

RAW-OS 实例教程

Raw-os 官网: www.Raw-os.org

作者: 樊文杰

审阅: jorya_txj

说 明

本文档作为 RAW-OS 的实例教程，重点在于演示如何使用 RAW-OS 内核提供的各种机制和服务，建议读者配合《高效实时操作系统设计》这本书的理论内容，结合模块化的程序来加深理解。

RAW-OS 的开发目标是做世界上最好的 RTOS，读者从官网上的 RAW-OS 特性就可以知道，RAW-OS 不是一款简单的重复别人的 RTOS，更不是作者 Just For Fun 的产物。其丰富的内核特计，以及创新的设计足以和目前任何主流商业 RTOS 可睥睨。

RAW-OS 在提供了 RTOS 最基本的特性和服务的同时，对其它特性进行扩展创新，精心设计出了独具特色的一套机制，比如：基于 Task_0 的最大关中断 0 us 特性，中断下半部特性(work_queue 工作队列)，基于 idle 任务的事件触发机制，支持多种情况应用的消息队列，支持 block,byte,page,malloc,slab 五种方式的内存管理等。相信这些特色机制的灵活运用能让你高效的完成应用程序设计。

本文档假定读者具有实时操作系统理论与基础知识，对于一些概念和原理，如有不懂请读者参阅《高效实时操作系统》书中所述。

对于本文档内容和例程，如有不懂，请到下面论坛提问：

正点原子论坛：<http://www.openedv.com/forums/show/35.htm>

EEWORLD 论坛：<http://bbs.eeworld.com.cn/forum-190-1.html>

或加入以下群进行交流：

Raw-os 官方 QQ 群: 147640063, 257989928

ALIENTEK 技术交流群: 333121886

1. 本教程中所采用的硬件和软件平台：

硬件平台：正点原子 STM32F4 探索者开发板
Raw-os 内核版本：V1.056
开发环境：KEIL 5.10

2. 例程文件目录结构与工程结构采用如下规则：



BSP 目录：存放外设配置，外围芯片驱动程序。

Libraries 目录：存放 ST 官方标准的固件库。

Project 目录：存放工程文件。

RAW-OS 目录：存放 RAW-OS 源码，下面的 port 目录是移植接口，Raw-os 目录是内核源码，Raw-os 目录下面又有 extension 目录存放 Raw-os 扩展功能的源码。

User 目录：用户程序，下面的 shell 子目录存放所有用户的 shell 命令实现。

Utilities 目录：存放 FATFS，LWIP 等第三方协议栈源码。

1. 任务创建

让我们从第一个函数开始，函数名为 `raw_task_create`，这个函数的参数比较多。下面一一解答：

```
// 创建启动任务,配置系统硬件,启动其它任务
raw_task_create(&Init_Task,           // 任务对像,一个任务所有属性的集合
                (RAW_U8 *) "Init_Task", // 任务的名子
                0,                     // 传给任务的参数,没有传入则为 0
                1,                     // 任务的优先级
                0,                     // 任务的时间片(为 0 则取默认值)
                Init_Task_Stack,       // 任务的栈基址
                INIT_TASK_STK_SIZE,    // 任务的栈大小
                Init_Task_Func,        // 任务的执行函数
                1);                    // 任务进入就绪态
```

看起来好多的样子，不过读者具有 RTOS 的基础知识，理解起来不是很难。

配合这个任务创建，读者还需要给一个任务定义任务对像，声明栈空间，定义它的执行函数入口。代码如下：

```
// 定义任务栈空间,任务对像

#define          INIT_TASK_STK_SIZE      256
PORT_STACK      Init_Task_Stack[INIT_TASK_STK_SIZE];
RAW_TASK_OBJ    Init_Task;

// 任务执行函数声明

void Init_Task_Func(void *parg);
```

OK，创建一个任务基本上就这些东西了，注意，上面代码是声明了任务执行函数，当然还需要实现它。完整代码参见例程 1。

关于第一个启动任务的设计

所有的 RTOS 设计有一个共通性是，在 RTOS 启动之前不允许产生任何硬件中断，因为那个时候系统还没开始运行任务，所以系统很显然就挂了。设计第一个启动任务的时候请遵循一点设计要求：

- 1 在第一个启动的最高优先级用户任务里面，初始化硬件中断相关的代码。
- 2 第一个启动的最高优先级任务初始化完成后可以继续当做其它任务继续使用，或者自己删除自己。
- 3 假设硬件中断中会使用内核 api，请先保证该内核 api 在初始化硬件中断时已经初始化完毕。

很显然时钟中断的初始化也应该是在第一个用户最高优先级任务里面去初始化的。

根据上面的理论，在 main 函数中只要创建一个高优先级的任务即可，然后启动系统，其它的工作都在这个高优先级任务中执行。

启动任务的执行函数源码：

```
// 启动任务的执行函数
void Init_Task_Func(void *parg)
{
    // 1. Initial all the used Hardware
    OS_CPU_SysTickInit();           // 配置系统时钟
    LED_Init();                     // LED 初始化

    // 2. Create other Task
    Task_Test();

    raw_task_delete(raw_task_identify()); // 删除自己
}
```

上面代码中 OS_CPU_SysTickInit()是配置 Raw-os 的工作时钟，具体就是配置 SysTick 的中断频率：

```
SysTick_Config(SystemCoreClock / RAW_TICKS_PER_SECOND);
```

其中 RAW_TICKS_PER_SECOND 在 raw_config.h 中配置，默认值为 100，表示 OS 的时钟周期为 10ms。

Task_Test()函数即创建两个任务，分别以不同的频率控制两个 LED 闪烁。

实验说明：

本实验简单易懂，即创建两个任务分别控制两个 LED 以不同的频率闪烁，源码注释已清楚，具体请读者自行熟悉。

同时本节例程也作为 STM32F4 平台的工程模板。

2. 时间片轮转调度

Raw-os 支持时间片轮转调度，多个任务可以设置相同的优先级，从理论上来说系统支持任意数量的任务。那同优先级的任务怎么调度呢，每个任务都有一个时间片，系统以这个时间片为间隔调度它。

从本节开始后面的实验都需要用到串口来打印输出结果，需要读者连接板上的 USART1(PA9, PA10 两个引脚)到电脑串口，波特率为 115200。

另外，本文档后面的例程，包括 shell，都需要在 SecureCRT 这个软件下使用，建立一个串口连接，使用方法请自行了解。

实验目的：演示时间片轮转调度的细节

实验思路：创建两个任务，具有相同的优先级，时间片为系统默认 50 个 tick，各打印 10 次语句，当任务的时间片到时，即被系统剥夺 CPU 使用权转而运行另一个任务，另一个任务的时间片到时再返回到第一个任务，根据打印结果查看调度细节。

实验关键代码：

```
// 第一个任务的函数入口
void First_Task(void * pParam)
{
    int i;
    for(i = 0; i < 10; i++)
    {
        LED_Toggle(1);      // 翻转 LED1
        USART1_Send_String("\r\nThis is First Task!\r\n");
    }
}

// 第二个任务的函数入口
void Second_Task(void * pParam)
{
    int i;
    for(i = 0; i < 10; i++)
    {
        LED_Toggle(2);      // 翻转 LED2
        USART1_Send_String("\r\nThis is Second Task!\r\n");
    }
}
```

实验结果： 在 Secure-CRT 中观察现象如下。

```

This is First Task!
This is First Task!
This is First Task!
This is First Task!
This is Fi
This is Second Task!
This is Second Task!
This is Second Task!
This is Second Task!
This is Second Task!
This is First Task!
This is First Task!
This is First Task!
This is First Task!
This is First Task!
This is Second Task!
This is Second Task!
This is Second Task!

```

1. 任务1执行到此处时其时间片到,系统执行任务2

2. 任务2执行到此处时其时间片到,系统接着执行刚才被打断的任务1

3. 任务1的时间片又到,系统接着执行任务2

从上面可以看出，两个函数各打印 10 次语句，两个任务都没有主动去放弃 CPU 使用权，但由于具有相同的优先级，那么当第一个任务在执行过程中其时间片到时，会被系统剥夺 CPU 控制权，转而运行第二个任务，第二个任务在执行过程中，其时间片到也同样会被剥夺 CPU 控制权，转而接着执行第一个任务。

需要注意的是，任务的时间片值是可被设置的，如果在任务创建时此项参数为 0 则为系统默认值，50 个 tick，当然，这个默认值在 raw_config.h 文件中也是可以配置的：

```

/*default task time slice*/
#define TIME_SLICE_DEFAULT 50

```

3. 信号量同步

本例用于演示按键中断与任务的同步，把板上四个按键配置为中断模式，每个按键按下时均会产生中断，创建一个信号量和两个任务，两个任务均阻塞在这个信号量上，按键按下时，产生中断，在中断中释放信号量，在退出中断的时候，执行中断级任务调度，两个阻塞的任务得以被执行，执行后会再次阻塞在信号量上。

实验目的：演示信号量用于同步的使用方法

实验思路：创建一个任务，阻塞在信号量上，当有按键按下时产生中断，在中断中释放信号量，退出中断时系统执行调度，唤醒该任务，任务执行后信号量被使用，所以任务重新阻塞在信号量上。

实验代码：见例程代码

实验结果：每次按键按下时你会发现 LED 进行多次翻转，按理说是按下一次，释放一次信号量，LED 只翻转一次的，那是什么原因呢？当然是因为按键没有进行消抖，你按下一次，事实上产生多次中断，释放多次信号量，从而任务执行了多次，所以 LED 也翻转多次。

怎么解决这个问题呢，在软件定时器的例程里将会设计稳定可靠的按键驱动程序。

注意：在 RAW-OS 系统平台下，中断服务程序的编写模板如下：

```
// 中断服务程序
void XXX_IRQHandler(void)
{
    raw_enter_interrupt();           // 1.进入中断

    XXXX                           // 2. 执行中断处理

    raw_finish_int();               // 3. 退出中断
}
```

`raw_enter_interrupt()` 和 `raw_finish_int()`是在中断服务程序的开始和退出处必须调用的函数，在 `raw_finish_int()`执行中断级任务调度，如果有更高优先级任务就绪，在中断执行完毕后不再回到刚才被中断的任务，而是去执行刚才就绪的更高优先级的任务。

请读者研究源码，明白信号量使用方法：

1. 创建一个信号量：`raw_semaphore_create`
2. 等待(阻塞)一个信号量：`raw_semaphore_get`
3. 释放一个信号量：`raw_semaphore_put`

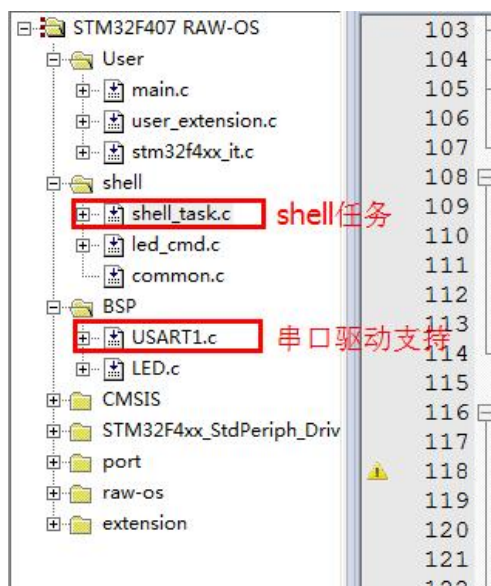
4. shell 详解

在开始本节之前，读者需要对前三个例程详细了解，读懂代码架构，明白任务创建的细节，信号量的同步原理等。

RAW-OS Shell 使用一种简单、易用、高效的机制使得应用程序支持命令行输入。系统默认支持一个“help”命令，如果有新的命令加入，则 RAW-OS 建立维护一个单向链表，新加入的命令依次挂 help 这个链表头后面，当在应用层面，比如串口终端中，输入命令时，执行 shell 解析的任务会读取这个命令，并通过 `rsh_process_command()` 函数从单链表头开始查找这个命令是否已注册（挂接到命令链表），每个命令会对应一个可执行函数，如果查找到该命令，则跳转到这个命令所对应的可执行函数中去执行。

在本例中，编写 UART 驱动，使其以中断方式支持串口输入的功能，同时，编写一个任务，阻塞在一个信号量上。每当用户在终端输入一个字符时，触发中断，然后在中断中释放这个信号量，中断退出后这个任务获理信号量得以调度，然后读取刚才输入的字符。当用户输入完一串字符命令后，输入回册，程序识别到回册符，则认为命令输入结束，从而对刚才输入的这一串字符进行解析，在系统的命令列表中进行比对，从而识别这个命令。

Shell 机制的实现需要依赖串口，所以在本例中，工程目录如下：



串口中断使用信号量同步 shell 任务的执行，具体实现，参见代码。

下面从加入一个自定义命令的角度来说明 Raw-os 的 shell 机制使用方法，当你用熟的时候，你会爱上 shell 的。

如果想详细研究 shell 的实现机制，请参见附录 2 RAW-OS Shell 机制分析。

1. 定义一个命令体及命令执行函数

首先分析下一个命令体所对应的数据结构类型，`rsh.h` 文件中定义了定义一

个命令所遵循的数据结构类型。

```
typedef struct xCOMMAND_LINE_INPUT
{
    const RAW_S8 * const pcCommand;
    const RAW_S8 * const pcHelpString;
    const pdCOMMAND_LINE_CALLBACK pxCommandInterpreter;
    RAW_S8 cExpectedNumberOfParameters;
} xCommandLineInput;
```

第一项 `pcCommand` 即为在执行应用时要输入的命令,它应是小写字母组成。

第二项 `pcHelpString` 是这个命令对应的使用说明。

第三项是命令行对应的回调函数,它实际上是一个函数指针,输入命令后,后跳转到该函数的地址处执行。

第四项是该命令支持的参数个数。

其中对于第三项函数指针 `pdCOMMAND_LINE_CALLBACK` 类型定义如下:

```
typedef RAW_S32 (*pdCOMMAND_LINE_CALLBACK)( RAW_S8 *pcWriteBuffer, size_t
xWriteBufferLen, const RAW_S8 * pcCommandString );
```

这是一个函数指针, `RAW_S32` 为函数返回类型,

`pdCOMMAND_LINE_CALLBACK` 为指针变量名。它有三个参数:

`pcWriteBuffer`: 存储命令执行的输出结果

`xWriteBufferLen`: `pcWriteBuffer` 长度

`pcCommandString`: 用户输入的整个命令行 (参数从此命令行中提取)

eg: 添加一个打印 Raw-os 内核版本的命令

```
// version 命令: 命令体定义
static const xCommandLineInput RAW_OS_Version = {
    (const RAW_S8 *) "version",
    (const RAW_S8 *) "version:          get the Raw-os kernel version.",
    get_os_version,
    0
};
```

对比上面的结构和 `struct xCOMMAND_LINE_INPUT` 结构,可知:

命令名为 `version`.

命令解释为 `get the Raw-os kernel version`.

命令的执行函数为 `get_os_version`.

命令的参数个数为 0.

命令的执行函数实现如下:

```
// version 命令回调: 打印 RAW-OS Kernel 版本
static RAW_S32 get_os_version(RAW_S8 *pcWriteBuffer, RAW_U32
xWriteBufferLen, const RAW_S8 * const pcCommandString)
{
    RAW_U8 version[6] = {0};
    // 将输出结果拷贝到输出缓冲区中
    raw_strcat((char *)pcWriteBuffer, (char *)"RAW-OS Version: ");
    version[0] = RAW_OS_VERSION/1000 + 0X30;
    version[1] = '!';
    version[2] = RAW_OS_VERSION%1000/100 + 0X30;
    version[3] = RAW_OS_VERSION%100/10 + 0X30;
    version[4] = RAW_OS_VERSION%10 + 0X30;
    version[5] = '\0';

    raw_strcat((char *)pcWriteBuffer, (char *)version);    // 拷贝到缓冲区
    return 1;
}
```

上面的命令执行函数具有固定的格式, 函数名随意, 但是函数的参数列表一定要是:

```
(RAW_S8 *pcWriteBuffer, RAW_U32 xWriteBufferLen, const
RAW_S8 * const pcCommandString)
```

其中, 第一个参数 `pcWriteBuffer` 指向一个命令输出缓冲区, 第三个参数 `pcCommandString` 即在串口终端输入的命令, 后面你要解析这个字符串来得到参数。

定义完命令后还需定义一个链表的结点, 以使该命令以链表结点的形式挂接在系统命令链表上。

```
static      xCommandLineInputListItem      pxNewListItem;
```

2. 注册这个新的命令

上面定义了一个新的命令, 命令体为 `RAW_OS_Version`, 接下来将该命令注册到系统中 (实际上添加进命令链表)。注册命令用到的函数如下:

```
RAW_VOID  rsh_register_command(const xCommandLineInput * const
pxCommandToRegister, xCommandLineInputListItem *pxNewListItem)
```

看起来, 这个函数有点麻烦啊, 不过用起来简单, 第一个参数为指向命令结构体的指针, 第二个参数为指向链表结点的指针, 把上面定义的命令体和链表结点注册到系统中, 代码如下:

```
rsh_register_command(&RAW_OS_Version, &pxNewListItem);
```

好了，这个时候，你的这个命令已经添加进系统当中去了。在终端输入 help，会打印出这个命令，当然，仅是以此例作为 shell 使用说明，在本例程中，实际上我们是以控制 LED 为例，添加了另外四个命令，输入 help 如下：

```
raw-os#  
raw-os#  
raw-os#help  
help: Lists all the registered commands  
  
led: <led number> <on or off> led number(1,2),on(1),off(0).  
led-water: Create led water task.  
led-stop: stop led water task.  
led-start: start led water task.  
raw-os#
```

以此四个命令为例说明了不带参数的命令注册使用方法和带参数的命令注册使用方法，上面命令中的 led 命令有两个参数，其命令体如下：

```
// led 开关,命令体定义  
static const xCommandLineInput led_command1 = {  
    (const RAW_S8 *)"led",  
    (const RAW_S8 *)"led: <led number> <on or off> led number(1,2),on(1),off(0).",  
    (pdCOMMAND_LINE_CALLBACK)led_test,  
    2  
};
```

上面命令体中的 led 命令有两个参数，第一个是 led number，第二个是开和关。那怎么获取命令参数呢？看代码：

```
// 获得第一个参数  
pParameter = (RAW_S8 *)rsh_get_parameter(pcCommandString,  
                                          1,  
                                          &paraStringLength);  
  
// 获得第二个参数  
pParameter = (RAW_S8 *)rsh_get_parameter(pcCommandString,  
                                          2,  
                                          &paraStringLength);
```

rsh_get_parameter 函数的第一个参数就是你输入的命令字符串，第二个参数表示你想获得第几个参数，第三个参数是获得的参数的长度。

该函数的返回值为指向目标参数字符串的指针。

实验目的：演示 shell 命令注册,使用的方法

实验思路：注册四个 LED 控制命令，实现控制指定 LED 的亮灭，创建 LED 流水灯任务，挂起流水灯任务，恢复流水灯任务。

实验代码：见例程代码 led_cmd.c

实验结果：

输入 help 查看系统已注册命令

```
raw-os#help
help: Lists all the registered commands

led: <led number> <on or off> led number(1,2),on(1),off(0).
led_water: Create led water task.
led_stop: stop led water task.
led_start: start led water task.
raw-os#led 1 1      点亮第一个LED

raw-os#led 1 0      熄灭第一个LED

raw-os#led 2 1      点亮第二个LED

raw-os#led 2 0      熄灭第二个LED

raw-os#led_water    创建LED流水灯任务并运行

raw-os#led_stop     挂起LED流水灯任务

raw-os#led_start    恢复LED流水灯任务

raw-os#led_stop

raw-os#
```

关于 shell 机制的命令体定义，命令注册部分更具体的内容请读者查看源码细细品味。总结一下就是：

1. 定义一个链表结点；
2. 定义一个命令结构体，结构体有四个成员，其中一个是函数指针；
3. 定义命令体的执行函数，函数名即为步骤 2 中命令体的第三个成员；
4. 注册这个命令；

其中需要注意的是，在有参数的命令中怎么获得指定的参数。无非就是一个结构体类型和几个 API 函数的使用啦，外加一点点链表知识。

5. 系统任务栈空间检测

在本节继续向系统注册了几个命令，包括打印内核版本号，打印 MCU 芯片 ID 号等，在 `system_cmd.c` 文件中，读者仔细分析源码，以期达到对 shell 随心所欲使用的目的。

查看内核版本的命令 `version` 和查看 MCU 芯片 ID 的命令 `chip_id` 命令执行如下：

```
raw-os#help
help: Lists all the registered commands

version:    get the raw-os kernel version.
chip_id:    get the chip 96 bits id and flash size.
reboot:     get the chip 96 bits id and flash size.
stack:      show all free task stack size

led:        <led number> <on or off> led number(1,2),on(1),off(0).
led_water:  Create led water task.
led_stop:   stop led water task.
led_start:  start led water task.
raw-os#version
RAW-OS Version: 1.056
raw-os#chip_id
chip id: 0x32383536 3333470C 0032003C
Flash Size: 1024 KB
raw-os#
```

查看内核版本

查看芯片ID和Flash大小

注意到在 `help` 命令列出的系统命令列表里还有个 `reboot` 命令，是系统重启，还有个 `stack` 命令则是本节重点要介绍的，先输入执行下试试看：

```
raw-os#stack
task name is idle_task *** task free stack size is 79
task name is shell Task *** task free stack size is 59
```

可以看到系统此时只有两个任务：`idle_task` 和 `shell_task`。`idle_task` 是系统必须有的一个任务，而 `shell task` 即为我们创建的 `shell` 任务。

下面讨论栈溢出问题。

栈溢出是 RTOS 使用当中经常遇到的问题，有时候莫名其妙的程序就跑飞了，裸机程序当然也有这种情况。比如调试时，在 STM32 平台经常出现程序死在 `HardFault_Handler` 中，这种情况很有可能就是栈空间溢出了，解决问题的方法就是给任务分配更大的栈，但是分配多大合适呢？你可以粗略估算，但是不可能精确掌握实际运行过程中的各种情况，所以动态查看栈空间使用情况是有必要的，而 RAW-OS 对此提供了很好的支持，经过简单的代码编写，向系统中注册一个命令 `stack` 则可以动态实时的查看系统中所有运行的任务的栈空间使用情况。

比如在终端中输入 `led_water` 执行流水灯任务，然后再输入 `stack` 执行结果如下：

```
raw-os#stack
task name is idle_task *** task free stack size is 79
task name is shell Task *** task free stack size is 59
task name is LED_Task *** task free stack size is 73
raw-os#
```


从上图可以看出，系统检查到 LED_Task 的任务栈剩余 73 个字空间(32 位平台一个字是 4 个字节)。

接着做测试，在 raw_config.h 中进行如下配置：

```
/*enable system zero interrupt*/
#define CONFIG_RAW_ZERO_INTERRUPT 1
#define CONFIG_RAW_TASK_0 1
#define RAW_CONFIG_CPU_TASK 1
```

即打开系统的最大关中断 0 微秒特性，创建 TASK_0 任务，打开 CPU_TASK 即统计任务，然后重新编译程序，下载，然后重新执行 stack 命令，查看结果如下：

```
raw-os#stack
task name is      idle_task *** task free stack size is 79
task name is      task_0_object *** task free stack size is 206
task name is      cpu_object *** task free stack size is 205
task name is      shell Task *** task free stack size is 49
task name is      LED_Task *** task free stack size is 73
raw-os#
```

可以看到，系统中已经有了 task_0_object 和 cpu_object 两个对像，变即优先级为 0 的 task_0 任务和统计任务。同时可查看他们的任务栈空间剩余。

好了，关于任务栈空间检测的更多探究请参阅《高效实时操作系统》中所论述，对 stack 命令的命令体定义，注册等不在列出，下面分析下其代码实现：

```
static RAW_S32 rsh_task_task_command(RAW_S8 *pcWriteBuffer, size_t
xWriteBufferLen, const RAW_S8 *pcCommandString)
{
    LIST *iter;
    LIST *iter_temp;
    RAW_U32 stack_free;
    RAW_TASK_OBJ *task_iter;
    // 1. 标识 task_debug 链表头
    iter = raw_task_debug.task_head.next;

    /*do it until list pointer is back to the original position*/
    while (iter != &(raw_task_debug.task_head)) {
        // 2. 遍历 task_debug 列表
        iter_temp = iter->next;
        // 3. 得到 task_debug 列表上的任务对像
        task_iter = raw_list_entry(iter, RAW_TASK_OBJ, task_debug_list);
        // 4. 获取该任务的剩余栈空间，存在 stack_free 变量中
        raw_task_stack_check(task_iter, &stack_free);
        // 5. 打印任务名子和对应的剩余栈空间
        Uart_Printf("task name is %15s *** task free stack size is %d\r\n",
task_iter->task_name, stack_free);
```

```

        /*move to list next*/
        iter = iter_temp;
    }

    return 1;
}

```

上面是 stack 命令的执行函数，其原理即为注释的 1, 2, 3, 4, 5 写的很清楚了，系统中有一个链表称为 task_debug，在任务创建的时候，每一个任务都会被挂着在这个链表上，当执行 stack 命令的时候，就会遍历这个链表，获得每一个任务对象，然后调用 raw_task_stack_check 函数去获取这个任务的剩余栈空间，然后打印出来结果。

接着分析 raw_task_stack_check 函数，分析源码发现该函数支持两种增长模式的栈空间，对于我们使用的 STM32 来说，栈增长方向当然是向下增长(高地址向低地址增长)，这个函数的关键代码如下：

```

task_stack = task_obj->task_stack_base;
while (*task_stack++ == 0) {
    free_stk++;
}

```

这段代码的原理是：在任务初始化的时候，它的栈空间被初始化为 0，如果栈被使用了，那么这块空间的值就不为 0，统计为 0 的空间大小，即得到剩余栈空间(未被使用)的大小。

因为栈是向下生长，所以先使用高地址的栈空间。首先得到任务的栈底(低地址)，然后以栈底为基地址向上增长判断是否为 0，如果为 0 则表示该区域未使用，free_stk 变量即最终得到的剩余栈大小。

栈空间的溢出与检测是个高级话题，本节演示了怎么动态查看系统中所有任务的栈剩余空间，又简要分析了 RAW-OS 系统中实现栈空间检测的原理，更深入的研究分析请查看本节例程原码和 RAW-OS 源码。

上面讲的是系统运行时手动的查看任务剩余栈空间。

然而，raw-os 的真正独到之处在于增加了第 4 种任务栈溢出的检测，大部分时候防患于未然，是最好的方式。只要在 raw_config.h 里面打开宏定义 RAW_SYSTEM_CHECK，系统的空闲任务就会无时不刻的帮用户去实时检测任务栈空间的使用情况，只要一检测出任务的栈空间已经不足 12% 的整体空间的时候，会立即停掉整个系统，来通知用户任务的栈溢出了。目前只有 raw-os 一家提供了这种任务栈的检测机制。具体的代码可以参考 raw_idle.c 不再细述。

6. 软件定时器

Raw-os 提供了定时器给用户去使用。软件定时器的代码在 `raw_timer.c` 里面。使用的时候需要打开 `raw_config.h` 中的 `CONFIG_RAW_TIMER` 和 `CONFIG_RAW_MUTEX`。

软件定时器类似于硬件定时器，当内部计数器达到某一值时会触发用户的自定义函数，这个函数也叫回调函数。这个函数可以用在各个场合，尤其是在一些协议栈中当做超时重发的定时器去使用，也可以用做去轮询一些外设的状态等。

用户可以创建任意多个软件定时器，只要内存足够大。

定时器的精度有 `raw_config.h` 里的 `RAW_TIMER_RATE` 去定义使用，具体的软件定时器的频率可以有公式：

$$\text{RAW_TICKS_PER_SECOND} / \text{RAW_TIMER_RATE}$$
 算出。

举一个例子，`RAW_TICKS_PER_SECOND` 为 100，`RAW_TIMER_RATE` 为 10，那说明软件定时器的频率是一秒 10 次，即一次是 100ms。

软件定时器的实现有一个系统开启的任务去实现，这个任务就是 `timer_task`，此任务的优先级和栈大小可以配置，默认为 5，配置代码如下：

```
#if (CONFIG_RAW_TIMER > 0)

/*set timer task stack size, adjust as you need*/
#define TIMER_STACK_SIZE          128
#define TIMER_TASK_PRIORITY      5

#endif
```

对于软件定时器的精度是有一定的时间误差的，误差时间一般在上下一个 tick 时间，有的人把系统定时器的频率开得很大，比如 1000Hz，这样对于系统损耗很大，对于 100MHz 以下的 cpu，推荐 50Hz 或者更小。如果系统中不存在超时机制，甚至可以不使用系统定时器，用来节约功耗和最大提升系统运行效率。

如果你一定要精度准的定时，怎么办？请使用硬件定时器。

raw os 的软件定时器有以下特点：

- 1 支持一次定时器触发用户的回调函数。
- 2 周期性的定时器触发用户的回调函数。
- 3 周期性的定时器触发用户的回调函数，但是第一次回调用户的函数的时间可以设置不同的时间。

对于软件定时器唯一需要特别注意的是因为软件定时器的回调函数是关了系统抢占运行的，所以需要回调函数运行时间要尽可能的短，否则系统的最大任务延迟会急剧增加，对系统的实时性损害太大。

上面说了软件定时器的理论知识，下面用它来解决实际的问题。

还记得在第 3 个例程里按键中断不稳定吗。下面创建一个周期性的定时器，在定时器周期性的服务函数里检测按键的状态，来稳定，可靠的获得按键的值。

需要定义一个定时器变量，一个回调函数，然后在 KEY_Init 中初始化硬件后创建一个定时器用来检测按键。

```
// 定义一个用于周期性检测按键的软件定时器
RAW_TIMER      Key_Timer;
// 声明按键定时器的回调函数
RAW_U16 Key_Timer_Function(RAW_VOID *expiration_input);
// 创建软件定时器,安装回调函数,设定周期等
// 定时器在创建后 100 个 ticks 后自动开始执行
// 周期为 2 个 ticks,即 20ms
raw_timer_create(&Key_Timer,(RAW_U8 )"Key_Timer",
                Key_Timer_Function, 0, 100, 2, 1);
```

采用状态机的思想检测按键的三个状态，以期达到稳定检测的目的。回调函数源码如下：

```
RAW_U16 Key_Timer_Function(RAW_VOID *expiration_input)
{

    switch(Key_State)                // 判断按键状态
    {
        case State_UP:              // 按键处于松开状态
        {
            // 按键 0,1,2 被按下
            if( !GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_2) ||
                !GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_3) ||
                !GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_4) )
            {
                Key_State = State_Press;    // 转到按下状态
            }
        }

        break;
        case State_Press:            // 按键处于按下状态
        {
            if(!GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_2) )
            {
                // 释放信号量(唤醒阻塞在该信号量上的最高优先级任务)
                raw_semaphore_put(&Key_Semaphore);
                Key_State = State_Press_OK; // 转到按下 OK 状态(等待松开)
            }
            else if( !GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_3) )
            {
```

```

        // 释放信号量(唤醒阻塞在该信号量上的最高优先级任务)
        raw_semaphore_put(&Key_Semaphore);
        Key_State = State_Press_OK; // 转到按下 OK 状态(等待松开)

    }
    else if( !GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_4))
    {
        // 释放信号量(唤醒所有阻塞在该信号量上的任务)
        raw_semaphore_put_all(&Key_Semaphore);
        Key_State = State_Press_OK; // 转到按下 OK 状态(等待松开)
    }
    else
    {
        Key_State = State_UP; // 按下无效返回到松开状态
    }
}
break;

case State_Press_OK: // 按键完成状态
{
    // 按键释放
    if( GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_2) &&
        GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_3) &&
        GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_4) )
    {
        Key_State = State_UP; // 松开状态
    }
}
break;
}

return 0;
}

```

以上代码采用了简单的状态机思想，按键按下后需要消抖一段时间，比如 20ms，转到下一个状态后再次判断，如果有按下则释放信号量，并转到按键完成状态等待按键松开；如果未按下则为无效(可能是受干扰)，然后直接返回到松开的状态。

需要注意的是，按下 KEY2 和 KEY1 是唤醒最高优先级任务，所以 TASK_2 一直得不到执行，它优先级低，按下 KEY0 则唤醒阻塞到该信号量上的所有优先级任务。实验效果为按下 KEY2 和 KEY1 板上红色 LED 翻转，按下 KEY0 则两个 LED 都翻转。

思考：现在按键只支持单次触发，按下后必须松开才能再次响应按键，怎样稍微更改代码使按键支持连续触发呢？

答：1. 可以 Post 唤醒一个任务，也可以 Post 唤醒所有任务，Post 唤醒一个任务时，可以唤醒优先级最高的 (RR 模式)，也可以唤醒最先等待消息的 (FIFO 模式)。

注：raw_config.h 中可以指定调度的方式为 FIFO 还是 RR（优先级）

```
/*enable SCHED_FIFO and SCHED_RR support*/
#define CONFIG_SCHED_FIFO_RR
```

1

2. 可以把消息 post 到队列的最前面,也可以 post 到队列的最后面,Raw-os 提供不同的 API 供用户选用。

实验目的：学习消息队列(raw_queue)的使用

实验思路：板上三个按键按下时分别发送“KEY0”，“KEY1”，“KEY2”三则消息到队列中，创建一个任务平时阻塞在消息上，一旦有消息则任务被唤醒读取队列中内容，并根据内容来控制 LED 的不同状态，实际的程序引入了等待超时机制。

实验关键代码：

```
RAW_QUEUE    Key_Queue;           // 定义一个消息队列
u8 *key_queue[10];               // 定义一个指针数组,存储指向消息的指针

// 创建一个消息队列,用于按键与任务同步
// 队列可以存放 10 条消息
raw_queue_create(&Key_Queue, (RAW_U8 *) "KEY_QUEUE", (RAW_VOID
**)&key_queue, 10);

// 发送消息到队列的尾部
raw_queue_end_post(&Key_Queue, "KEY2");

// 消息接收处理任务的执行函数
void Task_Queue_Fun(void * pParam)
{
    RAW_U16 res;
    RAW_U32 timeout_value = 500;           // 500 个 ticks 的超时等待
    signed char *msg;

    while(1)
    {
        // 等待消息(等待时间为 timeout_value, 5 秒钟, 超时则返回)
        res = raw_queue_receive(&Key_Queue, timeout_value, (RAW_VOID
**)&msg);

        if( RAW_BLOCK_TIMEOUT == res )           // 等待超时则返回
        {
            USART1_Send_String((RAW_S8 *) "\r\n");
            USART1_Send_String((RAW_S8 *) "receive queue timeout!\r\n");
        }
        else if( RAW_SUCCESS == res)             // 成功接收到消息
        {
```

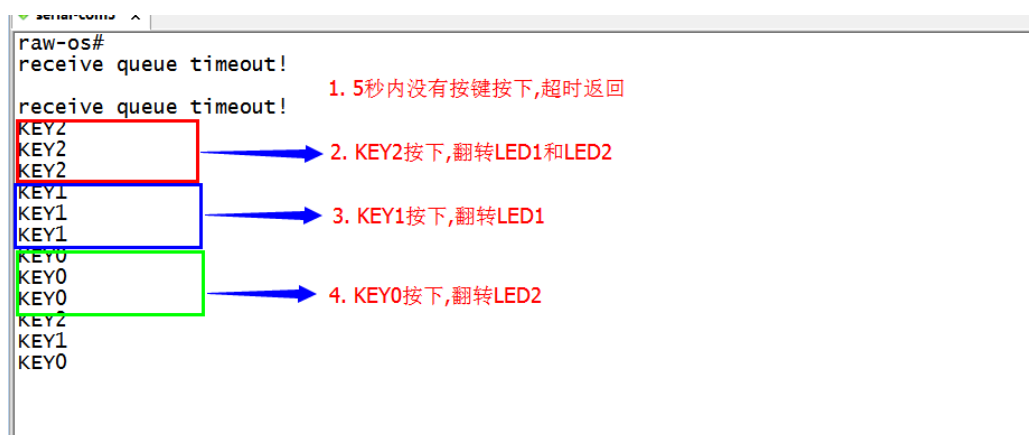
```

USART1_Send_String(msg);           // 打印消息
USART1_Send_String((RAW_S8 *)"\r\n");

// 根据消息内容控制不同的 LED
if( 0 == raw_strncmp((const char *)msg,"KEY0",4) )
{
    LED_Toggle(1);
}
else if(0 == raw_strncmp((const char *)msg,"KEY1",4) )
{
    LED_Toggle(2);
}
else if(0 == raw_strncmp((const char *)msg,"KEY2",4) )
{
    LED_Toggle(1);
    LED_Toggle(2);
}
}
}
}

```

实验结果:



由上结果可知，5 秒内没有按键按下，则任务超时返回，有按键按下时发送消息到队列中，任务接收后打印出消息内容，并根据内容控制 LED 翻转。

8. 消息队列(raw_queue_size)

queue 在通信时，为了加快数据的传递速度，是不直接发送数据的具体内容，而是发送指向用户数据的指针，而且这个指针是 void 指针（强制类型转换），在取出 queue 当中的数据时，强制转换这个 void 指针成发送的原始数据内容的指针类型，就可以准确获取原始数据。

queue_size 就是 queue 的扩展，就是将指向原始数据的指针和表示原始数据大小的变量，再打包封装成一个新的 msg，称这个 msg 就是 queue_size 的 msg。这个新的数组结构名为 RAW_MSG_SIZE，其定义如下：

```
typedef struct RAW_MSG_SIZE {

    struct RAW_MSG_SIZE    *next;
    void                    *msg_ptr;
    MSG_SIZE_TYPE           msg_size;

} RAW_MSG_SIZE;
```

封装好的 RAW_MSG_SIZE 即为 queue_size 存储的基本单元，这个 msg 包含了指向用户发送数据段的指针 void *msg_ptr，和用户数据段的大小 msg_size。

还有一个*next 的指针存在，这个*next 指针就是将这些 msg 组成一个单项链表而存在的。

所以在使用的时候，队列的定义如下：

```
RAW_QUEUE_SIZE Key_Queue;           // 定义一个消息队列
RAW_MSG_SIZE   key_queue[10];       // 消息队列中可以存储 10 条消息
```

创建队列如下：

```
raw_queue_size_create(&Key_Queue, "KEY_QUEUE", key_queue,
10);
```

发送消息时同时发送其大小：

```
raw_queue_size_end_post(&Key_Queue, "KEY2", sizeof("KEY2"));
```

接收消息时同时接收其大小：

```
raw_queue_size_receive(&Key_Queue, timeout_value, &msg, &msg_length);
```

其中 msg 和 msg_length 定义如下：

```
void *msg;           // 指向消息
MSG_SIZE_TYPE msg_length; // 指向消息大小
```

实验目的：学习消息队列(raw_queue_size)的使用

实验思路：板上三个按键按下时分别发送“KEY0 12345”，“KEY1 1234”，“KEY2 123”三则消息到队列中，创建一个任务平时阻塞在消息上，一旦有消

息则任务被唤醒读取队列中内容，并根据内容来控制 LED 的不同状态，实际的程序引入了等待超时机制。

实验关键代码：

```
// 消息处理任务执行函数
void Task_Queue_Size_Fun(void * pParam)
{
    RAW_U16 res;
    RAW_U32 timeout_value = 500;           // 500 个时钟周期,即 5s
    void *msg;                             // 消息指针
    MSG_SIZE_TYPE msg_length;              // 消息大小

    while(1)
    {
        // 等待消息(等待 5 秒钟,如果还没有消息,则超时返回)
        res = raw_queue_size_receive(&Key_Queue, timeout_value, &msg,
&msg_length);

        if( RAW_BLOCK_TIMEOUT == res )      // 超时返回
        {
            Uart_Printf("receive queue timeout!\r\n");
        }
        else if( RAW_SUCCESS == res)        // 队列中有消息
        {
            // 打印消息及其长度
            Uart_Printf("%s, queue_size: %d\r\n", (char*)msg, msg_length);
            // 根据消息内容点灯
            if( 0 == raw_strncmp((const char *)msg, "KEY0", 4) )
            {
                LED_Toggle(1);
            }
            else if(0 == raw_strncmp((const char *)msg, "KEY1", 4) )
            {
                LED_Toggle(2);
            }
            else if(0 == raw_strncmp((const char *)msg, "KEY2", 4) )
            {
                LED_Toggle(1);
                LED_Toggle(2);
            }
        }
    }
}
```


实验结果:

```

receive queue timeout!
KEY2 123, queue_size: 9
KEY1 1234, queue_size: 10
KEY0 12345, queue_size: 11
KEY0 12345, queue_size: 11
KEY0 12345, queue_size: 11
KEY2 123, queue_size: 9
KEY2 123, queue_size: 9
KEY2 123, queue_size: 9
KEY1 1234, queue_size: 10
KEY1 1234, queue_size: 10
KEY1 1234, queue_size: 10
receive queue timeout!
receive queue timeout!
KEY1 1234, queue_size: 10
KEY1 1234, queue_size: 10
KEY0 12345, queue_size: 11
KEY0 12345, queue_size: 11
receive queue timeout!
receive queue timeout!

```

1. 打印消息内容和大小

2. 打印消息内容和大小

3. 打印消息内容和大小

9. 消息队列(raw_queue_buffer)

前面已经详细讲述了 queue 和 queue_size 模块，结合代码和实验效果加深理解之后，再来看这个 queue_buffer 就非常之简单了，对于 queue 是转发消息的指针，queue_size 则是转发消息指针和消息大小，那么 queue_buffer 不在转存消息的指针，而是将消息 copy 一份到用户空间，也即转发消息内容和消息大小。

一个 queue_buffer 消息包括消息的大小，消息具体数据内容，作为一个整体直接 copy 到创建 queue_buffer 时开辟的缓冲区中，下一条消息到来时仍是这样在上一条消息的尾部接着 copy。一个 queue_buffer 内部维护着一个顺序的队列。

消息队列 queue_buffer 的定义如下：

```
RAW_QUEUE_BUFFER Key_Queue;           // 创建一个消息队列(queue_buffer)
RAW_U32           queue_buffer[1024]; // 消息队列 Buffer
```

其中 queue_buffer 数组即为 Key_Queue 内部维护的一个顺序队列(此顺序队列为数组结构理论中的队列,非 RTOS 中的队列), 用户发送消息时, 消息的内容即 copy 到这个顺序队列中, 创建队列如下:

```
raw_queue_buffer_create(&Key_Queue, (RAW_U8 *) "KEY_QUEUE", queue_buffer,
1024 * 4, 0xFFFF);
```

发送消息时同时发送其大小:

```
raw_queue_buffer_end_post(&Key_Queue, "KEY2", 10);
```

接收消息时同时接收其大小:

```
raw_queue_buffer_receive(&Key_Queue, timeout_value, &msg,
&msg_length);
```

其中 msg 和 msg_length 定义如下:

```
void *msg;           // 指向消息内容
MSG_SIZE_TYPE msg_length; // 消息大小
```

实验目的：学习消息队列(raw_queue_buffer)的使用

实验思路：板上三个按键按下时分别发送“KEY0”，“KEY1”，“KEY2”三则消息到队列中，创建一个任务平时阻塞在消息上，一旦有消息则任务被唤醒读取队列中内容，并根据内容来控制 LED 的不同状态，实际的程序引入了等待超时机制。

实验关键代码：

```
// 消息处理任务执行函数
void Task_Queue_Buffer_Fun(void * pParam)
{
    RAW_U16 res;
    RAW_U32 timeout_value = 500;           // 500 个时钟周期,即 5s
```

```
RAW_U8 msg[128]; // 消息缓冲
MSG_SIZE_TYPE msg_length; // 消息大小

// 接收缓冲清空
raw_memset(msg, 0, sizeof(msg));

while(1)
{
    // 等待消息(等待 5 秒钟,如果还没有消息,则超时返回)
    res = raw_queue_buffer_receive(&Key_Queue, timeout_value, msg,
&msg_length);

    if( RAW_BLOCK_TIMEOUT == res ) // 超时返回
    {
        Uart_Printf("receive queue timeout!\r\n");
    }
    else if( RAW_SUCCESS == res) // 队列中有消息
    {
        // 打印消息及其长度
        Uart_Printf("%s, queue_size: %d\r\n",msg,msg_length);

        // 根据消息内容点灯
        if( 0 == raw_strncmp((const char *)msg,"KEY0",4) )
        {
            LED_Toggle(1);
        }
        else if(0 == raw_strncmp((const char *)msg,"KEY1",4) )
        {
            LED_Toggle(2);
        }
        else if(0 == raw_strncmp((const char *)msg,"KEY2",4) )
        {
            LED_Toggle(1);
            LED_Toggle(2);
        }
    }
}
```

实验结果:

```
receive queue timeout!  
receive queue timeout!  
KEY0, queue_size: 30  
KEY0, queue_size: 30  
KEY0, queue_size: 30  
KEY1, queue_size: 20  
KEY1, queue_size: 20  
KEY1, queue_size: 20  
KEY2, queue_size: 10  
KEY2, queue_size: 10  
KEY2, queue_size: 10  
KEY2, queue_size: 10  
receive queue timeout!  
receive queue timeout!  
█
```

1. 按键0按下,打印接收到的消息内容和长度

2. 按键1按下,打印接收到的消息内容和长度

3. 按键2按下,打印接收到的消息内容和长度

注意: 在发送消息时,消息的长度是随意指定的,接收时并不会判断有效的数据长度,这也和 queue_buffer 的内部数据存储, copy 等实现机制有关,具体请研究内核源码。

10. 多任务带来的临界区资源使用问题

临界区问题是嵌入式软件编程一个不得不面对的关键性问题。特别对于底层驱动，代码在内存中只有一份，上层的多任务或者多进程，都会对同一个驱动去访问，这样不可避免的遇到了任务之间打架的问题，处理好这个问题是区分一个菜鸟和老鸟的根本性关键之一。

共享资源管理，临界区问题的产生及解决，具体的理论内容请参见《高效实时操作系统设计》一书中相关章节，书中对相关问题阐述的很清楚，本节例程对此问题作一个形象的演示，在下两节将说明采用 **mutex** 来解决此问题。

在本例程序中注册了 **lcd_test** 命令，请首先保证 LCD 驱动程序正确，LCD 设备能正常使用并显示文字。

实验目的：分析临界资源的产生原因及现象

实验思路：创建两个任务分时使用 LCD，在低优先级任务使用共享资源 LCD 设备的时候，会被高优先级任务抢占，这样造成 LCD 设备使用混乱，因为低优先级的任务并没有使用完，却被高优先级任务抢占了。

实验关键代码：

```
// 第一个任务的函数入口
void First_Task(void * pParam)
{
    while(1)
    {
        // 优先级高,2s 抢占一次 LCD 资源
        LCD_Clear(LCD_COLOR_MAGENTA);
        LCD_ShowString(20,180,200,16,16,(u8 *)"Hello This is First Task");
        raw_sleep(200);
    }
}

// 第二个任务的函数入口
void Second_Task(void * pParam)
{
    char str[] = "This is Second Task: ";
    int i = 0;

    while(1)
    {
        // 优先级低,在使用 LCD 过程中可能会被高优先级的任务 2 抢占
        while(++i < 10)
        {
            LCD_Clear(LCD_COLOR_GREY);
            str[sizeof(str) - 2] = i + 0x30; // 打印执行次数
            LCD_ShowString(20,200,200,16,16,(u8 *)str);
        }
    }
}
```

```
        delay_ms(200);                                /* 模拟 LCD 被长时间占用 */
    }
    if(i == 10)
    {
        i = 0;
    }
    raw_sleep(100);
}
}
```

从上面代码中是可以看到的，任务 1 的优先级高，会隔一定的时间去抢占 LCD 资源，这时候，正在使用 LCD 的任务 2 就被打断了。

实验结果：

任务 2 会占用 LCD，并以一定的间隔在屏幕上显示 1-9，本意是在使用完毕后，释放 CPU 和资源供其它任务使用，但实际效果是任务 2 每 2S 就被任务 1 抢占，此时它还未使用完 LCD。

11. mutex 使用 1

mutex 的出现是为了解决优先级反转的问题，由于优先级反转对实时性影响太大，所以 mutex 的稳定性直接影响了实时性。纵观目前多种实时操作系统 mutex 的设计原理是多多少少有一点问题的，很多 RTOS 没有实现 mutex 优先级逐步还原的问题，导致了实时性以及其它的一些逻辑错误等等。raw os 的 mutex 模块成功弥补了其它实时系统在这方面的不足。

Raw os 的 mutex 同时支持优先级置顶和优先级继承的方式来解决优先级反转的问题。

关于优先级反转的问题产生原因，优先级继承和优先级置顶的原理请参见《高效实时操作系统》书中相关章节的论述，本部分原理理解难度稍微有点大，需要读者细细探究体会。

对于 mutex 的使用，首先是定义并创建它：

```
// 定义一个 mutex 用于保护 LCD 设备资源互斥使用
RAW_MUTEX lcd_mutex;
// 创建一个 mutex, 解决优先级反转的方式为 RAW_MUTEX_INHERIT_POLICY (优先级继承)
raw_mutex_create(&lcd_mutex, (RAW_U8 *) "lcd_mutex",
RAW_MUTEX_INHERIT_POLICY , 0);
```

所谓优先级继承的含义是，当优先级反转发生的时候，低优先级任务的优先级被自动提升为高优先任务的优先级，比如在本例中两个任务的优先级分别 10 和 11，当优先级为 11 的任务占用锁时，它的优先级被提升为 10，这样低优先级的任务不会被其它中优先级的任务打断，从而保证了高优先级任务执行周期的确定性，也即保证了系统的实时性。

实验目的： 分析采用 mutex 保护共享资源后任务的执行细节

实验思路： 创建两个任务分时使用 LCD，在低优先级任务使用共享资源 LCD 设备的时候，因为使用了 mutex，它的优先级会被提升为高优先级任务的优先级，直到它使用完毕释放锁后，其优先级被还原为原来的优先级。

(注：本实验仅为演示 mutex 的使用和执行过程的细节)

实验关键代码：

```
// 第一个任务的函数入口
void First_Task(void * pParam)
{
    while(1)
    {
        // 1. 任务 1 优先级高, 但要使用临界资源, 也要先获得 mutex
        raw_mutex_get(&lcd_mutex, RAW_WAIT_FOREVER);
        Uart_Printf("Task1 get mutex, Task1 current Priority is:
%d\r\n", Task1.priority);

        // 2. 获得成功后, 使用临界资源
```

```

    LCD_Clear(LCD_COLOR_MAGENTA);
    LCD_ShowString(20,180,200,16,16,(u8 *)"Hello This is First
Task");
    delay_ms(500);                /* 模拟使用 LCD 一段时间 */

    // 3. 使用完毕,释放临界资源
    Uart_Printf("Task1 put mutex\r\n");
    raw_mutex_put(&lcd_mutex);

    raw_sleep(10);
}
}

// 第二个任务的函数入口
void Second_Task(void * pParam)
{
    char str[] = "This is Second Task: ";
    int i = 0;

    while(1)
    {
        // 1. 任务 2 优先级低,要使用临界资源,需等待 Task1 释放
        raw_mutex_get(&lcd_mutex,RAW_WAIT_FOREVER);

        // 任务切换,Task1 要获得锁但被 Task2 占有,Task1 把 Task2 的优先级提高,然后
        执行 Task2
        raw_sleep(100);
        // 打印 Task2 的运行优先级,此时已被提高到和 Task1 同一个优先级
        Uart_Printf("Task2 get mutex, Task2 current Priority is:
%d\r\n",Task2.priority);

        // 2. 获得锁成功后,使用临界资源
        i = 0;
        while(++i <= 10)                /* 模拟使用 LCD 一段时间 */
        {
            delay_ms(200);
            LCD_Clear(LCD_COLOR_GREY);
            str[sizeof(str) - 2] = i + 0x30; // 打印执行次数
            LCD_ShowString(20,200,200,16,16,(u8 *)str);
        }

        // 3. 使用完毕后,释放临界资源,此时会唤醒阻塞在该 mutex 上的 Task1
        Uart_Printf("Task2 put mutex!\r\n");
        raw_mutex_put(&lcd_mutex);
    }
}

```



```
// 释放锁之后,任务优先级还原
    Uart_Printf("Task2 put mutex complete, Task2 current Priority is:
%d\r\n",Task2.priority);
}
}
```

实验结果:

```
Task1 get mutex,Task1 current Priority is: 10
Task1 put mutex
raw-os#Task2 get mutex, Task2 current Priority is: 10
Task2 put mutex!
Task1 get mutex,Task1 current Priority is: 10
Task1 put mutex
Task2 put mutex complete, Task2 current Priority is: 11
Task2 get mutex, Task2 current Priority is: 10
Task2 put mutex!
Task1 get mutex,Task1 current Priority is: 10
Task1 put mutex
Task2 put mutex complete, Task2 current Priority is: 11
Task2 get mutex, Task2 current Priority is: 10
```

1. 任务1先获得mutex,直到其使用完毕。

2. 任务2获得mutex,它的优先级继承任务1,为10

3. 任务1在任务2释放mutex后又抢占,直到raw_sleep发生任务切换!

4. 任务切换后,任务2继续执行,释放锁后,它的优先级被还原到11

此实验的关键在于要理解任务 2 在获得锁后需要调用 raw_sleep 进行任务切换以使得任务 1 有机会运行从而提升任务 2 的优先为和自己一样(故称优先级继承),然后接着执行任务 2 使用 LCD 设备,直到使用完毕释放锁后,阻塞在在 mutex 上的任务 1 被唤醒,得到执行,此时任务 2 的优先级被还原为它原本自己的优先级。

12. mutex 使用 2

本例采用优先级置顶的方式来解决优先级反转问题。

对于 mutex 的使用，首先是定义并创建它：

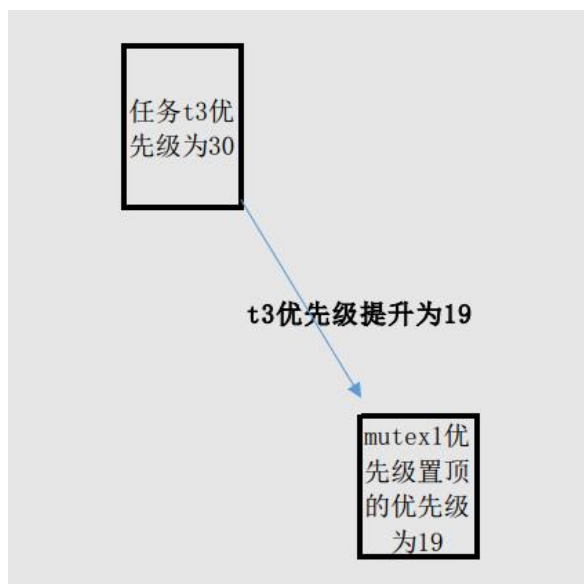
```
// 定义一个 mutex 用于保护 LCD 设备资源互斥使用
RAW_MUTEX lcd_mutex;
// 创建一个互斥锁, 用于保护 LCD 临界资源 (解决优先级反转的方式为优先级置顶方法)
// 任务占有 mutex 时置顶的优先级为 8
raw_mutex_create(&lcd_mutex, (RAW_U8 *) "lcd_mutex",
RAW_MUTEX_CEILING_POLICY, 8);
```

注意到 raw_mutex_create 函数三个参数，有三种取值：

```
#define RAW_MUTEX_INHERIT_POLICY    0x00000002U
#define RAW_MUTEX_CEILING_POLICY    0x00000003U
#define RAW_MUTEX_NONE_POLICY       0x00000004U
```

第一种即 Priority inherited protocol(优先级继承)，第二种即 Upper limit priority protocol(优先级置顶)。

所谓优先级置顶方法需要给每一个 mutex 静态指定一个优先级，第一个获得 mutex 的任务会把优先级提高到静态指定的优先级。优先级置顶的方法需要把竞争这个 mutex 的所有任务优先级都要搞清楚，通常适用于系统中运行的任务优先级不改变的情况。如果一个任务会获得多层嵌套的 mutex 锁，这个时候使用优先级继承的方法设计不好的话容易死锁，但是使用优先级置顶的方法能够避免此种情况出现。



假设访问 mutex1 的任务有 t1, t2, t3, t1 的优先级为 20, t2 的优先级为 25, t3 优先级为 30, 数字越小表明优先级越高。这个时候 mutex1 指定的优先级置顶的优先级是 19, 即比 t1, t2 的优先级要高。假设任务 t3 的

首先获得 mutex1, 这个时候任务 t3 的优先级会提升为 19, 所以 t3 会运行完之后释放锁轮到 t1 运行综上所述, 优先级置顶的话需要事先静态分析清楚, 的确是比较麻烦的, 好处是系统可以一目了然, 不存在暗箱, 做到系统每一个点都是很清楚。

实验目的: 分析采用 mutex 保护共享资源后任务的执行细节

实验思路: 创建两个任务分时使用 LCD, 在低优先级任务使用共享资源 LCD 设备的时候, 因为使用了 mutex, 它的优先级会被提升为创建 mutex 时指定的置顶优先级, 直到它使用完毕释放锁后, 其优先级被还原为原来的优先级。

(注: 本实验仅为演示 mutex 的使用和执行过程的细节)

实验关键代码:

```
// 第一个任务的执行函数入口
void First_Task(void * pParam)
{
    while(1)
    {
        // 1. 任务 1 优先级高, 但要使用临界资源, 也要先获得 mutex
        raw_mutex_get(&lcd_mutex, RAW_WAIT_FOREVER);
        // 在占用 mutex 期间, 其优先级被提高到创建 mutex 时指定的优先级 8
        Uart_Printf("Task1 get mutex, Task1 current Priority is:
%d\r\n", Task1.priority);

        // 2. 获得成功后, 使用临界资源
        LCD_Clear(LCD_COLOR_MAGENTA);
        LCD_ShowString(20, 180, 200, 16, 16, (u8 *) "Hello This is First
Task");
        delay_ms(500);          /* 模拟使用一段时间 */

        // 3. 使用完毕, 释放临界资源
        raw_mutex_put(&lcd_mutex);
        // 释放 mutex 后, 其优先级恢复为 10
        Uart_Printf("Task1 put mutex, Task1 current Priority is:
%d\r\n", Task1.priority);

        raw_sleep(10);
    }
}

// 第二个任务的执行函数入口
void Second_Task(void * pParam)
{
    char str[] = "This is Second Task: ";
    int i = 0;
```

```

while(1)
{
    // 1. 任务 2 优先级低,要使用临界资源,需等待 Task1 释放
    raw_mutex_get(&lcd_mutex,RAW_WAIT_FOREVER);

    // 在占用 mutex 期间,其优先级被提高到创建 mutex 时指定置顶的优先级 8
    Uart_Printf("Task2 get mutex, Task2 current Priority is:
%d\r\n",Task2.priority);

    // 2. 获得 mutex 成功后,使用临界资源
    i = 0;
    while(++i <= 10)
    {
        delay_ms(200);                      /* 模拟使用一段时间 */
        LCD_Clear(LCD_COLOR_GREY);
        str[sizeof(str) - 2] = i + 0x30;      // LCD 上打印执行次数
        LCD_ShowString(20,200,200,16,16,(u8 *)str);
    }

    // 3. 使用完毕后,释放临界资源,此时会唤醒阻塞在该 mutex 上的 Task1
    raw_mutex_put(&lcd_mutex);
    // 释放 mutex 后,其优先级恢复为 11
    Uart_Printf("Task2 put mutex,Task2 current Priority is:
%d\r\n",Task2.priority);

    raw_sleep(10);

}
}

```

实验结果:

```

Task1 get mutex,Task1 current Priority is: 8
Task1 put mutex,Task1 current Priority is: 10
Task2 get mutex, Task2 current Priority is: 8
Task2 put mutex,Task2 current Priority is: 11
Task1 get mutex,Task1 current Priority is: 8
Task1 put mutex,Task1 current Priority is: 10
Task2 get mutex, Task2 current Priority is: 8
Task2 put mutex,Task2 current Priority is: 11
Task1 get mutex,Task1 current Priority is: 8
Task1 put mutex,Task1 current Priority is: 10
Task2 get mutex, Task2 current Priority is: 8

```

1. 任务1获得锁,其优先级被置顶为8
2. 任务1释放锁,优先级恢复
3. 任务2获得锁,其优先级被置顶为8
4. 任务2释放锁,其优先级恢复为11

读者需要分析源码体会两种解决优先级反转方式的不同之处。

13. 内存管理(raw_block)

动态内存分配往往是一个必需的过程, 标准 C 库的 `malloc` 以及 `free` 可以满足动态内存的需求, 但是存在着以下的缺点:

- 1 嵌入式系统上不是总是有标准 C 库
- 2 标准 C 库可能会占据太多的代码位置
- 3 标准 C 库中的 `malloc` 以及 `free` 是非可重入的, 非多任务安全。
- 4 时间上存在不可预知性, 对实时性不利。

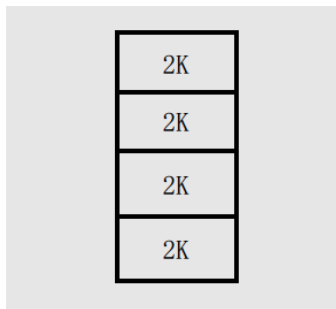
为了解决以上的缺陷, 本章提出了一些内存分配方法。

内存管理往往是一个复杂的过程, Raw-os 目前支持 5 种内存分配方法, 分别是 `block`,

`byte`, `page`, `malloc`, `slab`。以下会讲解这 5 种基本原理以及适用场合。

这 5 种里面只有 `block` 和 `slab` 是没有内存碎片的, 其余的都会有一定的内存碎片。这 5 种里面只有 `block` 和 `byte` 能用于中断内内存分配, 强烈推荐使用中中断内采用 `block` 内存分配, 因为 `byte` 内存的分配除了在最坏分配情况下, 时间是不确定的, 也就是说可能会占用大量的时间, 这个时候放在中断里面去做的话, 势必会阻塞其它低优先级的中断的相应, 进而可能造成硬件模块数据的丢失。

`block` 内存分配顾名思义是一块块的分配内存给用户, 每一块的内存空间都是固定的, 最大的好处是没有内存碎片, 而且速度很快, 缺点是内存空间大小固定了不利于不确定大小的内存分配。实战中可以分配多个不同大小的 `block` 去满足变长的分配完全是可行的。强烈建议大部分情况用 `block` 去解决内存分配问题。



如上如图所示分配了 4 个块, 每块大小为 2K。用户使用 `block` 内存分配一次即取得一个 2K 的大小。

`byte` 内存分配顾名思义是可以分配任何大小的字节的内存给用户, 缺点是速度比较慢, 而且有内存碎片, 不建议对时间敏感的任务去使用, 因为分配内存的速度比较慢, 而且很可能时间是不恒定的, 不利于实时任务。

`page` 内存分配顾名思义是可以分配以 `page` 为单位的内存给用户去使用, `page` 的大小可以配置的, 比如一个 `page` 2K 或者 4K 大小。好处是用户大内存分配的话很方便, 速度也很快。坏处是可能存在 `page` 级别的内存碎片。

`malloc` 内存分配是可以分配任意字节的小的, 也存在内存碎片。具体和标准 C 库的内存分配全部吻合。

slab 内存分配也是基于一块块的分配给用户，不过这一块块的各自大小是不同的，很可能会分配多余的内存空间给用户，优点是速度快没有内存碎片。缺点是可能会分配多余的内存给用户。

可以看到 raw os 支持的内存种类繁多，从小级别的内存分配到大级别的内存分配应有尽有，在项目实战中可以选择最适合自己的策略去使用。

block 内存分配方法对应的是 raw_block.c, byte 内存分配的方法对应的是 raw_byte.c, page 内存分配的方法对应的是 raw_page.c, malloc 的内存分配算法对应的是 raw_malloc.c, slab 内存分配算法对应的是 raw_slab.c。

在尊重实时操作系统 freeRTOS 的版权下，RAW-OS 引进了额外 4 种内存分配方式。分别是 heap_1, heap_2, heap_3, heap_4, 需要注意的是这 4 种内存分配方法不能用于中断中的内存分配。

heap_1 所采用的内存分配特点是方式是只分配，但是不能释放。

heap_2 采用了 best fit 的算法，但是不支持临进空闲区的并成一个空闲区。

heap_3 采用了 C 库自定义的 malloc 以及 free, 但是加了可重入机制，对于多任务是安全的。

heap_4 采用 first fit 的算法，但是支持临进空闲区的合并。

当使用 heap_1, heap_2, heap_3, heap_4 的时候，需要分别他们修改他们各自的配置文件来指定堆的大小等，比如 heap_1_config.h, heap_2_config.h, heap_3_config.h, heap_4_config.h。

对于内存分配算法，建议实际应用中以块内存分配为主，因为实时性和执行效率是可以兼得的。块内存分配方法理解比较容易，其它的分配算法都比较难理解，对于绝大部分人，只要能使用就足够了。

以上所列举的内存分配方式里面，只有 block 以及 byte 内存分配适合在中断内使用，其它的内存分配方式一律只能在任务空间使用，否则该内存分配方式的临界区将会出现问题。从而导致内存分配错误

本节讲解 block 内存分配方式的用法和简单演示，下节讲解 byte 内存分配方式的用法和简单演示，然后再讲解一下 heap_4 内存分配方式的用法和简单演示。

Block 内存管理的变量定义，内存池创建如下：

```
#define RAW_BLOCK_SIZE 1024 * 10 // block 内存大小 10K

// 用数组申请的静态空间交由内存控制器去动态管理
unsigned char raw_block_pool[RAW_BLOCK_SIZE];

MEM_POOL block_pool; // 定义一个块内存池(变量),用来管理 block 类型的内存

// 创建一个块内存池,总大小 10K,块大小 1K
raw_block_pool_create(&block_pool, (RAW_U8 *) "block_pool", 1024,
raw_block_pool, RAW_BLOCK_SIZE);
```

内存申请与释放如下：

```
raw_block_allocate(&block_pool, (RAW_VOID **) &block_ptr);
raw_block_release(&block_pool, (RAW_VOID *) block_ptr);
```

其中 block_ptr 为一个指针，用来指向申请得到的内存地址。

实验目的： 演示 block 内存管理方式的使用

实验思路： 创建一个具有 10K 大小的内存控制块(内存池)，按键 2 按下，则申请一块内存，并打印内存块的起始地址，按键 1 按下，则释放一块内存，按键 0 按下，则释放所有的内存块。

实验关键代码：

```
// 第一个任务的执行函数入口
void First_Task(void * pParam)
{
    int ret;
    int i = 0;

    while(1)
    {
        // 阻塞等待按键 2 信号量
        raw_semaphore_get(&Key2_Semaphore, RAW_WAIT_FOREVER);

        // 按键按下, 则分配内存块
        ret = raw_block_allocate(&block_pool, (RAW_VOID
        **)&block_ptr[i]);
        if ( RAW_NO_MEMORY == ret )           // 分配失败
        {
            Uart_Printf("block pool no memory to be allocated!\r\n");
        }
        else                                   // 分配成功
        {
            // 打印分配的内存地址
            Uart_Printf("block allocated success! block number: %d, block
            ptr: %p\r\n", i, block_ptr[i]) ;
            i ++;

            if(i >= 20)
            {
                i = 0;
            }
        }
    }
}

// 第二个任务的执行函数入口
void Second_Task(void * pParam)
{

```

```
int i = 0;
int ret;

while(1)
{
    // 阻塞等待按键 1 信号量
    raw_semaphore_get(&Key1_Semaphore, RAW_WAIT_FOREVER);

    ret = raw_block_release(&block_pool, (RAW_VOID *)block_ptr[i]);

    if(RAW_SUCCESS != ret)
    {
        Uart_Printf("block release error!, block number: %d, block ptr: %p\r\n", i, block_ptr[i]);
    }
    else
    {
        block_ptr[i] = NULL;           // 释放后要置 NULL
        Uart_Printf("block release success!, block number: %d, block ptr: %p\r\n", i, block_ptr[i]);
    }

    i ++;
}

// 第二个任务的执行函数入口
void Thrid_Task(void * pParam)
{
    int i;
    int ret;

    while(1)
    {
        // 阻塞等待按键 0 信号量
        raw_semaphore_get(&Key0_Semaphore, RAW_WAIT_FOREVER);

        for(i = 0; i < 20; i ++)        // 遍历
        {
            if( block_ptr[i] != NULL ) // 有效内存, 则释放
            {
                ret = raw_block_release(&block_pool, (RAW_VOID *)block_ptr[i]);
            }
        }
    }
}
```



```

        if(RAW_SUCCESS != ret)
        {
            Uart_Printf("block release error!,block number:
%d,block ptr: %p\r\n",i,block_ptr[i]);
        }
        else
        {
            block_ptr[i] = NULL;           // 释放后要置 NULL
            Uart_Printf("block release success!,block number:
%d,block ptr: %p\r\n",i,block_ptr[i]);
        }
    }
}
}
}
}

```

实验结果:

```

block pool create success!
raw-os#block allocated success! block number: 0, block ptr: 20000de4
block allocated success! block number: 1, block ptr: 200011e4
block allocated success! block number: 2, block ptr: 200015e4
block allocated success! block number: 3, block ptr: 200019e4
block allocated success! block number: 4, block ptr: 20001de4
block allocated success! block number: 5, block ptr: 200021e4
block allocated success! block number: 6, block ptr: 200025e4
block allocated success! block number: 7, block ptr: 200029e4
block allocated success! block number: 8, block ptr: 20002de4
block allocated success! block number: 9, block ptr: 200031e4
block pool no memory to be allocated!
block pool no memory to be allocated!
block release success!,block number: 0,block ptr: 00000000
block release success!,block number: 1,block ptr: 00000000
block release success!,block number: 2,block ptr: 00000000
block release success!,block number: 3,block ptr: 00000000
block release success!,block number: 4,block ptr: 00000000
block release success!,block number: 5,block ptr: 00000000
block release success!,block number: 6,block ptr: 00000000
block release success!,block number: 7,block ptr: 00000000
block release success!,block number: 8,block ptr: 00000000
block release success!,block number: 9,block ptr: 00000000
block release error!,block number: 4,block ptr: 00000000
block release error!,block number: 5,block ptr: 00000000

```

1. block内存池创建成功

2. 按下KEY2键10次,则分配10个内存块
编号0 - 9, 内存块首地址在后面显示,
两个内存块首地址相关1024字节
eg: 200015e4 - 200011e4 = 400
十六进制400即十进制1024字节

3. 再次按下KEY2键,则提示分配失败

4. 按下KEY1键4次,释放0 - 3四个内存块

5. 按下KEY0键一次,释放剩余所有的内存块

6. 再次按下KEY1,则释放内存失败(因为上面第5步中已释放所有内存块)

从上面的数值计算可以看出,对于 block 内存管理方式,一个内存池里的内存块是连续存放的,内存块大小 1K, 则两个块的起始地址间隔恰好是 1024 字节。因为内存块的大小是固定的,每次申请只能得到一个块大小,所以相对好管理,可以做到无内存碎片。

14. 内存管理(raw_byte)

本节演示 byte 内存管理方式。

首先定义内存池变量和创建内存池：

```
#define RAW_BYTE_SIZE 1024 * 10 // BYTE 内存池大小 10K
unsigned char raw_byte_pool[RAW_BYTE_SIZE]; // 静态数组,动态管理

RAW_BYTE_POOL_STRUCT byte_pool; // 定义一个内存池变量

// 创建一个内存池,大小 10K,用于管理以字节为单位申请和使用的内存
raw_byte_pool_create(&byte_pool, (RAW_U8 *) "byte_pool", raw_byte_pool,
RAW_BYTE_SIZE);
```

实验目的： 演示 byte 内存管理方式的用法

实验思路： 创建一个具有 10K 大小的内存控制块(内存池)，按键 2 按下，则申请一块内存，并打印内存块的起始地址，按键 1 按下，则释放一块内存，按键 0 按下，则释放所有的内存块。

实验关键代码：

```
// 第一个任务的执行函数入口
void First_Task(void * pParam)
{
    int ret;
    int i = 0;

    while(1)
    {
        // 阻塞等待按键 2 信号量
        raw_semaphore_get(&Key2_Semaphore, RAW_WAIT_FOREVER);

        // 按键按下,则分配内存块,分配大小 1024 * (i+1), i 按键次数
        ret = raw_byte_allocate(&byte_pool, (RAW_VOID **) &byte_ptr[i],
1024 * (i+1) );
        if ( RAW_NO_MEMORY == ret ) // 分配失败
        {
            Uart_Printf("byte pool no memory to be allocated!\r\n");
        }
        else // 分配成功
        {
            // 打印分配的内存地址
            Uart_Printf("byte allocated success! byte number: %d, byte ptr:
%p\r\n", i, byte_ptr[i]) ;
            i ++;
        }
    }
}
```

```
        if(i >= 20)
        {
            i = 0;
        }
    }
}

// 第二个任务的执行函数入口
void Second_Task(void * pParam)
{
    int i = 0;
    int ret;

    while(1)
    {
        // 阻塞等待按键 1 信号量
        raw_semaphore_get(&Key1_Semaphore, RAW_WAIT_FOREVER);
        // 释放内存
        ret = raw_byte_release(&byte_pool, (RAW_VOID *)byte_ptr[i]);

        if(RAW_SUCCESS != ret)
        {
            Uart_Printf("byte release error!,byte number: %d,byte ptr: %p\r\n", i, byte_ptr[i]);
        }
        else
        {
            byte_ptr[i] = NULL; // 释放后要置 NULL
            Uart_Printf("byte release success!,byte number: %d,byte ptr: %p\r\n", i, byte_ptr[i]);
        }

        i ++;
    }
}

// 第二个任务的执行函数入口
void Thrid_Task(void * pParam)
{
    int i;
    int ret;
```

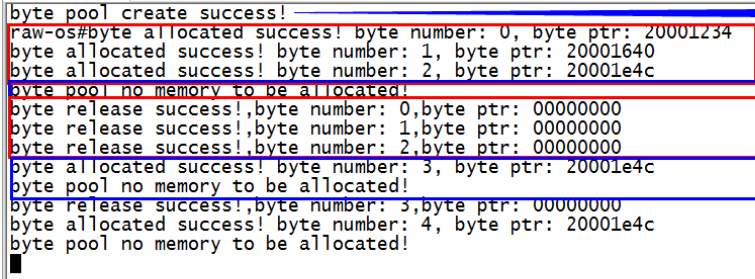
```

while(1)
{
    // 阻塞等待按键 0 信号量
    raw_semaphore_get(&Key0_Semaphore,RAW_WAIT_FOREVER);

    for(i = 0; i < 20; i ++)    // 遍历
    {
        if( byte_ptr[i] != NULL )    // 有效内存,则释放
        {
            ret = raw_byte_release(&byte_pool, (RAW_VOID
*)byte_ptr[i]);
            if(RAW_SUCCESS != ret)
            {
                Uart_Printf("block release error!,byte number: %d,byte
ptr: %p\r\n",i,byte_ptr[i]);
            }
            else
            {
                byte_ptr[i] = NULL;    // 释放后要置 NULL
                Uart_Printf("block release success!,byte number:
%d,byte ptr: %p\r\n",i,byte_ptr[i]);
            }
        }
    }
}
}

```

实验结果:



byte pool create success!

raw-os#byte allocated success! byte number: 0, byte ptr: 20001234

byte allocated success! byte number: 1, byte ptr: 20001640

byte allocated success! byte number: 2, byte ptr: 20001e4c

byte pool no memory to be allocated!

byte release success!,byte number: 0,byte ptr: 00000000

byte release success!,byte number: 1,byte ptr: 00000000

byte release success!,byte number: 2,byte ptr: 00000000

byte allocated success! byte number: 3, byte ptr: 20001e4c

byte pool no memory to be allocated!

byte release success!,byte number: 3,byte ptr: 00000000

byte allocated success! byte number: 4, byte ptr: 20001e4c

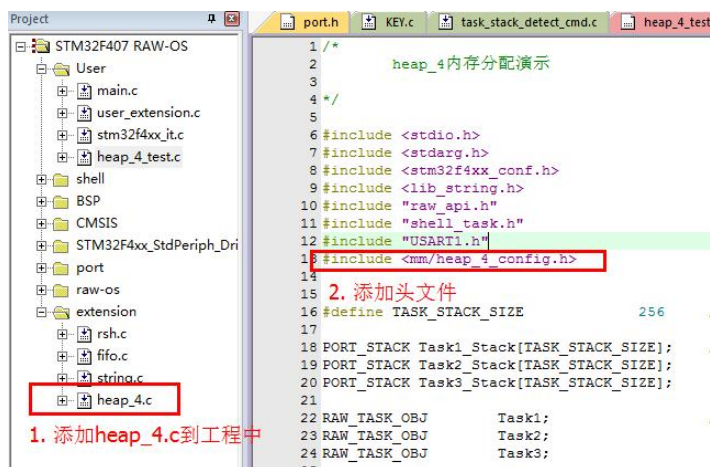
byte pool no memory to be allocated!

1. Byte内存池创建成功
2. 按下KEY2申请三次内存,大小分别是1024, 2048, 3072字节。
3. 第四次按下KEY2申请4096字节,已无内存可用
4. 按下KEY0释放所有的内存
5. 再次按下KEY2申请5K内存成功,再按申请6K,无内存可分配。

内存管理是操作系统中一个高级话题,本实验仅演示 byte 内存管理相关 API 的使用和执行效果。

15. 内存管理(heap_4)

本节演示基于 FreeRTOS 的 heap_4 内存管理使用，首先添加 heap_4.c 源文件到工程中：



其中 heap_4.c 文件在 Raw-os 源码目录下的 extension/mm 目录下面，heap_4_config.h 头文件在 Raw-os 源码目录下的 extension/include/mm 目录下，要添加头文件路径。

Heap_4 的使用就比较简单了，查看其头文件，有如下源码：

```
#define portBYTE_ALIGNMENT          4

#define configTOTAL_HEAP_SIZE      1024*10
```

指定内存地址对齐方式和内存大小，这里定义为 10K。

Heap_4 内存管理的 API 使用也相当简单：

```
void *mem_4_malloc(size_t xWantedSize);
void mem_4_free(void *pv);
size_t mem_4_free_get(void);
```

就这三个，第一个申请，第二个释放，第三个获得剩余空间。除此之外，不需要额外的定义变量和创建内存池。

实验目的： 演示 heap_4 内存管理方式的用法

实验思路： 创建一个具有 10K 大小的内存控制块(内存池)，按键 2 按下，则查看系统剩余内存，按键 1 按下则申请一块内存，大小为 1000 * i 字节,其中 i 为按键按下的次数，按键 0 按下，则释放所有内存。

(注：主要是为了查看内存分配与释放的一些细节)

实验关键代码：

```
/ 第一个任务的执行函数入口
void First_Task(void * pParam)
{
    unsigned int count;
```

```

while(1)
{
    // 阻塞等待按键 2 信号量
    raw_semaphore_get(&Key2_Semaphore, RAW_WAIT_FOREVER);
    // 获得系统剩余内存
    count = mem_4_free_get();
    Uart_Printf("free memory: %d \r\n",count);

}
}

// 第二个任务的执行函数入口
void Second_Task(void * pParam)
{
    int i = 1;

    while(1)
    {
        // 阻塞等待按键 1 信号量
        raw_semaphore_get(&Key1_Semaphore,RAW_WAIT_FOREVER);
        // 申请内存
        mem_ptr = mem_4_malloc(1000 * i);
        if(NULL == mem_ptr)
        {
            Uart_Printf("memory malloc failed!\r\n");
        }
        else
        {
            i ++;
            Uart_Printf("memory malloc success, mem_ptr:
%p\r\n",mem_ptr);
        }

    }
}

// 第三个任务的执行函数入口
void Thrid_Task(void * pParam)
{
    while(1)
    {
        // 阻塞等待按键 0 信号量
    }
}

```

```

raw_semaphore_get(&Key0_Semaphore, RAW_WAIT_FOREVER);
mem_4_free(mem_ptr);
mem_ptr = NULL;

}
}

```

实验结果:

free memory: 10236	1. 首先按下KEY2键查看剩余内存
memory malloc success, mem_ptr: 20001fe4	2. 按下KEY1键申请内存1000字节
free memory: 9220	3. 按下KEY2键查看剩余内存,注意消耗多少?
free memory: 10228	4. 按下KEY0释放内存,再按KEY1查看剩余内存!
memory malloc success, mem_ptr: 20001fe4	5. 按下KEY1再次申请内存
free memory: 8220	6. 按下KEY2查看剩余,然后按KEY0释放,再按KEY2查看剩余
free memory: 10228	7. 按下KEY1申请两次,再按则申请5000字节,申请失败! 查看剩余则只有3212字节!
memory malloc success, mem_ptr: 20001fe4	
memory malloc success, mem_ptr: 20002ba4	
memory malloc failed!	
free memory: 3212	
free memory: 3212	
free memory: 3212	

计算: 分配10240字节大小内存,只有10236字节可用, 第一次申请1000字节,系统消耗了10236 - 9220=1016, 释放后系统回收10228-9220=1008字节, 再次申请2000字节则系统消耗10228-8220 = 2008字节,再次释放,则系统回收2008字节。由此可见第一次申请内存会额外的消耗一些内存,并且不能被系统回收(8字节), 第二次申请的内存能被系统全部回收, 且每次申请内存都会有额外的8字节消耗!

从上面实验结果可以看出内存管理, 申请与释放的细节, 请读者自行研究。

16. 事件(event_or)

当一个任务需要和多种事件同步的时候可以使用事件标志来同步。一组事件标志有 32 个 bit。一个任务可以同步任何一个事件的发生，这个时候就叫事件或的机制，也可以同步所有事件的发生，这个时候叫事件与的机制。

理论上 raw os 可以无限的创建事件标志，但是实践上内存制约了这一点。事件标志的代码在 raw_event.c。

假设一个任务需要等待几个事件标志位满足才能继续运行，这种关系叫做与(and)，假如一个任务等待几个事件标志位的任何一个满足条件，就能运行，这种关系叫做或(or)的关系。

事件标志和信号量最大的区别是，信号量内部有一个累加器，但是事件标志是没有的，如果满足事件标志了就相当于状态机的状态转换。

本节演示事件或的机制，event 用起来相对简单的多，主要 API 就三个：创建，设置事件，获取事件。代码如下：

```
RAW_EVENT Key_Event; // 定义一个事件,用于按键与任务同步
// 创建一个事件,事件的 32 个 BIT 初始化为 0
raw_event_create(&Key_Event, (RAW_U8 *) "KEY_EVENT", 0);
```

设置事件：

```
// 设置事件组的 BIT2
raw_event_set(&Key_Event, 0x2, RAW_OR);
```

获取事件：

```
// 获取事件的 BIT3 或 BIT2,有任一满足返回,事件的值存储到 event_flag 中
raw_event_get(&Key_Event, 0x6, RAW_OR_CLEAR, &event_flag,
RAW_WAIT_FOREVER);
```

实验目的： 演示 Event OR 的用法

实验思路： 创建一个事件，三个按键按下时分别置位 BIT3，BIT2，BIT1，三个任务阻塞到这个事件上，当按键按下时，相应的任务被唤醒。

注意： BIT 置位时，可能会唤醒不止一个任务，所有满足关系的任务都会得到唤醒。

实验关键代码：

```
// 第一个任务的执行函数入口
void First_Task(void * pParam)
{
    while(1)
    {
        // 阻塞等待事件的 BIT2,如果事件的 BIT2 置位(按键 2 按下),则此任务会被唤醒
        raw_event_get(&Key_Event, 0x2, RAW_OR_CLEAR, &event_flag,
RAW_WAIT_FOREVER);
```



```
    LED_Toggle(2);

    Uart_Printf("Task1 Receive Event BIT2,the event flag Value is:
0X%X\r\n",event_flag);
}
}

// 第二个任务的执行函数入口
void Second_Task(void * pParam)
{
    while(1)
    {
        // 阻塞等待事件的 BIT1,如果事件的 BIT1 置位(按键 1 按下),则此任务会被唤醒
        raw_event_get(&Key_Event, 0x1, RAW_OR_CLEAR, &event_flag,
RAW_WAIT_FOREVER);

        LED_Toggle(1);

        Uart_Printf("Task1 Receive Event BIT1,The Event Flag Value is:
0X%X\r\n",event_flag);
    }
}

// 第二个任务的执行函数入口
void Thrid_Task(void * pParam)
{
    while(1)
    {
        // 阻塞等待事件的 BIT3 和 BIT2,如果两个 BIT 任一个置位(按键 0 或按键 2 按下),
        则任务会被唤醒
        raw_event_get(&Key_Event, 0x6, RAW_OR_CLEAR, &event_flag,
RAW_WAIT_FOREVER);

        LED_Toggle(1);
        LED_Toggle(2);

        Uart_Printf("Task3 Receive Event BIT3 Or BIT2,The Event Flag Value
is: 0X%X\r\n",event_flag);
    }
}
```

实验结果:

Task1 Receive Event BIT2,the event flag Value is: 0x2
Task3 Receive Event BIT3 Or BIT2,The Event Flag Value is: 0x2
Task2 Receive Event BIT1,The Event Flag Value is: 0x1
Task2 Receive Event BIT1,The Event Flag Value is: 0x1
Task2 Receive Event BIT1,The Event Flag Value is: 0x1
Task3 Receive Event BIT3 Or BIT2,The Event Flag Value is: 0x4
Task3 Receive Event BIT3 Or BIT2,The Event Flag Value is: 0x4

- 1. 按键2按下,置位BIT2唤醒任务1和任务3
- 2. 按键1按下,置位BIT1唤醒任务2
- 3. 按键0按下,置位BIT3,唤醒任务3

再次注意: 条件满足时所有的任务都会被唤醒执行

17. 事件(event_and)

事件的与和或所用的 API 一样，只是标志位不一样，对于与的用法如下。

设置事件：

```
// 设置事件组的 BIT2
raw_event_set(&Key_Event, 0x2, RAW_OR);
```

获取事件：

```
// 获取事件的 BIT3 和 BIT2,两个 BIT 同时置位时,条件才成立,函数才会唤醒否则阻塞
raw_event_get(&Key_Event, 0x6, RAW_AND_CLEAR, &event_flag,
RAW_WAIT_FOREVER);
```

本实验的源码在 event set 部分和上节一样，在 event get 部分的条件为 RAW_AND_CLEAR。

实验的效果如下：

1. 按下KEY2,置位BIT2唤醒任务1
2. 按下KEY1,置位BIT1唤醒任务2
3. 注意! 先按下KEY0置位BIT3,然后再按KEY2置位BIT2,此时任务1和任务3同时被唤醒!!!
4. 再按下KEY2,此时BIT2置位只会唤醒任务1!!!
5. 再按KEY0置位BIT3,然后再按KEY2置位BIT2,此时任务1和任务3均被唤醒!!!

总结：对于“与”来讲,任务3监控两个位BIT3和BIT2,只有当两个位均被置位时,才会唤醒,而且唤醒后清零应标置位,所以再次按KEY2置位BIT2此时只会唤醒任务1

对于 Event 的 AND 和 OR 机制，读者细心研究例程代码，比较异同。

18. 统计 CPU 利用率

在 RAW-OS 系统中，系统预定义的任务有：Task_0, Tick_Task, Timer_Task, Idle_Task, CPU_Task 这五个。其中在 raw_config.h 中打开第一个宏 CONFIG_RAW_ZERO_INTERRUPT 时，同时也要打开 CONFIG_RAW_TASK_0 宏，这样系统中就有一个预定义的优先级为 0 的任务 Task_0，它的源码在 raw_task_0.c 中。

Tick_Task 即专门用于系统时钟更新的任务，其优先级为 1，如果系统没有开 Task_0 也没有开 Tick_Task，那么时钟更新 Tick_Update 则在 SysTick_Handler 中，这样会影响到实时性，所以建议开启 Tick_Task。

如果使用软件定时器，那么软件定时器的相关处理在 Timer_Task 任务中，其预定义的优先级为 5，在 raw_config.h 中可以自由配置其优先级和堆栈大小。

Idle_Task 即空闲任务了，是系统必须开启的一个任务。

CPU_Task 即统计任务，用于统计 CPU 利用率，其配置宏如下：

```
#define RAW_CONFIG_CPU_TASK      1
#define CPU_TASK_PRIORITY        (CONFIG_RAW_PRIO_MAX - 2)
#define CPU_STACK_SIZE          256
```

第一项 1 表示开启 CPU_Task，第二项表示它的优先级仅高于 Idle_Task，第三项表示它的栈是 256。

在 raw_os_init() 中有如下代码：

```
#if (RAW_CONFIG_CPU_TASK > 0)
cpu_task_start();
#endif
```

即是开启了 RAW_CONFIG_CPU_TASK 宏后，系统会创建 CPU_Task。

需要注意的是，用户还必须在启动代码中自己调用 cpu_task_init() 函数才行，这个函数通常放在配置完 OS 时钟之后：

```
OS_CPU_SysTickInit();           // 配置系统时钟
cpu_task_init();                // CPU统计任务初始化
```

在 shell 中运行 stack 命令可以看到有 cpu_object 即为统计任务的名子。

```
raw-os#stack
task name is      idle_task *** task free stack size is 206
task name is      timer_object *** task free stack size is 68
task name is      cpu_object *** task free stack size is 201
task name is      Shell Task *** task free stack size is 49

raw-os#
```

对于统计任务的使用就是 开启宏，调用初始化函数两步，然后系统提供给用户两个变量：cpu_usage 和 cpu_usage_max，分别表示实时的 CPU 利用率和最大的 CPU 利用率。这两个变量是 CPU 利用率扩大 100 倍后的值，采用如下方式打印出来：

```
// 打印 CPU 利用率和最大 CPU 利用率
```

```
Uart_Printf("CPU Usage: %d %% CPU Max Usage: %d %%\r\n",cpu_usage,cpu_usage_max);
```

本例程向系统中注册 `cpuusage` 命令，在 shell 中输入此命令，查看实时的值和最大的值。

实验结果：

```
raw-os#cpuusage
CPU Usage: 2 % CPU Max Usage: 3 % 实时的CPU利用率和最大的CPU利
                                     用率
raw-os#cpuusage
CPU Usage: 2 % CPU Max Usage: 6 %
raw-os#cpuusage
CPU Usage: 2 % CPU Max Usage: 6 %
```

猛敲键盘，然后再次输入命令查看：

```
raw-os#
raw-os#cpuusage
CPU Usage: 2 % CPU Max Usage: 10 % CPU最大利用率增大
raw-os#
raw-os#cpuusage
CPU Usage: 2 % CPU Max Usage: 10 %
```

19. 内核扩展模块 FIFO 的使用

FIFO 即环形缓冲区，是只能在一端写数据，在另一端读的结构。是一种使用极为广泛的数据结构，Raw-os 提供了一个通用 FIFO 模块作为内核的扩展模块，放在 extension 目录下的 lib 子目录中，实现文件为 fifo.c 或 fifo_lock.c，两个的区别在于后者在读写 FIFO 过程中加入了锁机制，这样对多任务环境下操作更安全。

FIFO 具体的原理和实现请参见本文档的附录四 RAW-OS FIFO 机制分析，原理讲解和源码注释都非常清楚。

本例使用 FIFO 重写 Shell 的底层 USART 驱动，之前的思路是：当用户在终端中输入一个字符时，触发 USART 的接收中断，在中断中释放信号量去同步 Shell 任务，然后读取输入的字符存到缓冲区中。如果收到的字符是回车符，则认为命令输入结束并进行解析执行。

使用 FIFO 的思路：创建一个 FIFO，用户输入一个字符触发 USART 中断，读取这个输入的字符判断它是不是回车符，如果不是，则把字符压入 FIFO 中；如果是回车符，则释放信号量去同步 Shell 任务去读取 FIFO 中的内容(命令字符串)，然后解析执行。

USART 接收端相关代码如下：

```
/* 接收这个字符,如果不是回车符,则压入 FIFO,如果是则同步 Shell_Task */
ch = USART_ReceiveData(USART1);

if(ch == ENTER_KEY)                // 输入是回车符
{
    // 同步 Shell_Task 运行
    raw_semaphore_put(&USART1_Receive_Semaphore);
}
else if(ch == BACKSPACE_KEY)        // 输入是 backspace
{
    /* 本例中由于 FIFO 特性,不易处理输入 backspace key 的情况,
       故此处无处理,仅为演示 Raw-os 内核扩展模块 FIFO 的使用,
       用户若输入错误,回车重新输入即可.*/
}
else
{
    fifo_in(&USART1_FIFO,&ch,1);    // 压 ch 到 FIFO 中
    USART1_Send_Char(ch);          // 终端回显
}
```

Shell 任务中读取 FIFO 的相关代码如下：

```
/* 等待信号量(永远阻塞,直到终端输入回车符后唤醒) */
raw_semaphore_get(&USART1_Receive_Semaphore,RAW_WAIT_FOREVER);
```

```
// print a enter
USART_Send_Enter();

// 从FIFO中读出命令字符串
len = fifo_out_all(&USART1_FIFO, (void *)input_buffer);
if( len )
{
    input_buffer[len] = '\\0';

    // 清空输出缓冲区
    raw_memset(output_buffer,0,1024);

    do
    {
        // 命令处理
        ret = rsh_process_command((const RAW_S8
*)input_buffer,output_buffer,1024);

        // 打印处理结果
        USART_Send_String((RAW_U8 *)output_buffer);

        USART_Send_Enter();    // 回册换行

    }while(ret == 0);
}
```

20. 多对象阻塞 1

细心的读者会发现，在前面的例程中，都是任务阻塞在一个内核对象上，比如，在例程 4 中 shell 任务阻塞一个信号量上，在例程 7-9 中任务阻塞在一个队列上，这些都是最简单的同步与通信方式。但是，当系统变的复杂或应用程序的逻辑处理复杂的时候，那一个任务可能就不只是阻塞在一个信号量或一个队列上了，这就引出了本节所要讲述的多对象阻塞

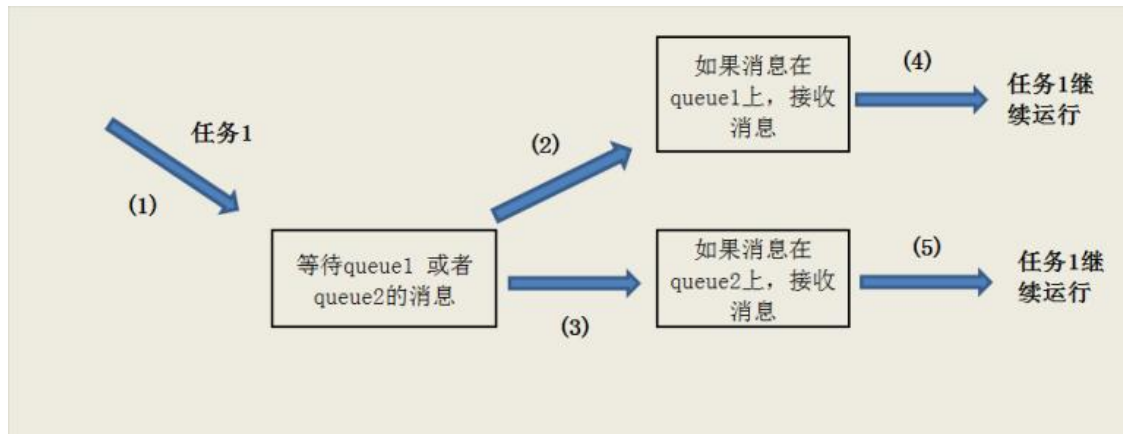
本节和下节会阐述单个任务如何阻塞在多个内核对象上。目前 raw os 支持任务同时阻塞在队列(queue) 或者信号量 (semaphore) 上。

raw os 实现多对象阻塞的功能主要是通过函数通知的功能来实现的。通过此项技术能够简洁的实现单任务阻塞在多个内核对象上的功能。函数通知的功能能够通过注册一个回调函数，使得释放信号量以及发送消息到队列上的时候调用这个注册的函数。怎么样注册一个函数通知功能的回调函数呢？

举例如下：

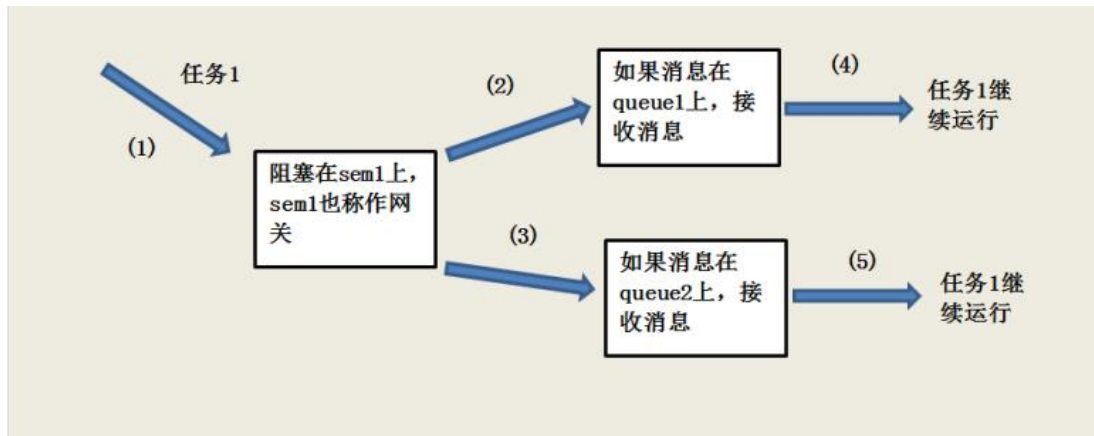
`raw_semaphore_send_notify(&sem1, notify_function)` 通过此函数调用可以注册一个回调函数 `notify_function` 到信号量 `sem` 上，当用户使用函数 `raw_semaphore_put` 的时候就可以触发此回调函数。

同样对于队列也有此功能。`raw_queue_send_notify(&queue1, notify_function)` 通过此函数可以注册一个回调函数 `notify_function` 到 `queue1` 上。当用户使用函数 `raw_queue_end_post` 时会触发此注册的回调函数。



上图(1)处任务 1 需要等待 queue1 或者 queue2 有消息才能继续运行。queue1 或者 queue2 没有消息的话，任务 1 就阻塞住。(2)和(3)处说明 queue1 或者 queue2 来消息了，任务 1 需要去接收，(4)和(5)处说明任务 1 接收了消息后继续运行。下图演示的是 raw os 对于上面问题的解决方式，不同的操作系统有不同的解决方式，无疑通过函数通知这种实现是最简洁的。

上图(1)处任务 1 阻塞在网关处当 `sem1` 的内部计数器值为 0 时说明 queue1 和 queue2 都没有消息，当不为 0 时说明 queue1 或者 queue2 有消息。(2)和(3)处说明 queue1 或者 queue2 来消息了，任务 1 需要去接收，(4)和(5)处说明任务 1 接收了消息后继续运行。



以上过程只是比前面一个图多了一个 sem1 的网关, 却是解决问题的关键所在。

下面总结描述下解决方案。

1. Queue1 以及 queue2 上注册一个通知函数, 通知函数的内容是发送一个信号量。

2 任务阻塞在 sem1 上, 当 queue1 以及 queue2 获得消息时, 会唤醒任务 1, 然后接收相应的消息。

具体的代码演示如下: queue1 以及 queue2 上注册一个通知函数 notify_queue, 代码如下所示:

```
raw_queue_send_notify(&queue1, notify_queue);
raw_queue_send_notify(&queue2, notify_queue);
```

通知函数内容如下:

```
void notify_queue(RAW_QUEUE *ptr)
{
    raw_semaphore_put(&sem1);
}
```

可以看到只要 queue1 和 queue2 里面一有消息就会触发 notify_queue 回调函数。回调函数会释放一个信号量, 同时会打开网关 sem1, 进而唤醒任务 1。

任务 1 的代码如下:

```
raw_semaphore_get(&sem1, RAW_WAIT_FOREVER);

ret = raw_queue_receive (&queue1, RAW_NO_WAIT, (RAW_VOID
**) &msg_app2);

if (ret == RAW_SUCCESS)
{
    /*Access queue1 msg*/;
}
```

```
ret = raw_queue_receive (&queue2, RAW_NO_WAIT, (RAW_VOID
**)&msg_app2);

if (ret == RAW_SUCCESS)
{
    /*Access queue2 msg*/;
}
```

上述代码中可以看到任务 1 阻塞在 sem1 上，当 queue1 以及 queue2 上有消息的时候会触发 notify_queue 这个回调函数，回调函数内会释放一个信号量，进而唤醒任务 1。

需要注意的是调用 raw_queue_receive 时需要采用没有消息立马返回的功能，即超时参数为 RAW_NO_WAIT。

关于多对象阻塞的更深入理解请研读本节和下节的源码，本节例程是一个任务同时阻塞在三个不同的消息队列上。下节例程是一个任务同时阻塞在一个信号量和消息队列上。

实验目的： 演示多对象阻塞的用法

实验思路： 创建一个任务阻塞在多个内核对像上，在本节例程中是阻塞在三个不同的消息队列上，当不同的按键按下时发送不同的消息到这三个队列，队列绑定的回调函数得到执行，任务被唤醒，然后判断具体是哪个队列得到了消息，并打印结果。

实验关键代码：

```
// RAW_QUEUE 测试,创建一个任务阻塞在三个队列上
void MultiPend_Test()
{
    // 1. 创建三个队列,任务阻塞在这三个队列上
    raw_queue_create(&Queue_1, (RAW_U8 *)"queue1", (RAW_VOID
**)&multi_queue1, 10);
    raw_queue_create(&Queue_2, (RAW_U8 *)"queue2", (RAW_VOID
**)&multi_queue2, 10);
    raw_queue_create(&Queue_3, (RAW_U8 *)"queue3", (RAW_VOID
**)&multi_queue3, 10);

    // 2. 创建一个信号量用作通知任务的网关
    raw_semaphore_create(&GateWay_Sem, (RAW_U8 *)"GateWay_Sem", 0);

    // 3. 注册每个队列的回调函数,当队列中有数据时会调用该回调函数
    raw_queue_send_notify(&Queue_1, notify_queue_1);
    raw_queue_send_notify(&Queue_2, notify_queue_2);
    raw_queue_send_notify(&Queue_3, notify_queue_3);
```

```
// 3. 创建消息处理任务
raw_task_create(&Task_MultiPend, (RAW_U8 *)"Task_Multi", 0,
               10, 0, Task_MultiPend_Stack,
               TASK_STACK_SIZE, Task_MultiPend_Fun, 1);
}

// 队列 1 回调函数
void notify_queue_1(RAW_QUEUE *ptr)
{
    raw_semaphore_put(&GateWay_Sem);
}

// 队列 2 回调函数
void notify_queue_2(RAW_QUEUE *ptr)
{
    raw_semaphore_put(&GateWay_Sem);
}

// 队列 3 回调函数
void notify_queue_3(RAW_QUEUE *ptr)
{
    raw_semaphore_put(&GateWay_Sem);
}

// 任务执行函数
void Task_MultiPend_Fun(void * pParam)
{
    RAW_U16 res;
    signed char *msg;

    while(1)
    {
        // 等待消息(永久)
        raw_semaphore_get(&GateWay_Sem, RAW_WAIT_FOREVER);
        /* 判断是哪一个队列中有消息 */
        // 如果 Queue_1 中有消息则读取, 如果 无, 则立即返回
        res = raw_queue_receive(&Queue_1, RAW_NO_WAIT, (RAW_VOID
**)&msg);

        if (res == RAW_SUCCESS) // Queue_1 中得到消息
        {

        }

        else // Queue 2 或 Queue 3 中得到消息
```

```

{
    res = raw_queue_receive(&Queue_2, RAW_NO_WAIT, (RAW_VOID
**)&msg);
    if (res == RAW_SUCCESS)      // Queue_2 中得到消息
    {
    }
    else                          // Queue_3 中得到消息
    {
        res = raw_queue_receive(&Queue_3, RAW_NO_WAIT, (RAW_VOID
**)&msg);
    }
}

USART1_Send_String(msg);          // 打印消息
USART1_Send_String((RAW_S8 *)"\r\n");
// 根据消息内容点灯
if( 0 == raw_strncmp((const char *)msg, "KEY0", 4) )
{
    LED_Toggle(1);
}
else if(0 == raw_strncmp((const char *)msg, "KEY1", 4) )
{
    LED_Toggle(2);
}
else if(0 == raw_strncmp((const char *)msg, "KEY2", 4) )
{
    LED_Toggle(1);
    LED_Toggle(2);
}
}
}

```

实验结果:

serial-com3 x

KEY2
KEY2
KEY1
KEY1
KEY0
KEY0

三个按键按下,则发送不同的消息到三个不同的队列中,
任务阻塞在三个队列上,任何一个按键按下都会唤醒任务去读取消息,
然后打印出消息内容。

21. 多对像阻塞 2

上一小节中是一个任务阻塞在了不同的消息队列上，本节演示一个任务同时阻塞在信号量和消息队列上。

实验目的： 演示多对像阻塞的用法

实验思路： 创建一个任务阻塞在多个内核对像上，在本节例程中是阻塞在一个消息队列和一个信号量上，当 **KEY2** 按下时发送消息到队列中，当 **KEY1** 按下时释放一个信号量，这两个同步信息最终都会触发一个“网关”信号量来同步任务，然后任务再从消息队列中读取消息或信号量。

实验关键代码：

和上节实验类似，具体请参照实验源码。

实验效果：

按下 **KEY2** 和 **KEY1** 分别产生不同的同步信号，但都会唤醒任务的执行，从而控制不同的 LED 工作。

22. 中断下半部 work queue(工作队列)

Raw-os 提供独特的中断下半部机制，其实现方法有多种，具体原理与源码分析请参考[附录五、RAW-OS 中断下半部和 work_queue 机制分析](#)。其中分析了 raw-os 中断下半部的原理和具体的 work queue 实现方法。在这里做出实验说明。

实验目的：演示中断下半部 work queue 用法

实验思路：首先创建一个工作队列用来，初始化一个定时器产生 500ms 的中断，在中断中模拟产生一个 32 位的整形数和一条数据消息，并把整形数和指向数据消息的指针作为参数传递给工作队列，工作队列调用用户传入的回调函数来处理数据。

实验关键代码：

```
#define WORK_QUEUE_TASK_STACK_SIZE    512

// 定义一个工作队列,申请工作队列所需栈空间
PORT_STACK Work_Queue_Task_Stack[WORK_QUEUE_TASK_STACK_SIZE];
WORK_QUEUE_STRUCT Work_Queue_Task;

// 定义工作队列存储空间
RAW_VOID *Work_Queue_Msg[10];
OBJECT_WORK_QUEUE_MSG work_queue_msg[100];

// 定义一个用户回调函数,它在 work queue task 中调用,用于数据处理
void My_Func(RAW_U32 arg, void *msg)
{
    LED_Toggle(1);
    LED_Toggle(2);

    // 这里打印数据来模拟数据的处理
    Uart_Printf("%d ",arg);           //打印传参
    Uart_Printf("%s\r\n",(char *)msg); //打印传参
}

// 以下变量用来模拟中断中采集到的数据
unsigned char msg[] = "hello world! msg count: ";
unsigned int count = 0;

// 下面函数在定时器中断中被调用,模拟在中断中采集到数据,然后传送给 work queue 处理
void Data_Produce_simulate()
{
    /* 模拟采集到数据 */
    count ++;
    if(count > 999)count = 0;
```

```

msg[sizeof(msg) - 4] = count/100 + 0X30;
msg[sizeof(msg) - 3] = count%100/10 + 0X30;
msg[sizeof(msg) - 2] = count%10 + 0X30;

/* 触发work queue并传入数据count和msg,数据处理函数为My_Func */
sche_work_queue(&Work_Queue_Task, count, msg, My_Func);
}

// raw-os 的中断下半部模块 Work_Queue 测试
void WorkQueue_Test()
{
    // 初始化work queue system
    global_work_queue_init(work_queue_msg, 100);

    // 创建一个work queue
    work_queue_create(&Work_Queue_Task, 6,
                     WORK_QUEUE_TASK_STACK_SIZE,
                     Work_Queue_Task_Stack,
                     (RAW_VOID **)Work_Queue_Msg, 10);

    // 初始化硬件,在硬件中断里产生数据并触发中断下半部执行
    TIM3_Int_Init(4999, 8399);
}

```

实验效果:

首先可以看到两个 LED 在闪烁，然后串口终端显示如下。

```

1 hello world! msg count: 001
raw-os#2 hello world! msg count: 002
3 hello world! msg count: 003
4 hello world! msg count: 004
5 hello world! msg count: 005
6 hello world! msg count: 006
7 hello world! msg count: 007
8 hello world! msg count: 008
9 hello world! msg count: 009
10 hello world! msg count: 010
11 hello world! msg count: 011
12 hello world! msg count: 012
13 hello world! msg count: 013
14 hello world! msg count: 014
15 hello world! msg count: 015
16 hello world! msg count: 016
17 hello world! msg count: 017
18 hello world! msg count: 018
19 hello world! msg count: 019
20 hello world! msg count: 020
21 hello world! msg count: 021
22 hello world! msg count: 022
23 hello world! msg count: 023
24 hello world! msg count: 024
25 hello world! msg count: 025
26 hello world! msg count: 026

```

前面的整数是中断传递给work queue的整型参数，后面的字符串是中断传递给work queue的消息指针所指向的内容。

附录一、解析 RAW-OS 内核中的链表

raw_os 内核中采用了一套通用的、一般的、可以用到各种不同数据结构的链表操作。即是把链表前向指针 `previous` 和后向指针 `next` 从具体的“宿主”数据结构中抽象出来成为一种通信数据结构 `LIST`，这种数据结构既可以“寄宿”在具体的宿主数据结构内部，成为该数据结构的一个“连接件”；也可以独立存在而成为一个队列的头。

在 `raw_list.h` 中定义了链表的数据结构，并实现了通用的插入和删除操作。

1. 通用链表数据类型定义如下，它仅定义了一个前向指针和一个后向指针。

```
typedef struct LIST {
    struct LIST *next;
    struct LIST *previous;
} LIST;
```

2. 链表如始化代码如下，因为是双向循环链表，所以两个指针都指向了链表头结点。

```
RAW_INLINE void list_init(LIST *list_head)
{
    list_head->next = list_head;
    list_head->previous = list_head;
}
```

3. 链表判空。

```
RAW_INLINE RAW_BOOLEAN is_list_empty(LIST *list)
{
    return (list->next == list);
}
```

3. 链表结点插入。将 `element` 结点插入到以 `head` 结点前面（仅需调整指针指向）。

```
RAW_INLINE void list_insert(LIST *head, LIST *element)
{
    element->previous = head->previous;
    element->next = head;

    head->previous->next = element;
    head->previous = element;
}
```

4. 链表结点删除。删除 `element` 结点（仅需调整两个指针指向）。

```
RAW_INLINE void list_delete(LIST *element)
{

```



```

element->previous->next = element->next;
element->next->previous = element->previous;
}

```

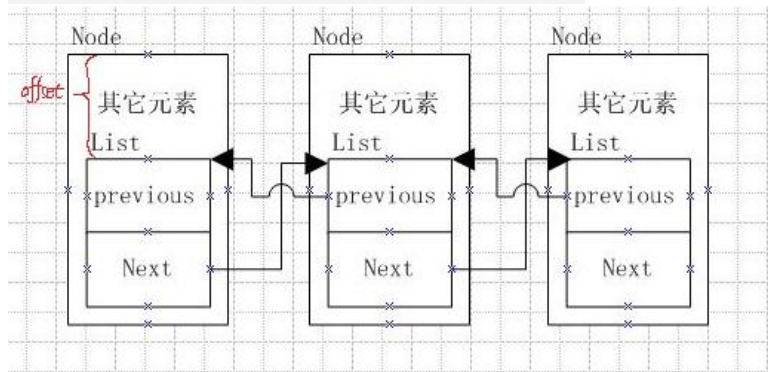
5. 获取宿主结构体指针。

```

#define list_entry(node, type, member) ((type *)(((RAW_U8 *) (node) -
(RAW_U32)&((type *)0)->member)))

```

上面这种双向循环链表的应用模型如下：



把上面的代码分开来理解，先说 `(RAW_U32)&((type *)0)->member`

`(type *) 0` 将 0 地址强制转化为 `type` 类型

`&(type *)0->member` 为取得 `type` 类型中 `member` 成员的地址

`(RAW_U32)&((type *)0)->member` 将地址转换为 `RAW_U32` 型，事实上它就是 `offset` 大小，

再看 `(type *)(((RAW_U8 *) (node)`

`node` 为指向 `member` 成员的指针，即 `list` 类型的结点，它是嵌入到宿主结构体中的，那 `node` 地址减去偏移量 `offset` 即得宿主结构体的起始地址，当然，前提是在内存中的存储是从低地址到高地址增长的。

`(RAW_U8 *) (node) - (RAW_U32)&((type *)0)->member` 已经是宿主结构体的起始地址，再对它进行一下强制类型转换 `(type *)` 就得到宿主结构体类型了。

附录二、RAW-OS Shell 机制分析

RAW-OS Shell 使用一种简单、易用、高效的机制使得应用程序支持命令行输入。系统默认支持一个“help”命令，如果有的命令加入，则 RAW-OS 建立维护一个单向链表，新加入的命令依次挂 help 这个链表头后面，当在应用层面输入命令时，执行 shell 解析的任务会读取这个命令，并通过 `rsh_process_command()` 函数从单链表头开始查找这个命令是否已注册（挂接到命令链表），每个命令会对应一个可执行函数，如果查找到该命令，则跳转到这个命令所对应的可执行函数中去执行。

下面从加入一个自定义命令的角度来分析一下这种 shell 实现的机制。

1. 定义一个新的命令

首先分析下一个命令所对应的数据结构类型，`rsh.h` 文件中定义了定义一个命令所遵循的数据结构类型。

```
typedef struct xCOMMAND_LINE_INPUT
{
    const RAW_S8 * const pcCommand;

    const RAW_S8 * const pcHelpString;

    const pdCOMMAND_LINE_CALLBACK pxCommandInterpreter;

    RAW_S8 cExpectedNumberOfParameters;
} xCommandLineInput;
```

// 上面的数据结构中第一项 `pcCommand` 即为在执行应用时要输入的命令，它应是小写字母组成。

// 第二项是这个命令对应的使用说明。

// 第三项是命令行对应的回调函数，它实际上是一个函数指针，输入命令执行后跳转到函数的地址处执行。

// 第四项是该命令支持的参数个数。

其中对于第三项函数指针 `pdCOMMAND_LINE_CALLBACK` 类型定义如下：

```
typedef RAW_S32 (*pdCOMMAND_LINE_CALLBACK) ( RAW_S8 *pcWriteBuffer,
size_t xWriteBufferLen, const RAW_S8 * pcCommandString );
```

`RAW_S32` 为函数返回类型，`pdCOMMAND_LIN_CALLBACK` 为指针变量名。

`pcWriteBuffer`: 存储命令执行的输出结果

xWriteBufferLen: pcWriteBuffer 长度

pcCommandString: 用户输入的整个命令行（参数从此命令行中提取）

eg: 添加一个打印 MCU ID 的命令

```
static const xCommandLineInput xMCU_ID_PrintCommand =
{
    "MCU_ID_Print",           // 命令名称
    "MCU_ID_Print  no parameter:", // 命令使用说明
    MCU_ID_Read,             // 执行命令的函数指针
    0                        // 无参数
};
```

命令对应的函数实现如下:

MCU_ID_Read()

XXXXXXXXXXXXXXXXXXXX

定义完命令后还需定义一个数据链表的结点,以使该命令以链表结点的形式挂接在系统命令链表上(后面会介绍)。

```
static xCommandLineInputListItem pxNewListItem;
```

2. 注册这个新的命令

上面定义了一个新的命令,接下来将该命令注册(实际上添加进命令链表)。

注册命令用到的函数如下:

```
RAW_VOID rsh_register_command(const xCommandLineInput * const
pxCommandToRegister, xCommandLineInputListItem *pxNewListItem)
{
    if (pxCommandToRegister == 0) {           //参数检查
        RAW_ASSERT(0);
    }

    if (pxNewListItem == 0) {                 //参数检查
        RAW_ASSERT(0);
    }

    pxNewListItem->pxCommandLineDefinition = pxCommandToRegister;

    pxNewListItem->pNext = 0;
```

```

current_command_list_point->pNext = pxNewListItem;

current_command_list_point = pxNewListItem;
}

```

// 第一个参数即为上面定义的命令（变量）

// 第二个参数是定义链表的结点（一个命令作为一个结点挂接在一个单向链表上）

在 RAW-OS 的 Shell 机制中，所有的命令组成一个单向的链表，命令注册即是把新定义的命令添加到这个链表中。xCommandLineInputListItem 数据类型如下：

```

typedef struct xCOMMAND_INPUT_LIST
{
    const xCommandLineInput *pxCommandLineDefinition;
    struct xCOMMAND_INPUT_LIST *pNext;
} xCommandLineInputListItem;

```

对于一个链表的结点来说，即有所谓的数据域和指针域，上面结构体中的 pxCommandLineDefinition 即为数据域变量，是一个命令所遵循的结构类型。pNext 即为指针域，指向链表中的下一个结点。

那对于这个注册函数来说，首先是将欲添加的命令结构体 pxCommandToRegister 赋值给新建的链表结点的数据域；然后将该结点的指针域指向 0；最后将该命令挂接在链表的尾部。

系统定义了一个指向链表尾部结点的指针，新添加的命令总是挂接在链表的尾部，并将这个指针指向新挂接的命令。该指针定义如下：

```

static xCommandLineInputListItem *current_command_list_point =
&xRegisteredCommands;

```

上面代码是定义了一个指针，并初始化该指针指向链表的头结点—help 命令。因为系统默认只有一个 help 命令，所以此时该指针仍是指向最后一个命令的。

Help 命令定义如下：

```

/* The definition of the "help" command. This command is always at the
front
of the list of registered commands. */
static const xCommandLineInput xHelpCommand =
{
    "help",
    "help: Lists all the registered commands\r\n",

```

```

    rsh_help_command,
    0
};

```

Help 命令结点（亦即命令链表的头结点）定义如下：

```

/* 定义一个命令链表，所有的新注册命令均挂接在这个链表上 */
static xCommandLineInputListItem xRegisteredCommands =
{
    &xHelpCommand,      /* The first command in the list is always the help
                           command, defined in this file. */
    0                   /* The next pointer is initialised to NULL, as there
                           are no other registered commands yet. */
};

```

3. 执行命令

当启动系统时，首先启动 shell_Task，在 shell_Task 中完成一些硬件初始化工作，然后依次注册各个命令，然后改变 Task 的优先级使其最低，并且让这个系统阻塞在消息队列上，如果有命令输入则唤醒这个任务，并执行命令解析函数来完成命令的解析执行。其实现的关键为命令处理函数，代码如下解释。

/* 定义一个指针，指向当前正在检索的链表结点 */

```

static const xCommandLineInputListItem *current_process_command_point;

// 命令执行函数
RAW_S32 rsh_process_command( const RAW_S8 * const pcCommandInput, RAW_S8
*pcWriteBuffer, size_t xWriteBufferLen )
{
    RAW_S32 xReturn = 0;
    const RAW_S8 *pcRegisteredCommandString;
    RAW_U8 s_input_length;
    RAW_U8 s_command_length;
    RAW_U8 s_commpare_length;

    if (pcCommandInput == 0) {                                // 参数检查
        RAW_ASSERT(0);                                         // 死循环，注意！
    }

    if (pcWriteBuffer == 0) {                                  // 参数检查
        RAW_ASSERT(0);
    }

    if (xWriteBufferLen == 0) {                                 // 参数检查
        RAW_ASSERT(0);
    }
}

```

```
}

if (current_process_command_point == 0) {
    // 从头结点遍历链表
    for (current_process_command_point = &xRegisteredCommands;
current_process_command_point != 0; current_process_command_point =
current_process_command_point->pNext) {

        // 获得当前结点所对应的命令名子
        pcRegisteredCommandString =
current_process_command_point->pxCommandLineDefinition->pcCommand;
        // 得到命令链表中当前结点所对应命令的长度
        s_command_length = command_length_get((const RAW_S8
*)pcRegisteredCommandString);
        // 计算当前输入命令的长度
        s_input_length = command_length_get((const RAW_S8
*)pcCommandInput);

        // 命令匹配的字符长度
        if (s_input_length > s_command_length) {
            s_commpare_length = s_input_length;
        }
        else {
            s_commpare_length = s_command_length;
        }
        // 返回为 0 表示命令匹配成功
        if (raw_strncmp((const char *)pcCommandInput, (const char
*)pcRegisteredCommandString, s_commpare_length) == 0 ) {

            // 当前结点所对应的命令的参数个数>=0
            if
(current_process_command_point->pxCommandLineDefinition->cExpectedNum
berOfParameters >= 0) {
                // 实际输入的命令所带的参数个数错误
                if (rsh_get_parameters_numbers( pcCommandInput ) !=
current_process_command_point->pxCommandLineDefinition->cExpectedNumb
erOfParameters) {

                    xReturn = 1;
                }
            }
            break;          // 终止遍历循环
        }
    }
}
```

```

}
// 输入命令正确，参数错误，则输出提示
if (current_process_command_point && (xReturn == 1)) {

    raw_strncpy( ( char * ) pcWriteBuffer, "\rIncorrect command
parameter(s). Enter \"help\" to view a list of available
commands.\r\n\r\n", xWriteBufferLen );

    current_process_command_point = 0;           // 遍历指针归 0
}
// 输入命令正确
else if (current_process_command_point != 0) {
    // 清空输出缓冲区
    raw_memset(pcWriteBuffer, 0, xWriteBufferLen);
    // 执行命令解析函数
    xReturn =
current_process_command_point->pxCommandLineDefinition->pxCommandInte
rpreter(pcWriteBuffer, xWriteBufferLen, pcCommandInput);
    // 执行正确，归 0 遍历指针
    if (xReturn == 1) {
        current_process_command_point = 0;
    }
}
// 输入命令错误，打印错误信息
else {
    raw_strncpy( ( char * ) pcWriteBuffer, (const char *) "\rCommand
not recognised. Enter \"help\" to view a list of available
commands.\r\n\r\n", xWriteBufferLen );
    xReturn = 1;
}
return xReturn;
}

```

至此，shell 机制中添加命令，执行命令的过程分析完了，下面分析默认的 help 命令的实现过程。

4: rsh.c 中其它相关函数

// 得到命令行的长度，rsh_process_command 函数中调用

```

static RAW_U8 command_length_get(const RAW_S8 *command)
{
    const RAW_S8 *sc;
    for (sc = command; (*sc != '\0') && (*sc != ' '); ++sc)
        return sc - command;
}

```

// 得到参数的个数，rsh_process_command 函数中调用

```
static RAW_S8 rsh_get_parameters_numbers( const RAW_S8 *
pcCommandString )
{
    RAW_S8 cParameters = 0;           // 参数的个数
    RAW_S32 xLastCharacterWasSpace = 1; // 通过空格字符来区分参数
    while( *pcCommandString != 0x00 ) // 遍历字符串
    {
        if( ( *pcCommandString ) == ' ' ) // 当前字符为空格
        {
            if( xLastCharacterWasSpace != 0 )
            {
                cParameters++;           // 参数+1
                xLastCharacterWasSpace = 0;
            }
        }
        Else
        {
            xLastCharacterWasSpace = 1;
        }
        pcCommandString++;
    }

    return cParameters;
}
```

// 得到第 uxWantedParameter 个参数（字符串），并返回参数的地址

```
const RAW_S8 *rsh_get_parameter( const RAW_S8 *pcCommandString, RAW_S32
uxWantedParameter, RAW_S32 *pxParameterStringLength )
{
    RAW_S32 uxParametersFound = 0;           // 当前遍历的参数序号
    const RAW_S8 *pcReturn = 0;

    *pxParameterStringLength = 0;           // 参数长度
    // 遍历到要找的那个参数
    while( uxParametersFound < uxWantedParameter )
    {
        // 跳过一个字符串（包括命令字符串本身和非预期参数字符串）
        while( ( ( *pcCommandString ) != 0x00 ) && ( ( *pcCommandString ) !=
' ' ) ) )
        {
            pcCommandString++;
        }
    }
}
```



```

    }

    // 跳过空格
    while( ( ( *pcCommandString ) != 0x00 ) && ( ( *pcCommandString )
== ' ' ) )
    {
        pcCommandString++;
    }

    // 判断当前字符串是否是要找的参数
    if( *pcCommandString != 0x00 )
    {
        // 参数个数计数
        uxParametersFound++;
        // 是预期的参数
        if( uxParametersFound == uxWantedParameter )
        {
            // 得到该参数（字符串）的地址
            pcReturn = pcCommandString;
            // 统计参数（字符串）长度
            while( ( ( *pcCommandString ) != 0x00 ) &&
( ( *pcCommandString ) != ' ' ) )
            {
                ( *pxParameterStringLength )++;    // 长度+1
                pcCommandString++;
            }

            break;
        }
    }
    else
    {
        break;
    }
}

return pcReturn;    // 返回参数（字符串）地址
}

```

5: rsh.c 中 help 命令执行的实现

```

// 命令结点
static const xCommandLineInputListItem *current_help_command_point;

```

```
static RAW_S32 rsh_help_command( RAW_S8 *pcWriteBuffer, size_t
xWriteBufferLen, const RAW_S8 *pcCommandString )
{
    RAW_S32 xReturn;

    ( void ) pcCommandString;

    if (current_help_command_point == 0) {

        /* Reset the pxCommand pointer back to the start of the list. */
        current_help_command_point = &xRegisteredCommands;
    }

    /* Return the next command help string, before moving the pointer on
to
the next command in the list. */
    raw_strncpy( ( char * ) pcWriteBuffer, ( const char
* )current_help_command_point->pxCommandLineDefinition->pcHelpString,
xWriteBufferLen );
    current_help_command_point = current_help_command_point->pxNext;

    if (current_help_command_point == 0) {

        /* There are no more commands in the list, so there will be no more
strings to return after this one and 1 should be returned. */
        xReturn = 1;
    }
    else {

        xReturn = 0;
    }

    return xReturn;
}
```

附录三、RAW-OS 时钟系统

1. raw_time_tick 分析

RTOS 的时钟节拍是系统运行的时基，对于 RAW-OS 来说，由于其引入了一些的创新机制，在硬件时钟 ISR 设计上并不是简单的调用系统节拍函数，而是要考虑两个方面，RAW-OS 引入了 Task_0，这个最高优先级的任务是为了配合 RAW-OS 最大关中断 0us 特性存在，所以在 STM32 平台编写的时钟节拍服务函数如下：

```
void SysTick_Handler(void)           // M3,M4 的 SsyTick 定时器中断服务程序
{
    raw_enter_interrupt();             // 进入中断前必须调用
    // 如果开了最大关中断 0 特性, TASK_0 也必须开, 所以此时只能用 task_0_tick_post
    // 来处理时钟节拍
    #if (CONFIG_RAW_TASK_0 > 0)
        task_0_tick_post();
    #else
        // 如果没有开最大关中断 0 特性, 则调用 raw_time_tick() 来更新处理时钟节拍
        raw_time_tick();
    #endif

    raw_finish_int();                 // 退出中断必须调用
}
```

TASK_0 机制为 RAW-OS 创新引入的，其实现也比较复杂，这里先对关最大 0 中断特性时的处理方法作一个分析。

raw_time_tick() 这个函数在 raw_system.c 中定义，其源码如下：

```
RAW_VOID raw_time_tick(void)
{
    // 开了 task_0, 则必须用 task_0_tick_post() 来处理时钟节拍
    #if (CONFIG_RAW_TASK_0 > 0)

        if (raw_int_nesting) {

            RAW_ASSERT(0);

        }

    #endif

    #if (CONFIG_RAW_USER_HOOK > 0)           // 自定义 hook 函数
        raw_tick_hook();
    #endif
}
```

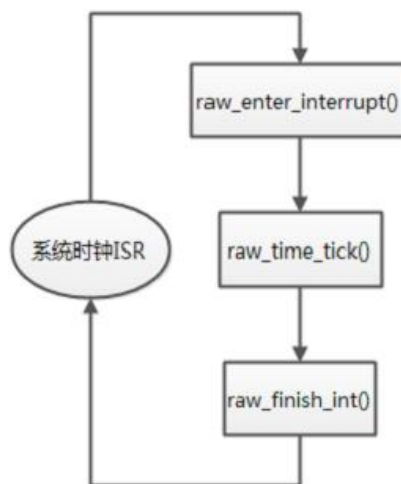
```

/*update system time to calculate whether task timeout happens*/
// 如果开了 tick_task 任务，则发出信号量，同步 tick_task 任务，把 tick 更新放
    到任务级，这里是最大限度的保证实时性（后面会分析）
#ifdef (CONFIG_RAW_TICK_TASK > 0)
    raw_task_semaphore_put(&tick_task_obj);
#else
    // 否则直接在中断中更新 tick_list
    tick_list_update();
#endif

/*update task time slice if possible*/
#ifdef (CONFIG_SCHED_FIFO_RR > 0) // 计算时间片
    calculate_time_slice(raw_task_active->priority);
#endif

/*inform the timer task to update software timer*/
#ifdef (CONFIG_RAW_TIMER > 0) // 开了定时器
    call_timer_task();
#endif
}
    
```

系统时钟节拍的执行流程一般是这样的：



在这里详细讨论这个 `raw_enter_interrupt()` 和 `raw_finish_int()` 函数，前者是入中断 ISR 时首先调用，后者是退出中断 ISR 时最后调用，那么这两个函数做了什么工作，看看源码注释：

```

RAW_U16 raw_enter_interrupt(void)
{
    RAW_SR_ALLOC();
    // 检查中断嵌套层数，超过默认阈值立即返回
    
```

```

if (raw_int_nesting >= INT_NESTED_LEVEL) {
    return RAW_EXCEED_INT_NESTED_LEVEL;
}

// 关 CPU 中断，因为 raw_int_nesting 变量属于临界资源
RAW_CPU_DISABLE();

raw_int_nesting++;          // 中断嵌套层数++，中断也可以被中断
RAW_CPU_ENABLE();

return RAW_SUCCESS;
}

```

Raw_finish_int() 函数源码，退出中断时必须调用，在 raw_system.c 中定义：

```

RAW_VOID raw_finish_int(void)
{
    RAW_SR_ALLOC();          // 定义一个变量来存放 cpu 的状态寄存器

    USER_CPU_INT_DISABLE();  // 保存 cpu 状态寄存器到上面定义的变量中

    // raw_finish_int 必须与 raw_int_enter() 配套使用，raw_int_nesting 为 0
    // 说明已经处理完了中断，不能再调用，所以这里恢复 cpu 状态寄存器，并直接返回
    if (raw_int_nesting == 0) {
        USER_CPU_INT_ENABLE();
        return;
    }

    // 中断嵌入层数-1
    raw_int_nesting--;

    // 如果中断仍在嵌套中，则恢复 cpu 状态寄存器，返回
    if (raw_int_nesting) {
        USER_CPU_INT_ENABLE();
        return;
    }

    // 关调度时直接返回
    if (raw_sched_lock) {

        USER_CPU_INT_ENABLE();
        return;
    }

    // 运行到这里说明是最后一层中断了，所以要在退出时执行任务调度
    // 查找最高优先级任务
    get_ready_task(&raw_ready_queue);

    // 如果被中断的任务仍为最高优先级，则无需调度直接返回

```

```

if (high_ready_obj == raw_task_active) {
    USER_CPU_INT_ENABLE();
    return;
}

// 系统调试相关
TRACE_INT_TASK_SWITCH(raw_task_active, high_ready_obj);

// 如果有更调优先级任务就绪，则进行中断级任务调度，切换到最高优先级任务
raw_int_switch();

USER_CPU_INT_ENABLE();
}

```

对于上面函数中出现的宏，他们的定义如下，很好理解，不作解释。

```

#define RAW_SR_ALLOC() unsigned int cpu_sr = 0

#define USER_CPU_INT_DISABLE() {cpu_sr = OS_CPU_SR_Save();}

#define USER_CPU_INT_ENABLE() {OS_CPU_SR_Restore(cpu_sr);}

```

对于 raw_time_tick 函数中的关键部分：

```

#if (CONFIG_RAW_TICK_TASK > 0) // 将 tick_list 更新转到任务级
raw_task_semaphore_put(&tick_task_obj);
#else
tick_list_update(); // 直接在中断中更新
#endif

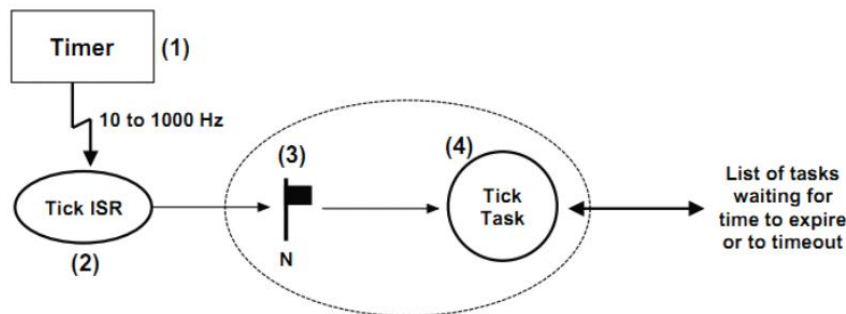
```

总的来说是这样的：如果开了最大关中断 0 特性，则必须开 task_0 机制，则时钟节拍的处理更新由 task_0 来处理（以后会分析原理），如果没有开最大关中断 0 特性，则还可以开 tick_task 任务，将时钟节拍的处理更新转到 tick_task 这个任务来处理，如果两个都没开，则在 raw_time_tick 中直接 update tick list。

总之就是，能不在中断中做的就转到任务级处理，最大保证系统的实时性，txj 的这句话让我记忆深刻，Raw-os 最大关中断 0us 特性确实牛 X，task_0 特性以后再慢慢研究。

2. tick_task（时钟节拍任务）分析

又引入了 tick_task，这是一个任务，专门用于更新系统时钟节拍的任务，为什么要引入这个东西？还是那句话，能不在中断中做的都不在中断中做，为的是最大保证系统实时性，当然，ucos ii 是没有这些东西的，他是直接更新 tick list，这个机制的原理如下图所示：



如果在 raw_config.h 中配置了如下宏：

```
#define CONFIG_RAW_TICK_TASK
```

那么以下的配置就会生效，即 tick_task 的任务栈空间和优先级，优先级为 1。

```
#if (CONFIG_RAW_TICK_TASK > 0)
#define TICK_TASK_STACK_SIZE      256
#define TICK_TASK_PRIORITY        1
#endif
```

那么在 raw_os_init()函数，即 Raw-os 的初始化函数中会有下面的代码：

```
#if (CONFIG_RAW_TICK_TASK > 0)
tick_task_start();
#endif
```

这个 tick_task_start()函数就是创建 tick_task 任务的，关于 tick_task，相关源码如下：

```
#if (CONFIG_RAW_TICK_TASK > 0)                                // 使能了 tick_task

// tick_task 处理函数
static void tick_task_process(void *para)
{
    RAW_U16 ret;
    while (1) {
        // 等待信号量。。。还记得 raw_time_tick 函数中的：
        // raw_task_semaphore_put(&tick_task_obj) 吗
        // 第时在时钟节拍 ISR 中，发送信号量来同步这个 tick_task
        ret = raw_task_semaphore_get(RAW_WAIT_FOREVER);

        // 获得信号量成功且系统正在运行的性况下，更新 tick_list_update
    }
}
```

```

        if (ret == RAW_SUCCESS) {
            if (raw_os_active == RAW_OS_RUNNING) {
                tick_list_update();
            }
        }
    }
}

// tick_task_start 函数用于 tick_task 的初始化, 在 raw_os_init 中调用
void tick_task_start(void)
{
    // 创建 tick_task 任务, 用于 update tick list
    raw_task_create(&tick_task_obj, (RAW_U8 *) "tick_task_object", 0,
        TICK_TASK_PRIORITY, 0, tick_task_stack, TICK_TASK_STACK_SIZE,
        tick_task_process, 1);

    // 创建一个用于 tick 同步的信号量
    raw_task_semaphore_create(&tick_task_obj, &tick_semaphore_obj,
        (RAW_U8 *) "tick_semaphore_obj", 0);
}

#endif

```

其中对于 tick_task 相关的定义（栈，任务对象，同步信号量）是在 raw_obj.c 中定义：

```

#ifdef CONFIG_RAW_TICK_TASK > 0

RAW_TASK_OBJ            tick_task_obj;
PORT_STACK              tick_task_stack[TICK_TASK_STACK_SIZE];
RAW_SEMAPHORE           tick_semaphore_obj;

#endif

```

额，好了，总算是弄完了，从这函数分析还可以知道 raw_task_semaphore 是这样用的，raw_task_semaphore_put(&tick_task_obj)，直接把信号量发给了 tick_task 这个任务，在 RAW-OS 中提供了任务信号量这种机制，当时确的知道 ISR 和任务或任务和任务之间的同步关系时，可以直接把信号量发给这个任务。

3. tick_list_update

好像还漏了点什么，对，tick_list_update()，在 raw_time_tick() 中，关键就是对这个玩意的调用，看上面 tick_task_process() 的源码知道，tick_task 的存在就是为了调用这个东西，再来分析一下它，不分析源码，只分析原理。

系统内核维护了一个任务就绪队列，所有处于就绪态的任务都挂在这个就绪队列上，当进行任务调度的时候，会从这个队列中取出最高优先级的任务来运行，

任务调度必然伴随着任务状态的切换（看上一篇学习笔记——任务篇），那么就涉及到任务对象在就绪队列的插入，删除等，`tick_list_update()`正是维护这个就绪队列，同时也更新系统，任务的一些状态标志，此过程比较复杂，自行看源码结合 `txj` 的视频讲解看，理解的更好。

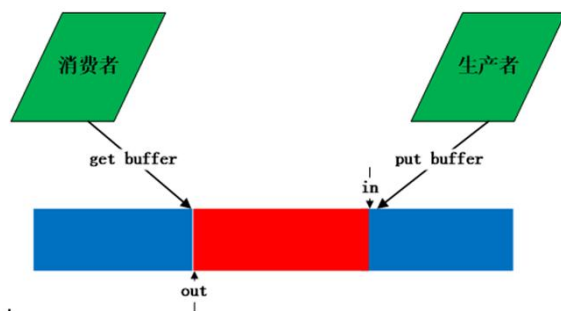
另外，除了就绪队列外，`tick_list_update()`还更新了 `tick_list` 这个链表，这个链表上挂接的就是那需要等待系统时基来获得运行权的任务，具体还没看到这些个功能，后面再看，再分析。

总之来说，`tick_list_update()`就是遍历系统中中所有等待期满的任务、等待事件超时的任务，就绪的任务，然后发生一系列的转换，使系统有序的运转。

附录四、RAW-OS FIFO 机制分析

RAW-OS 内核扩展出通用的 FIFO 功能, 这给一些数据处理应用提供了极大的方便。这种 FIFO 写法与 Linux 内核中的 FIFO 有异曲同工之妙, 是一种典型的生产者, 消费者问题解决方案, 下面分析之。

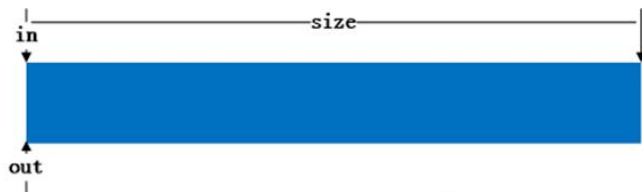
FIFO 模型如下:



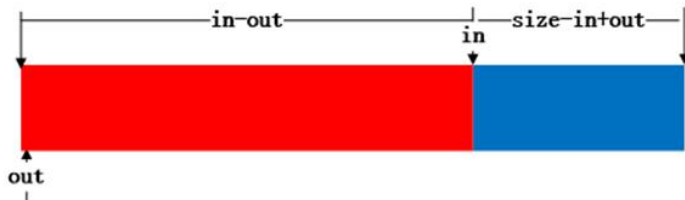
1. FIFO 数据结构定义

```
struct raw_fifo {
    RAW_U32    in;           // 写指针
    RAW_U32    out;          // 读指针
    RAW_U32    mask;
    void       *data;        // 数据指针
    RAW_U32    free_bytes;    // 空余大小(字节)
    RAW_U32    size;         // FIFO 大小
};
```

1.1 空的 FIFO



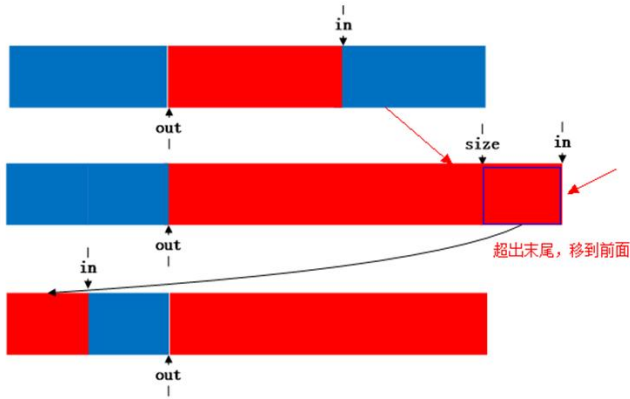
1.2 put 一个 buffer 后的 FIFO



1.3 get 一个 buffer 后的 FIFO



1.4 写入一个 buffer 的长度大于 $size - in$ 的长度了，则返回到 FIFO 头部接着写



入队使 in 递增，出队使 out 递增， in 和 Out 永远递增!!! 但是却不会溢出，这正是该 $fifo$ 实现的精妙之处。

根据上面的图可知求得 FIFO 中剩余的可用空间代码如下：

```
static RAW_U32 fifo_unused(struct raw_fifo *fifo)
{
    return (fifo->mask + 1) - (fifo->in - fifo->out);
}
```

2. FIFO 结构操作方法

```
// 初始化 FIFO
RAW_S8 fifo_init(struct raw_fifo *fifo, void *buffer, RAW_U32 size);
// 写 FIFO
RAW_U32 fifo_in(struct raw_fifo *fifo, const void *buf, unsigned int len);
// 读 FIFO (len 长度数据)
RAW_U32 fifo_out(struct raw_fifo *fifo, void *buf, RAW_U32 len);
RAW_U32 fifo_out_peek(struct raw_fifo *fifo, void *buf, RAW_U32 len);
// 读 FIFO (所有 FIFO 中的数据)
RAW_U32 fifo_out_all(struct raw_fifo *fifo, void *buf);
```

3. FIFO 操作实现

FIFO 本质是一个环形循环队列，在使用前必须定义一个 FIFO 类型的变量和一个缓冲区：

```
static struct raw_fifo uart0_receive_fifo; // 定义一个变量(uart 接收数据缓冲)
static char fifo_receive_buf[1024];       // 数据存储缓冲区
```

3.1 FIFO 初始化

```
/*
函数名称:   fifo_init
参数说明:   fifo: 要初始化的 fifo 变量; buffer: fifo 存储内存地址; size: fifo
大小
功能描述:   对指定大小 fifo 结构成员进行初始化
返回值:     0: 初始化成功  1: 失败
*/
RAW_S8 fifo_init(struct raw_fifo *fifo, void *buffer, RAW_U32 size)
{
    if (!is_power_of_2(size)) { // FIFO 大小必须量 2 的幂次方
        RAW_ASSERT(0);
    }
    fifo->in = 0;                // 读写指针归 0
    fifo->out = 0;
    fifo->data = buffer;         // 数据存储缓冲区

    if (size < 2) {              // size 为 0 则返回
        fifo->mask = 0;
        return 1;
    }
    fifo->mask = size - 1;       // 读写指针位置掩码
    fifo->free_bytes = size;     // FIFO 可用空间

    return 0;                   // 初始化成功返回 0
}
```

3.2 FIFO 写（入队）

```
/*
函数名称:   fifo_in
参数说明:   fifo: 目标 FIFO; buf: 要写入的数据; len: 要写入的数据长度
功能描述:   把 buf 中 len 长度数据写入 FIFO
返回值:     成功写入的数据长度
*/
RAW_U32 fifo_in(struct raw_fifo *fifo, const void *buf, RAW_U32 len)
{
    RAW_U32 l;
    RAW_SR_ALLOC();
    RAW_CPU_DISABLE();

    l = fifo_unused(fifo);      // 计算 FIFO 剩余空间
    if (len > l)                // 要写入的数据量大于剩余空间
```

```

        len = 1;                                // 则写满为止

// 把数据写入 FIFO, 写入起始位置为 fifo->in
fifo_copy_in(fifo, buf, len, fifo->in);
fifo->in += len;                                // 写指针后移 len (写入的长度)

fifo->free_bytes -= len;                        // 剩余空间-len

RAW_CPU_ENABLE();
return len;                                    // 返回写入的长度
}
/*
函数名称: fifo_copy_in
参数说明: fifo: 目标 FIFO; buf: 要写入的数据; len: 要写入的数据长度; off: 写入起始地址
功能描述: 把 src 中的数据写入到 fifo 中 off 位置处 (内部调用)
返回值: 无
*/
static void fifo_copy_in(struct raw_fifo *fifo, const void *src, RAW_U32
len, RAW_U32 off)
{
    RAW_U32 l;

    RAW_U32 size = fifo->mask + 1;

    off &= fifo->mask;                          // 写入的位置 (精妙所在)

    // 如果 len 小于 size - fifo->in, 则直接写, l = len, 只 copy l 长度
    // 如果 len 大于 size - fifo->in, 则要回写到队头
    l = fifo_min(len, size - off);
    // copy 到 fifo 队尾 (长度为 l)
    raw_memcpy((unsigned char *)fifo->data + off, src, l);
    // 若 len 大于 size - fifo->in, 则下面是回写到队头, 长度为 len - (size -
    fifo->in)
    raw_memcpy(fifo->data, (unsigned char *)src + l, len - l);
}

```

上面代码的精妙之处在于, `fifo->in` 为写指针, 它的本质是一个 `unsigned int` 型变量 `off &= fifo->mask` 即是 `fifo->in & (fifo->size - 1)`, 在初始化代码保证了 `fifo->size` 为 2 的幂次方, 比如上面定义的 1024, 二进制表示 100 0000 0000, `fifo->size - 1` 即为 011 1111 1111, OK, 有点明白了, 这句代码就是 `fifo->in` 对 `fifo->size` 取余, 这样 `fifo->in` 写到尾部后会从头开始写。那 `fifo->in` 本质上为 `unsigned int`, 溢出了呢? 后面再分析。

3.4 FIFO 读 (出队)

```

/*
函数名称: fifo_out
参数说明: fifo:目标 FIFO; buf:读出数据的存储地址; len:读出数据长度
功能描述: 从 fifo 中读出 len 长度的数据到 buf 中
返回值:   读出数据的长度
*/
RAW_U32 fifo_out(struct raw_fifo *fifo, void *buf, RAW_U32 len)
{
    RAW_SR_ALLOC();

    RAW_CPU_DISABLE();
    // 读出数据到 buf 中, len 为实际读取的长度
    len = internal_fifo_out_peek(fifo, buf, len);
    fifo->out += len;           // 移动读指针

    fifo->free_bytes += len;    // fifo 可用空间增加 len

    RAW_CPU_ENABLE();

    return len;                // 返回读取的数据长度
}

/*
函数名称: internal_fifo_out_peek
参数说明: fifo:目标 FIFO; buf:读出数据的存储地址; len:读出数据长度
功能描述: 从 fifo 中 off 位置读出 len 长度的数据到 buf 中
返回值:   读出数据的长度
*/
static RAW_U32 internal_fifo_out_peek(struct raw_fifo *fifo, void *buf,
RAW_U32 len)
{
    RAW_U32 l;

    l = fifo->in - fifo->out;    // 实际存储的数据长度
    if (len > l)                // 要读的数据超过实际长度
        len = l;               // 只读取有效长度

    kfifo_copy_out(fifo, buf, len, fifo->out);
    return len;                // 返回读取的长
}

/*
函数名称: kfifo_copy_out
参数说明: fifo:目标 FIFO; dst:读出数据的存储地址; len:读出数据长度; off: 读取位置
功能描述: 从 fifo 中 off 位置读出 len 长度的数据到 buf 中

```

返回值： 读出数据的长度

```

*/
static void kfifo_copy_out(struct raw_fifo *fifo, void *dst, RAW_U32 len,
RAW_U32 off)
{
    RAW_U32 l;
    RAW_U32 size = fifo->mask + 1;

    off &= fifo->mask;          // 读指针定位

    // 如果 len 小于 size - fifo->out, 则直接读, l = len, 只 copy len 长度
    // 如果 len 大于 size - fifo->out, 则要回读到队头
    l = fifo_min(len, size - off);
    // copy 到 fifo 队尾 (长度为 l = len)
    raw_memcpy(dst, (unsigned char *)fifo->data + off, l);
    // 若 len 大于 size - fifo->out, 则下面是回读到队头, 长度为 len - (size -
    fifo->out)
    raw_memcpy((unsigned char *)dst + l, fifo->data, len - l);
}

```

3.5 读出所有 fifo 中的数据

```

/*
函数名称: fifo_out_all
参数说明: fifo: 目标 FIFO; buf: 读出数据的存储地址
功能描述: 从 fifo 中读出所有的数据
返回值:  读出数据的长度
*/
RAW_U32 fifo_out_all(struct raw_fifo *fifo, void *buf)
{
    RAW_U32 len;

    RAW_SR_ALLOC();
    RAW_CPU_DISABLE();

    len = fifo->size - fifo->free_bytes; // 已用空间

    if (len == 0) {

        RAW_CPU_ENABLE();
        return 0;
    }

    // 读出数据到 buf 中
    len = internal_fifo_out_peek(fifo, buf, len);
}

```

```
fifo->out += len;           // 更新读指针
fifo->free_bytes += len;    // 更新可用空间
RAW_CPU_ENABLE();
return len;
}
```

接下来分析溢出问题，`fifo->in` 和 `fifo->out` 一直在加，超过其数据类型限制怎么样呢？分情况分析：

1. `in` 和 `out` 都没有溢出或都溢出了，则 `size - (fifo->in - fifo->out)` 及 `in` 和 `out` 。
2. 不可能 `in` 没溢出, `out` 溢出了，从上面代码知道，读写数据的长度都是进行了有效检查的。
3. `out` 没溢出，`in` 溢出了。

附录五、RAW-OS 中断下半部和 work_queue 机制分析

1. 中断下半部原理

中断一般可以分成上半部以及下半部, 所谓中断上半部 `cpu` 通常是关中断的, 这个时候主要是从硬件那里接收一些数据, 然后触发中断下半部。一出中断后, 根据相应的中断下半部的优先级去执行相应的中断下半部。中断下半部主要用于数据的处理, 数据的处理过程一般比较缓慢, 所以这个时候中断往往是打开的。为什么要有中断上下半部的原因主要有如下。

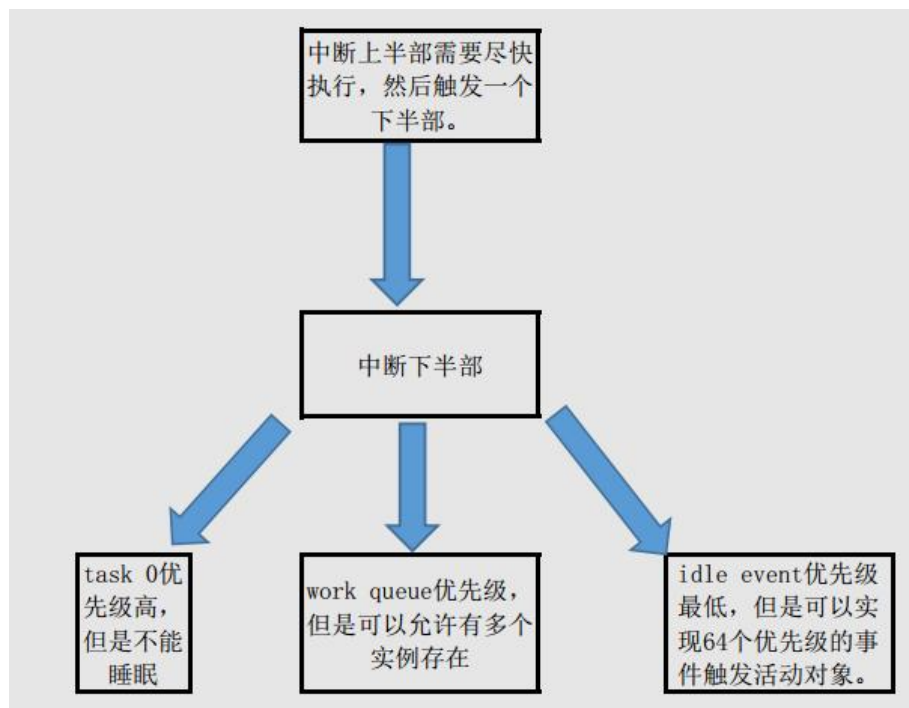
1. 如果中断函数内处理的事情非常多的话, 意味着中断运行时间过长。这个时候 `cpu` 通常是关中断的, 这样会无法及时响应其它的中断, 造成其他中断数据丢失。

2. 对于 `cortex-m` 系列的 `cpu` 进入中断后是开着中断的, 但是中断运行时间太长的话, 其它低优先级的中断也得不到运行, 因为被高优先级的任务占着中断不放, 通常也会引发低优先级中断的数据丢失。所以尽可能减低中断内处理的时间是刻不容缓, 非常重要的事情, 直接决定了整个系统的实时性。

中断下半部是人为的划分, 这个时候是处于任务环境下, 所以可以响应中断。中断下半部的划分主要是首先判断某具体中断内的执行时间, 如果时间很长的话, 就需要把数据以及数据处理部分人为的划分开来, 接收数据部分往往叫中断上半部, 处理数据部分往往叫中断下半部。

中断的下半部处理是可以是有优先级的, 有的硬件设备的中断数据处理可以延迟处理, 但是有的必须马上要处理, 事情处理有轻重缓急之分, 这也是为什么中断下半部有 `task 0` 和 `workqueue`, `idle event` 的原因。

下图是 raw os 中可能使用到的中断流程:



Raw-os 支持的中断下半部方法有 task 0 以及 work queue, task 0 的优先级要高于 work queue。work queue 的优先级是高于 idle event 的。task 0 和 work queue 的区别是 task 0 运行的时候不能调用能引起睡眠的函数, 但是 work queue 是能调用引起睡眠的函数的, idle event 是基于事件驱动的, 优先级是最低的, 也可以做为中断下半部去使用, 如果这个中断的处理优先级是很低的话, idle event 也是不能允许睡眠的。

raw os 的中断下半部最具特色的是支持上半部向下半部传递数据, 这一点非常重要, 把数据接收和数据处理彻底的分割开来。有些系统比如 linux, 上半部是无法向下半部传递数据的, 因为内核的 api 不支持这个特性。

综上所述, raw os 为软件化的区分中断优先级提供了有效地手段, 即 task 0, workqueue 以及 idle event。对于系统的快速中断响应有了根本的保障。

2. work queue 机制分析

上面已经讲述了中断下半部的原理, 下面重点分析一下 raw os 中的 work queue 机制实现, work queue 是有优先级的, 因为它的本质是把数据处理转移到任务级。

举个例子, GPS 通过串口以一定的频率返回数据, 在串口中断中接收, 当接收完一个完整的数据包时, 可以同步一个任务, 然后把数据包发给任务, 任务再进行解析处理得到经纬度, 时间, 航向, 速度等数据, 然后再进行下一部处理, 这种处理方法也是一种中断下半部的处理思路。

再比如, 网卡驱动程序以中断的方式接收数据包, 这个数据包很大, 协议也比较复杂, 用中断下半部的思路, 接收完数据包后, 把数据传送给任务, 让任务去进行协议的解析, 这样中断的时间就会比较短, 从而保证了系统的实时性。

好了, 总结一下就是, 中断尽可能的短, 把数据处理放到任务级去做。那 work queue 的工作原理也即是把相关数据结构进行一些封装, 提供统一的系统 API, 以一种通用的模型来解决此类问题。下面分析源码。

任务作为 raw os 内核的基本运行单位, 它的内核对像是 RAW_TASK_OBJ, Work queue 作为 raw os 内核一个独立的模块, 其运行和处理的基本单位是 WORK_QUEUE_STRUCT, 该结构体定义如下:

```
typedef struct WORK_QUEUE_STRUCT {  
  
    RAW_QUEUE queue;  
    RAW_TASK_OBJ work_queue_task_obj;  
  
} WORK_QUEUE_STRUCT;
```

从上面可以看出, WORK_QUEUE_STRUCT 的本质就是一个队列和一个任务的封装, 结合上面的分析不难理解, 中断通过 queue 给 work queue 传递数据, 然后任务就是处理数据。

Work queue 对数据的传送和处理不仅仅是一个队列和任务这么简单，而是把数据的属性进行抽象，得到了如下数据模型，它的源码如下：

```
typedef struct OBJECT_WORK_QUEUE_MSG {

    struct OBJECT_WORK_QUEUE_MSG *next;    // 指向下一条数据消息的指针
    WORK_QUEUE_HANDLER            handler;  // 数据处理回调函数
    void *msg;                     // 指向数据消息的指针
    RAW_U32                       arg;      // 存储数据的 32 位变量

} OBJECT_WORK_QUEUE_MSG;
```

上面结构体的成员意义也很好理解，对于后面三项还是有必要说一下。

Work queue 的本质是在任务中对数据进行处理，且是一种抽象的代码封装，一种通用的模型，它不问你要处理什么数据，怎么处理，只是提供给你数据传送的机制和处理的机制。所以 msg 和 arg 即是中断向 work queue 传递的待处理的数据消息的地址和 32 位整形变量。而 handler 则是用户自定义的数据处理回调函数，它会在 work queue 的任务执行中调用。

回调函数的类型定义如下：

```
typedef RAW_VOID (*WORK_QUEUE_HANDLER)(RAW_U32 arg, void *msg);
```

它可以接收一个 32 位和一个指针参数。

在使用 work queue 前需要调用以下函数初始化指向数据消息的指针。这个函数的作用是对用户定义的 work_queue_msg 指针数组进行初始化为一个单链表。

```
void global_work_queue_init(OBJECT_WORK_QUEUE_MSG *work_queue_msg,
RAW_U32 size)
{
    OBJECT_WORK_QUEUE_MSG *p_msg1;
    OBJECT_WORK_QUEUE_MSG *p_msg2;
    // 指向存储消息的指针数组
    free_work_queue_msg = work_queue_msg;
    /*init the free msg list*/
    p_msg1 = work_queue_msg;
    p_msg2 = work_queue_msg;
    p_msg2++;
    while (--size) { // 把指针数组处理成单链表
        p_msg1->next = p_msg2;
        p_msg1++;
        p_msg2++;
    }
    /*init the last free msg*/
    p_msg1->next = 0;
}
```

然后创建数据队列的代码如下：

```
RAW_U16 work_queue_create(WORK_QUEUE_STRUCT *wq,      // work queue 对像
                          RAW_U8 work_task_priority,  // 任务优先级
                          RAW_U32 work_queue_stack_size, // 任务栈大小
                          PORT_STACK *work_queue_stack_base, // 任务栈
                          RAW_VOID **msg_start,      // 存储消息用的指针数组
                          RAW_U32 work_msg_size)     // 指针数组的大小
{
    RAW_U16 ret;
    // 创建一个队列,work queue 中的任务阻塞在队列上,有消息时则唤醒处理
    ret = raw_queue_create(&wq->queue, "work_queue", msg_start,
work_msg_size);
    // 创建失败返回
    if (ret != RAW_SUCCESS) {
        return ret;
    }
    // 创建一个任务,平时阻塞在队列上,有数据需要处理时唤醒执行
    ret = raw_task_create(&wq->work_queue_task_obj,    // 任务对像
                        (RAW_U8 *) "work_queue",      // 任务名子
                        wq,                            // 传给任务的参数
                        work_task_priority, 0,         // 任务的优先级
                        work_queue_stack_base,         // 任务栈首地址
                        work_queue_stack_size,         // 任务栈大小
                        work_queue_task,               // 任务执行函数
                        1);
    // 任务创建失败返回
    if (ret != RAW_SUCCESS) {
        return ret;
    }
    return RAW_SUCCESS;
}
```

对于上面的队列创建，没什么好说的，注意到任务创建时是传入了参数的，把工作队列的地址传给了任务，那么在任务的执行函数中可以接收这个参数然后处理。

任务的执行函数如下：

```
static void work_queue_task(void *pa)
{
    RAW_U16 ret;
    OBJECT_WORK_QUEUE_MSG *msg_rcv;           // 接收消息
    WORK_QUEUE_STRUCT *wq;
```

```

RAW_SR_ALLOC();

wq = pa;                                // 接收传入参数

while (1) {
    // 任务阻塞在队列上,直到接收到消息时唤醒
    ret = raw_queue_receive (&wq->queue, RAW_WAIT_FOREVER, (void
**) (&msg_rcv));

    if (ret != RAW_SUCCESS) {
        RAW_ASSERT(0);
    }
    // 接收到了消息,执行用户回调函数,并传入参数
    msg_rcv->handler(msg_rcv->arg, msg_rcv->msg);

    RAW_CPU_DISABLE();

    msg_rcv->next = free_work_queue_msg;    // 下一条存储空间
    free_work_queue_msg = msg_rcv;

    RAW_CPU_ENABLE();
}
}

```

最后,在中断中调用下面的函数,把接收到数据地址传递给 work queue, work queue 被唤醒然后执行。

```

OBJECT_WORK_QUEUE_MSG *free_work_queue_msg;

RAW_U16 sche_work_queue(WORK_QUEUE_STRUCT *wq, RAW_U32 arg, void *msg,
WORK_QUEUE_HANDLER handler)
{
    void *msg_data;
    RAW_U16 ret;

    RAW_SR_ALLOC();

    RAW_CPU_DISABLE();
    // 没有消息存储空间
    if (free_work_queue_msg == 0) {
        RAW_CPU_ENABLE();
        return RAW_WORK_QUEUE_MSG_MAX;
    }
}

```

```
msg_data = free_work_queue_msg;

free_work_queue_msg->arg = arg;           // 传参, 32 位整数
free_work_queue_msg->msg = msg;           // 传参, 指向数据的地址
free_work_queue_msg->handler = handler;   // 传参, 数据处理回调函数

free_work_queue_msg = free_work_queue_msg->next; // 指向下一条存储空间

RAW_CPU_ENABLE();

ret = raw_queue_end_post(&wq->queue, msg_data); // 发送消息,同步任务

return ret;
}
```

Work queue 机制就涉及到两个结构体, 三个函数, 上面写的很清楚了, 其工作原理也不难理解, 结合具体的例程, 请读者体会。