }else{
```

图 2-8 gui_task_queue_callback 函数示例

3. 系统编程

3.1. 编程实例

3.1.1. 任务创建

通过上一节的介绍我们已经知道当前 sdk 框架中已经创建了部分任务，一般情况下能够满足大部分功能需求，当我们开发中有其他需求时。通常系统中将一个大的模块单独用一个任务来实现，比如我们要定时采集传感器的数据时，我们可以再创建一个任务去处理，先定义一个 TaskHandle_t 句柄，采用 xTaskCreate 来创建一个任务，例如：

```
static TaskHandle_t sensor_task_handle;

static void Sensor_task(void *arg)
{
    Sensor_driver_init();
    while(1) {
        vTaskDelay(2000);
        Sensor_handler();
    }
}

void Sensor_Init(void)
{
    xTaskCreate(Sensor_task, "Sensor_task", 256, NULL, (tskIDLE_PRIORITY+1), &sensor_task_handle);
}
```

图 3-1 创建任务示例

创建了一个任务单独去处理传感器数据的读取，这里注意一定要加 vTaskDelay 或加等待某些事件或消息通知的 api,让当前任务进入阻塞态让出 cpu 使用权给其他任务，不能一直占用，这个参数根据自己的实际情况进行调整，这里是等待指定个数的 Tick 中断才能变为就绪状态，当前系统中配置是 1ms 一个 Tick,当采集的数据要通过传递到 gui 任务显示时可以调用 gui_task_msg_send 发送，需要传递到 app 任务时采用 app_task_event_post 发送，如果需要往其他任务中传递数据也可参考这两个接口的实现方式来做。

3.1.2. 消息队列

消息队列是一种 RTOS 提供的服务接口，用于在任务或中断服务子程序之间传递数据。在 FreeRTOS 中，消息队列允许任务或中断服务子程序将消息放入队列，并且一个或多个任

务可以通过 RTOS 内核服务从队列中获取消息，具体实现参考 gui_task_msg_send 内部实现。

3.2. RTOS 编程注意事项

1) 减少全局变量的使用

避免多个任务使用共享一个全局变量的情况，如果必须用到的情况下需要加入互斥信号量进行资源保护，不然会出现问题，同理多个任务都会使用到的 api 函数接口内部实现时也需要加入互斥信号量进行保护，比如读写 flash 的接口。这里我们用一个例子来说明多个任务共享资源不做保护会引发的隐患：

a) 首先创建两个任务，task1，task2 两个任务里面都对全局变量 a1 进行自增操作。

b) 查看变量 a1++的反汇编代码，可以看出代码分 4 步运行,分别涉及到读改写。

```
0x0800163C 4C3C    LDR     r4,[pc,#240] ; @0x08001730
0x0800163E 6821    LDR     r1,[r4,#0x00]
0x08001640 1C49    ADDS    r1,r1,#1
0x08001642 6021    STR     r1,[r4,#0x00]
```

i. 先从内存中将 a1 的地址读取到 r4 寄存器中。

ii. 将 r4 指向的地址读取值到寄存器 r1 中，也就是变量 a1 的当前值。

iii. 将寄存器 r1 中的值加 1。

iv. 将寄存器 r1 中的新值写回到 r4 指向的地址，完成对 a1 变量的自增操作。

c) 可以看到完成整个操作是不止一条指令，操作过程可能被中断，也就是一个”非原子”操作。当两个任务都正常运行完一次 a1++时 a1 的值应该变成 3，但是考虑到这种情况，当 task1 执行过程中只执行完第 2 步，这时候 task2 高优先级任务打断了 task1 去执行 task2，那么保存现场时存在寄存器 r4 的值就还是 1，当 task2 运行完时回到运行 task1，这时恢复现场时读取任务栈保存的寄存器信息，r4 寄存器还是 1，继续执行 3，4 步骤，这样得到的结果 a1 的值将是 2，明显不满足我们预期的值这样就会出现问题。

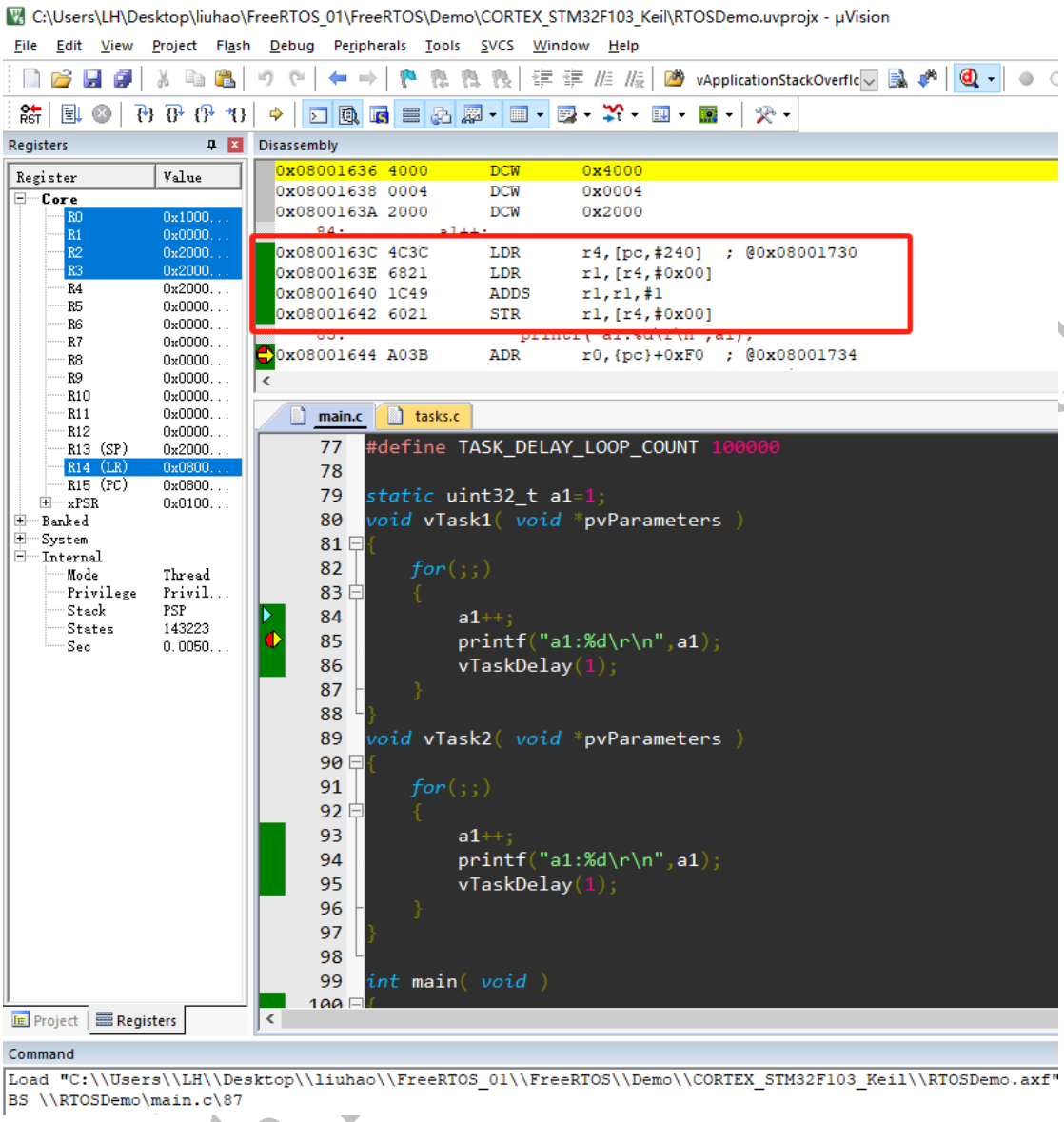

图 3-2 反汇编示例

因此在多任务系统中要访问一个被多任务共享，或是被多任务与中断共享的资源时，需要采用”互斥锁”以保证数据在任何时候都保持一致性，函数封装具体实现可以参考我们 sdk 工程中读写外部 flash 的接口封装，变量互斥访问保护如下图。当然也可以通过进入临界区进行保护，可以通过 taskENTER_CRITICAL()&taskEXIT_CRITICAL()进入退出,也可以使用 vTaskSuspendScheduler()&xTaskResumeScheduler()进入退出。

```
79  static uint32_t a1=1;
80  static SemaphoreHandle_t Test_Mutex;
81  void vTask1( void *pvParameters )
82  {
83      for(;;)
84      {
85          xSemaphoreTake(Test_Mutex, portMAX_DELAY);
86          a1++;
87          xSemaphoreGive(Test_Mutex);
88          printf("a1:%d\r\n",a1);
89          vTaskDelay(1);
90      }
91  }
92  void vTask2( void *pvParameters )
93  {
94      for(;;)
95      {
96          xSemaphoreTake(Test_Mutex, portMAX_DELAY);
97          a1++;
98          xSemaphoreGive(Test_Mutex);
99          printf("a1:%d\r\n",a1);
00          vTaskDelay(1);
01      }
02  }
```

2) 任务中与中断里面使用 RTOS 中 api 接口的区别

中断里面调用时要加 FromISR 结尾的函数，比如串口中断里面接收一组数据，要通过消息队列发送到任务里面，这时我们需要调用 xQueueSendFromISR，任务中调用 xQueueSend 即可。

3) 任务栈和 RTOS 系统栈大小的分配

RTOS 栈大小在 FreeRTOSConfig.h 文件中设置 configTOTAL_HEAP_SIZE 宏来配置这个根据当前工程中 ram 使用情况来调整不能设置太小，任务栈大小在创建任务时设置，任务栈是从 RTOS 系统栈里面进行动态分配的，计算任务栈大小的建议：将每一级函数嵌套调用时可能用到的栈空间大小累加，以得到一个粗略的栈大小估算。需要考虑函数局部变量、形参、返回地址以及内部状态保存等因素。当我们不好估算大小时可以将当前任务栈设置大一点，然后通过 vTaskList 函数运行时查看当前任务栈使用情况再进行调整。

4) RTOS 系统时基使用了 SYSTICK 定时器，应用层不要再使用。

5) 官方总结的问题汇总链接:http://www.freertos.org/FAQ.html

联系方式

欢迎大家针对富芮坤产品和文档提出建议。

反馈：doc@freqchip.com.

网站：www.freqchip.com

销售：sales@freqchip.com

电话：+86-21-5027-0080