

摘 要

本课题主要完成了四轴飞行器上操作系统的移植编写，实现了各功能在操作系统上的任务及调度切换。

本设计在上学期项目的基础上，选取 $\mu\text{C}/\text{OS-II}$ 操作系统作为移植的对象，并在将 $\mu\text{C}/\text{OS-II}$ 移植到 STM32F4 平台后，逐步实现各外设功能模块的任务创建和调度切换，实现在四轴飞行器上搭载操作系统进行任务级调度的目标。

关键词：四轴飞行器， $\mu\text{C}/\text{OS-II}$ ，任务调度

目 录

第一章 针对复杂工程问题的方案设计与实现	1
1.1 针对复杂工程问题的方案设计	1
1.2 针对复杂工程问题的推理分析	1
1.3 针对复杂工程问题的方案实现	2
1.3.1 在 keil 中重构 $\mu C/OS-II$ 新工程	2
1.3.2 实现 STM32 中 $\mu C/OS-II$ 任务堆栈初始化和任务上下文切换	10
1.3.3 在 STM32 上完成 $\mu C/OS-II$ 移植	16
1.3.4 在 STM32 上实现任务级调试	20
1.3.5 用 Makefile, HAL 库重构工程	21
1.3.6 在 $\mu C/OS-II$ 上移植外设代码	23
1.3.7 系统优化	24
第二章 系统测试	29
2.1 测试环境	29
2.2 $\mu C/OS-II$ 移植测试	29
2.2.1 任务调度测试	29
2.2.2 含浮点数任务测试	30
2.2.3 外设功能测试	31
2.3 SYSTEMVIEW 移植测试	32
2.4 差分时间链测试	33
2.4.1 正确性测试	33
2.4.2 性能测试	36
第三章 知识技能学习情况	37
第四章 分工协作与交流情况	40
4.1 分工情况	40
4.2 交流情况	40
参考文献	42
致谢	43

说明:

- 1、报告要求 8000 字以上。**
- 2、本模板仅为基本参考，请各位同学根据个人情况进行目录结构扩展。**
- 3、报告正文必须双面打印。**

第一章 针对复杂工程问题的方案设计与实现

本章主要包括课程目标的分析与问题划分，以及各模块化方案的实现。

1.1 针对复杂工程问题的方案设计

本学期，我们要实现将 $\mu\text{C}/\text{OS-II}$ 操作系统移植到四轴飞行器上，并保证各任务在操作系统下正确、有序的调度切换。为实现以上目标，我们可以分为以下几点：

- ①用 KEIL 重构 $\mu\text{C}/\text{OS-II}$ 新工程
- ②ARM STM32 事件驱动机制
- ③ARM STM32 时间驱动机制
- ④启动 ARM STM32
- ⑤任务上下文切换
- ⑥ $\mu\text{C}/\text{OS-II}$ 在 STM32 上的移植
- ⑦SystemView 与任务级调试
- ⑧系统工程重构与优化
- ⑨内核资源互斥访问机制
- ⑩用 Makefile 构建自己的工程

硬件设计方面，我们仍沿用上学期的方案；软件设计方面，我们从了解学习 $\mu\text{C}/\text{OS-II}$ 入手，逐步实现将 $\mu\text{C}/\text{OS-II}$ 移植到四轴飞行器，编写其上下文切换函数等，最终实现任务调度切换，并在系统优化后，提高运行的性能。

1.2 针对复杂工程问题的推理分析

经过对设计目标的精确分析，我们小组将目标任务分为 7 个模块：

- ①在 keil 中重构 $\mu\text{C}/\text{OS-II}$ 新工程
- ②实现 STM32 中 $\mu\text{C}/\text{OS-II}$ 任务堆栈初始化和任务上下文切换
- ③在 STM32 上完成 $\mu\text{C}/\text{OS-II}$ 移植
- ④在 STM32 上实现任务级调试
- ⑤用 Makefile, HAL 库重构工程

⑥在 $\mu\text{C}/\text{OS-II}$ 上移植外设代码

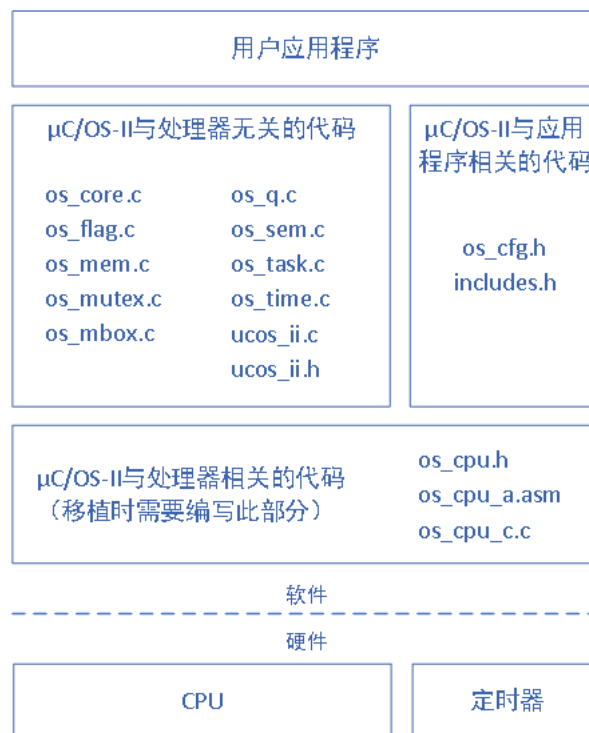
⑦系统优化

在①模块，我们实现将 $\mu\text{C}/\text{OS-II}$ 初步移植到 STM32 上，并在简单的修正后实现成功编译；在②模块，我们实现堆栈初始化和上下文切换函数的编写，实现任务的调度切换；在③模块，我们实现操作系统运行和任务创建所需其他函数的编写，完成 $\mu\text{C}/\text{OS-II}$ 的移植；在④模块，我们使用 SystemView 对任务运行进行直观的展示，方便调试；在⑤模块，我们使用 Makefile 重构在 keil 中实现的工程；在⑥模块，我们将所有外设代码移植到 $\mu\text{C}/\text{OS-II}$ 中，实现任务的调度切换；在⑦模块，我们通过差分时间链的编写，实现了调度时间的缩短，提高了性能。

1.3 针对复杂工程问题的方案实现

在本节中，我们将详细描述各模块是如何具体实现的。

1.3.1 在 keil 中重构 $\mu\text{C}/\text{OS-II}$ 新工程



图表 1-1 $\mu\text{C}/\text{OS-II}$ 体系结构

在本步骤中，我们将在 keil 中添加 $\mu\text{C}/\text{OS-II}$ 源代码，构建 `os_cpu.h` 等与处理

相关代码文件，并顺利通过编译。

① 在 keil 中新建一个工程并拷入 μ C/OS-II 源代码

首先，需要新建一个文件夹，并在文件夹下再新增三个文件夹。

setup	2022/5/22 21:09	文件夹
ucos	2022/5/22 21:09	文件夹
user	2022/5/22 21:09	文件夹

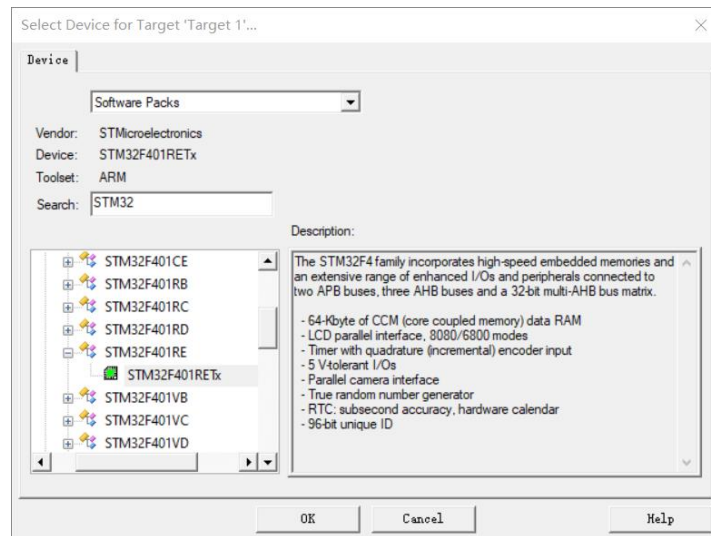
图表 1-2 工程所需文件夹

如图：在 setup 文件夹中放置启动文件 startup_stm32f401xx.s，在 uclos 文件夹中放置 uclos ii 的代码文件，在 user 文件夹中放置自己编写的主文件。

名称	修改日期	类型	大小
os.h	2021/5/20 1:00	C Header File	2 KB
os_core.c	2021/5/20 1:00	C Source File	87 KB
os_dbg_r.c	2021/5/20 1:00	C Source File	13 KB
os_flag.c	2021/5/20 1:00	C Source File	56 KB
os_mbox.c	2021/5/20 1:00	C Source File	32 KB
os_mem.c	2021/5/20 1:00	C Source File	20 KB
os_mutex.c	2021/5/20 1:00	C Source File	40 KB
os_q.c	2021/5/20 1:00	C Source File	43 KB
os_sem.c	2021/5/20 1:00	C Source File	30 KB
os_task.c	2021/5/20 1:00	C Source File	60 KB
os_time.c	2021/5/20 1:00	C Source File	11 KB
os_tmr.c	2021/5/20 1:00	C Source File	45 KB
os_trace.h	2021/5/20 1:00	C Header File	11 KB
uclos_ii.c	2021/5/20 1:00	C Source File	2 KB
uclos_ii.h	2021/5/20 1:00	C Header File	79 KB

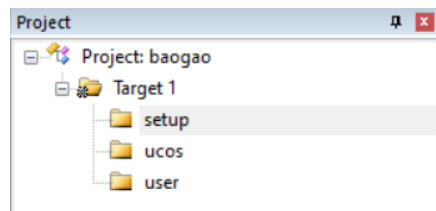
图表 1-3 uclos 文件中需拷贝的 μ C/OS-II 源代码文件

然后，打开 keil 5，把它们导入到项目中。选择 project->new uversion project, 然后选好芯片型号：STM32F401RETx。



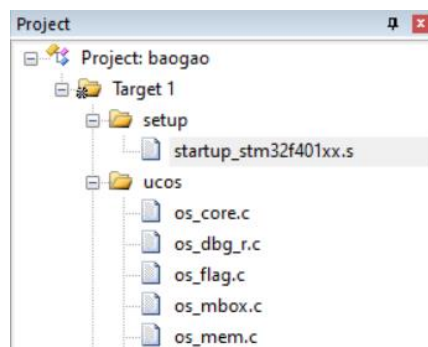
图表 1-4 新建工程中芯片选择页面

右键单击 target 1-add group 新建文件夹，并重命名为 setup, ucoss 和 user。



图表 1-5 为 keil 工程添加文件夹

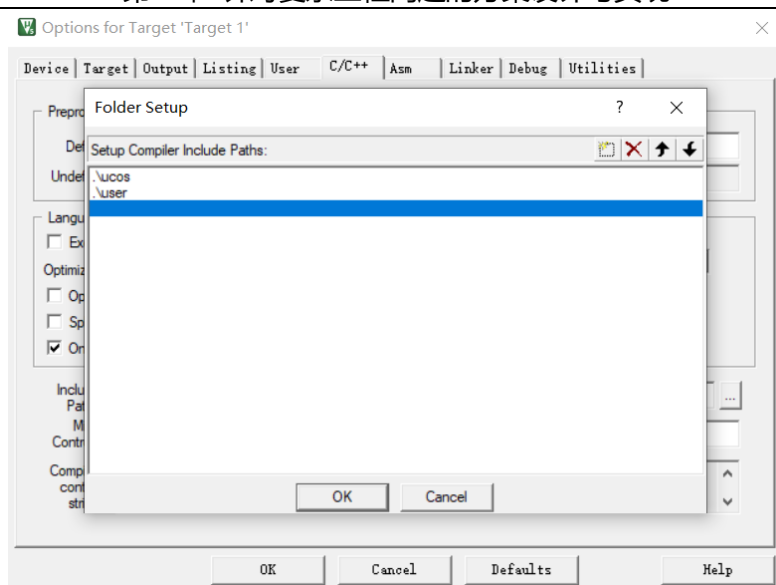
右键单击文件夹，add existing files to group 'xxx'添加文件。



图表 1-6 为 keil 中文件夹添加需要编译的文件

添加头文件路径。打开魔术棒，在 C/C++和 Asm 里的 Include Path 中写入 C 程序和汇编文件的路径。

第一章 针对复杂工程问题的方案设计与实现



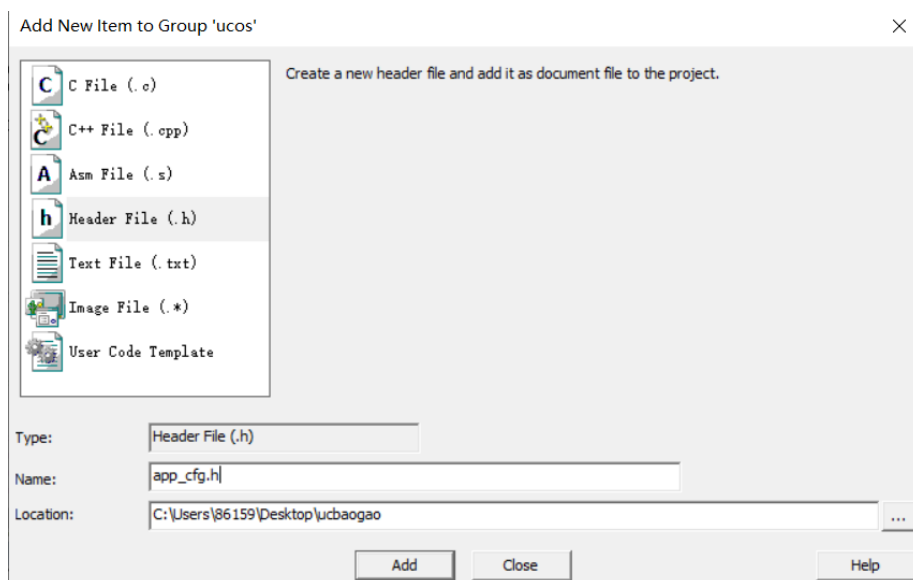
图表 1-7 添加头文件所在路径

① 编译文件并修改相关错误

错误类型 1：缺少文件或者找不到文件。

.ucos\ucos_ii.h(45): error: #5: cannot open source input file "xxx": No such file or directory

解决方案：手动在 ucos 中添加相应文件。



图表 1-8 新建与应用程序、处理器相关代码文件

错误类型 2：INTxxU 型变量未定义

.\ucos\ucos_ii.h(1207): error: #20: identifier "INTxxU" is undefined

解决方案：在 os_cpu.h 中用 typedef 定义相应变量。32 位单片机上，32U 对应 unsigned int，16U 对应 unsigned short，8U 对应 unsigned char。

例如：

```
typedef unsigned char INT8U;  
typedef unsigned int INT32U;  
typedef unsigned short INT16U;
```

代码 1-1 typedef 定义变量

错误类型 3：由非 INTxxxU 型变量未定义引发的错误：

.\ucos\ucos_ii.h(719): error: #20: identifier "xxx" is undefined

解决方案：在 os_cfg.h 里，用#define 补上所需的宏定义。

```
#define OS_MAX_TASKS 64  
#define OS_LOWEST_PRIO 64  
#define OS_TASK_IDLE_STK_SIZE 64
```

代码 1-2 宏定义相关参数值

再次编译，未出现相应错误。

错误类型 4：宏定义中的变量未定义，具体解决办法和错误类型 4 相同。

.\ucos\ucos_ii.h(1482): error: #35: #error directive: "OS_CFG.H, Missing OS_FLAG_EN: Enable (1) or Disable (0) code generation for Event Flags"

对应 ucos_ii.h 中报错的宏定义：

```
#ifndef OS_FLAG_EN  
#error "OS_CFG.H, Missing OS_FLAG_EN: Enable (1) or Disable (0) code  
generation for Event Flags"  
#endif
```

代码 1-3 ucos_ii.h 中检测宏定义错误的相关语句

在 os_cfg.h 中添加

```
#define OS_FLAG_EN 0
```

代码 1-4 在 os_cfg.h 中添加的宏定义

注意：

1. 以 _EN 结尾的变量代表一个开关。设置为 0 表示关闭相应功能，设置为 1 代表启用相关功能。而且任务创建开关必须保证有一个是打开的。

2. 有的变量的上下文附近会有对变量的限制要求。如果把某一个变量定义了，

但是还是会报错，就需要看一看上下文是否对这个变量的取值有特殊要求。

为了改正错误类型 4 而添加的宏定义：

```
#define OS_MAX_TASKS 64
#define OS_LOWEST_PRIO 64
#define OS_TASK_IDLE_STK_SIZE 64
#define OS_FLAG_QUERY_EN 0
#define OS_MBOX_EN 0
#define OS_MBOX_ACCEPT_EN 0
#define OS_FLAG_NAME_EN 0
#define OS_FLAG_EN 0
#define OS_MAX_FLAGS 0
#define OS_FLAGS_NBITS
#define OS_FLAG_WAIT_CLR_EN 0
#define OS_FLAG_ACCEPT_EN 0
#define OS_FLAG_DEL_EN 0
#define OS_MBOX_DEL_EN 0
#define OS_MBOX_PEND_ABORT_EN 0
#define OS_MBOX_POST_EN 0
#define OS_MBOX_POST_OPT_EN 0
#define OS_MBOX_QUERY_EN 0
#define OS_MEM_EN 0
#define OS_MAX_MEM_PART 1
#define OS_MEM_NAME_EN 0
#define OS_MEM_QUERY_EN 0
#define OS_MUTEX_EN 0
#define OS_MUTEX_ACCEPT_EN 0
#define OS_MUTEX_DEL_EN 0
#define OS_MUTEX_QUERY_EN 0
#define OS_Q_EN 0
#define OS_Q_ACCEPT_EN 0
#define OS_Q_DEL_EN 0
#define OS_MAX_QS 1
#define OS_MAX_EVENTS 64
#define OS_Q_FLUSH_EN 0
#define OS_Q_PEND_ABORT_EN 0
#define OS_Q_POST_EN 0
#define OS_Q_POST_FRONT_EN 0
#define OS_Q_POST_OPT_EN 0
#define OS_Q_QUERY_EN 0
#define OS_SEM_EN 0
```

```
#define OS_SEM_ACCEPT_EN 0
#define OS_SEM_DEL_EN 0
#define OS_SEM_PEND_ABORT_EN 0
#define OS_SEM_QUERY_EN 0
#define OS_SEM_SET_EN 0
#define OS_TASK_STAT_EN 0
#define OS_TASK_STAT_STK_SIZE 64
#define OS_TASK_STAT_STK_CHK_EN 0
#define OS_TASK_CHANGE_PRIO_EN 0
#define OS_TASK_CREATE_EN 1
#define OS_TASK_CREATE_EXT_EN 0
#define OS_TASK_DEL_EN 0
#define OS_TASK_NAME_EN 0
#define OS_TASK_SUSPEND_EN 0
#define OS_TASK_QUERY_EN 0
#define OS_TASK_REG_TBL_SIZE 64
#define OS_TICKS_PER_SEC 16000
#define OS_TIME_DLY_HMSM_EN 0
#define OS_TIME_DLY_RESUME_EN 0
#define OS_TIME_GET_SET_EN 0
#define OS_TMR_EN 0
#define OS_TMR_CFG_NAME_EN 0
#define OS_ARG_CHK_EN 0
#define OS_CPU_HOOKS_EN 0
#define OS_APP_HOOKS_EN 0
#define OS_DEBUG_EN 0
#define OS_SCHED_LOCK_EN 0
#define OS_EVENT_MULTI_EN 0
#define OS_TASK_PROFILE_EN 0
#define OS_TASK_SW_HOOK_EN 0
#define OS_TICK_STEP_EN 0
#define OS_TIME_TICK_HOOK_EN 0
```

代码 1-5 在 os_cfg.h 中添加的宏定义完整代码

错误类型 5：因为找不到指定函数导致链接错误：

.\Objects\baogao.axf: Error: L6218E: Undefined symbol

解决方案：查看报错中缺少的函数并添加。因为函数的声明和定义的语法规则区别很大，编译器可以轻松地区分什么是声明，什么是定义。如果分别编译单个文件时，只有声明没有定义，编译器暂时不会报错，会尝试在链接时去其他文件中

找相应的函数定义，但是如果链接时还没有找到函数体，编译器就会报错。

新建一个 `os_cpu.c`，补充缺少的函数定义。

```
#include "os_cpu.h"

void SystemInit(){}
void OSInitHookBegin(){}
void OSInitHookEnd(){}
void OSTCBInitHook(){}
void OSTaskCreateHook(){}
void OSTaskIdleHook(){}
void OSTaskReturnHook(){}
OS_STK *OSTaskStkInit(void (*task)(void *p_arg), void *p_arg, OS_STK
*ptos, INT16U opt){
    return ptos;
}
void OSTaskSwHook(){}
void OS_CPU_ExceptStkBase(){}
void OS_KA_BASEPRI_Boundary(){}

```

代码 1-6 `os_cpu.c` 中与处理器相关的代码

错误类型 6：临界区函数未定义错误：包括

`.\Objects\baogao.axf: Error: L6218E: Undefined symbol OS_ENTER_CRITICAL (referred from ucos_ii.o).`

`.\Objects\baogao.axf: Error: L6218E: Undefined symbol OS_EXIT_CRITICAL (referred from ucos_ii.o).`

warning: #223-D: function "OS_EXIT_CRITICAL" declared implicitly

改正方法：在 `os_cfg.h` 中加上临界区保护函数：

```
extern OS_CPU_SR; //在 os_cfg.h 中定义为 unsigned char 类型。
extern OS_CPU_SR OS_CPU_SR_Save(void); //引用汇编文件 os_cpu_a.asm 中的函数
extern void OS_CPU_SR_Restore(OS_CPU_SR); //引用汇编文件 os_cpu_a.asm 中的函数
extern void OSCtxSw(void); //引用汇编文件 os_cpu_a.asm 中的函数
#define OS_ENTER_CRITICAL() (cpu_sr = OS_CPU_SR_Save())
#define OS_EXIT_CRITICAL() (OS_CPU_SR_Restore(cpu_sr))
#define OS_CRITICAL_METHOD 3u //表示采用第三种临界区保护方式。

```

代码 1-7 `os_cfg.h` 中临界区保护相关代码

② 编译文件并修改相关警告

.\ucos\os_core.c(723): warning: #223-D: function "OSIntCtxSw" declared implicitly

.\ucos\os_core.c(876): warning:#223-D: function "OSStartHighRdy" declared implicitly

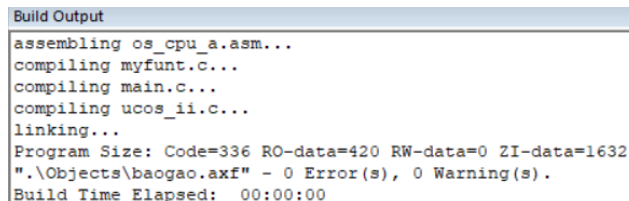
这里需要使用到 `extern` 关键字。这两个函数名标号在其他文件中定义，而在 `os_core.c` 中没有定义，故需使用 `extern` 关键字。

在 `os_core.c` 中：

```
extern void OSIntCtxSw(void);  
extern void OSStartHighRdy(void);
```

代码 1-8 `os_core.c` 中补充 `extern` 关键字部分代码

最终实现了 0 error 0 warning。



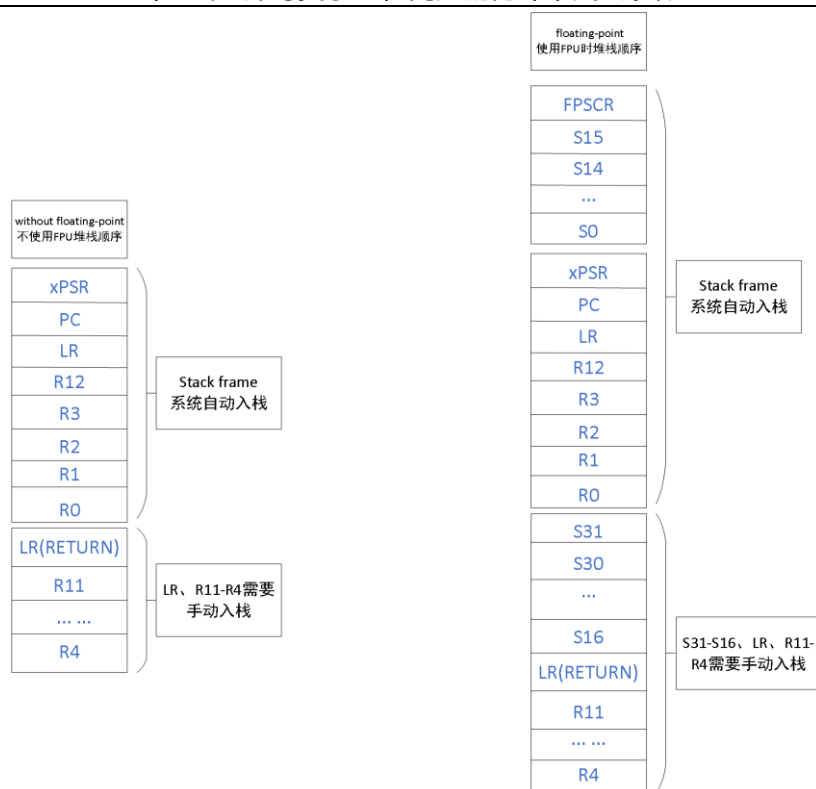
```
Build Output  
assembling os_cpu_a.asm...  
compiling myfunt.C...  
compiling main.C...  
compiling ucos_ii.C...  
linking...  
Program Size: Code=336 RO-data=420 RW-data=0 ZI-data=1632  
".\Objects\baogao.axf" - 0 Error(s), 0 Warning(s).  
Build Time Elapsed: 00:00:00
```

图表 1-9 最终编译结果

1.3.2 实现 STM32 中 μ C/OS-II 任务堆栈初始化和任务上下文切换

1.3.2.1 任务堆栈初始化

当操作系统要建立一个任务时，操作系统首先要对任务的堆栈空间进行初始化，这里的栈是为任务在内存里开辟的一块模拟的堆栈区域，主要用来保存 CPU 的所有寄存器和任务的入口函数地址，在任务切换时，用于保存 CPU 寄存器和程序入口。而系统在进入中断服务程序时，会自动保存一部分寄存器入栈，在堆栈初始化时，我们需要将系统未自动保存的寄存器进行手动入栈。不同情况的堆栈初始化顺序如图表 1-10 所示。



图表 1-10 堆栈初始化顺序

μ C/OS-II 的任务堆栈初始化主要通过 OSTaskStkInit()实现。函数原型为：OS_STK *OSTaskStkInit(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)，初始化时所需的参数 task 为任务执行函数的入口地址，p_arg 是任务参数地址，ptos 是栈顶指针，opt 为预留参数，函数最终返回初始化后的堆栈指针。

在堆栈初始化过程中需要保存任务的起始地址、处理器状态字、EXC_RETURN 参数和寄存器等信息。根据 STM32 的寄存器结构，堆栈初始化函数需要规定在发生上下文切换时寄存器的保存顺序，从而达到保存运行环境的目的。OSTaskStkInit()的具体实现如代码 1-9 所示。

```
OS_STK *OSTaskStkInit(void (*task)(void *p_arg), void *p_arg, OS_STK
*ptos, INT16U opt) {
    OS_STK *stk;
    stk = ptos;    //加载 stack 指针

    if ((opt & OS_TASK_OPT_SAVE_FP) > 0u) {
        //填充 S0-S15,FPSCR,保留位
        memset(stk - 17, 0, 18 * sizeof(OS_STK));
        stk -= 18;
    }
}
```

(1)

```

}

*(stk) = 0x01000000;    //xPSR                                (2)
*(--stk) = (OS_STK) task;    //PC
*(--stk) = 0xFFFFFFFF;    //任务保存的 lr
*(--stk) = 0;    //R12
*(--stk) = 0;    //R3
*(--stk) = 0;    //R2
*(--stk) = 0;    //R1
*(--stk) = (OS_STK) p_arg;    //R0

if ((opt & OS_TASK_OPT_SAVE_FP) > 0u) {                        (3)
    //填充 S16-S31
    stk -= 16;
    memset(stk, 0, 16 * sizeof(OS_STK));
}

if ((opt & OS_TASK_OPT_SAVE_FP) > 0u) {                        (4)
    *(--stk) = 0xFFFFFED;    //中断的 lr
} else {
    *(--stk) = 0xFFFFFFD;    //中断的 lr
}

*(--stk) = 0;    //R11                                (5)
*(--stk) = 0;    //R10
*(--stk) = 0;    //R9
*(--stk) = 0;    //R8
*(--stk) = 0;    //R7
*(--stk) = 0;    //R6
*(--stk) = 0;    //R5
*(--stk) = 0;    //R4
return stk;
}

```

代码 1-9 OSTaskStkInit 函数

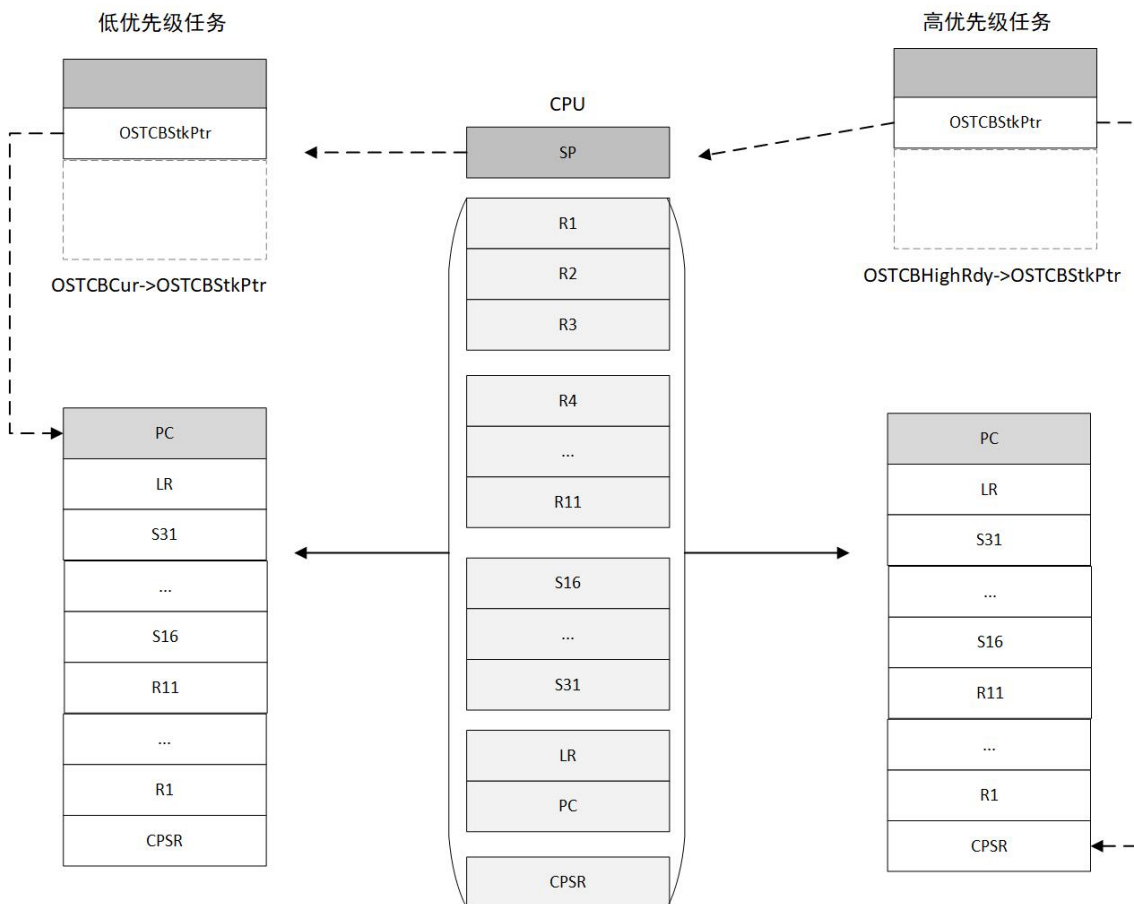
在如代码 1-9 所示 OSTaskStkInit 函数中，我们选择根据参数 opt 的值来决定任务的入栈方式。因为在 STM32 中，采用浮点运算单元要额外保留 FPU 寄存器来完整保存任务环境，否则只需保存通用寄存器。若 opt 的 bit2 为 1，则使用浮点运算单元的方式入栈，否则使用不带浮点运算单元的方式入栈。

在 STM32 任务切换时，硬件会将 xPSR、PC、LR、R12、R0-R3 寄存器自动

入栈，如代码 1-9 L(2)部分即按照硬件寄存器入栈顺序进行寄存器保存,如果开启 FPU，硬件还会额外将 FPSCR、S0-S15 寄存器进行入栈，L(1)根据 opt 的值以及 OS_TASK_OPT_SAVE_FP 判断是否需要保存这些寄存器;L(3)同样判断是否需要额外保存 S16-S31 寄存器;L(4)根据是否采用浮点运算单元保存不同 LR 值;L(5)将 R11-R4 寄存器顺序入栈，最终返回最新的 stk 指针。

1.3.2.2 任务上下文切换

要实现任务上下文切换，不仅需要改变 PC 指针，同时需要保存当前任务的环境，然后恢复将要执行的任务的环境。在 $\mu\text{C}/\text{OS-II}$ 中任务上下文切换分为任务主动切换与中断导致任务被动切换。其中 OSTCtSw()函数用于实现主动切换，如任务 A 切换为任务 B;OSIntCtSw()函数用于实现被动切换，用于从中断退出时切换到下一个任务中。



图表 1-11 上下文切换示意图

如上图所示，进行上下文切换时，依次保存各寄存器到被抢占任务的堆栈中，

堆栈的指针最初为 SP 指向的地址，是上一次任务切换时，根据任务 TCB 的 *OSTCBStkPtr 确定的，保存完后将新的 SP 保存到 OSTCBCur->OSTCBStkPtr 中，将抢占任务的堆栈地址（OSTCBHighRdy->OSTCBStkPtr）给到 SP，恢复其运行环境从而完成任务切换。

OSCtSw:		
OSIntCtSw:		
LDR	R0, =NVIC_INT_CTRL	(1)
LDR	R1, =NVIC_PENDSVSET	
STR	R1, [R0]	(2)
BX	LR	

代码 1-10 OSCtSw 与 OSIntCtSw 汇编语句

在 OSCtSw()函数与 OSIntCtSw()函数实现中，两者采用了一致的实现方法。L(1)触发 PendSV 异常，L(2)向 NVIC_INT_CTRL 写入 NVIC_PENDSVSET 触发 PendSV 中断。

在 OSCtSw()函数实现中，我们可以看到 OSCtSw 实际上是触发了 PendSV 中断，再由 PendSV 中断例程完成真正的任务切换。实际上 $\mu\text{C}/\text{OS-II}$ 的任务上下文切换，只发生在以下几种情况：

- i. 外部中断引起变化，导致高优先级任务就绪。

例如串口中断收到数据，在中断例程中接收数据后调用 OSSemPost()，互斥量的 Post 使 $\mu\text{C}/\text{OS-II}$ 知道相应的串口接收任务转变到就绪状态。

- ii. SysTick 的定时中断，导致之前处于 OStimeDly 挂起的高优先级任务就绪。
- iii. 当前任务执行如 OSSemPend、OStimeDly 等操作导致当前任务挂起，需要切换到其他次优先级的任务。

前两种都是在中断退出后引起任务切换。

外部中断退出时调用 OSIntExit 函数激活 PendSV 中断，真正的 task 切换在 PendSV 中断例程中实现。SysTick 中断优先级较高，也不直接做任务切换，其同样在中断退出时调用 OSIntExit 函数激活 PendSV 中断，真正的任务切换在 PendSV 中断例程中实现。

第 3 种是在需要任务切换时调用 OS_Sched，后者会通过 OS_TASK_SW()调用 OSCtSw，而 OSCtSw 也是激活 PendSV 中断，再由 PendSV 中断例程完成真正的任务切换。

所以 $\mu\text{C}/\text{OS-II}$ 中所有的任务切换最终都是由 PendSV 中断例程完成的，而且

PendSV 的中断优先级必须是系统所有中断中最低的。PendSV_Handler 函数实现如代码 1-11 所示。

```
.thumb_func
PendSV_Handler:
    CPSID I

    MRS r0,PSP
    CBZ r0,PendSV_Handler_Nosave    //first task    (1)

    AND R1, lr, #0x10                (2)
    CBNZ R1,NO_STR_FP
    VSTMDB.32 r0!, {S16-S31}
NO_STR_FP:
    STMFD r0!,{r4-r11, lr}           (3)

    LDR r1,=OSTCBCur
    LDR r2,[r1]
    STR r0,[r2]

    B PendSV_Handler_Nosave

.thumb_func
PendSV_Handler_Nosave:

    LDR R0,=OSPrioCur
    LDR R1,=OSPrioHighRdy
    LDRB R2,[R1]
    STRB R2,[R0]

    LDR R2,=OSTCBCur
    LDR R1,=OSTCBHighRdy
    LDR R0,[R1]
    STR R0,[R2]

    LDR R0,[R0]

    LDMFD r0!,{r4-r11, lr}

    AND R1, lr, #0x10                (4)
    CBNZ R1,NO_LDR_FP
```

```

VLDmia.32 r0!, {S16-S31}
NO_LDR_FP:

MSR    PSP,r0                                     (5)

PUSH {LR}
BL OSTaskSwHook
POP {LR}

CPSIE I
BX LR
NOP

```

代码 1-11 PendSV_Handler 与 PendSV_Handler_Nosave

如代码 1-11 所示，在进入 PendSV 处理函数后，关闭中断，L(1)同时判断 PSP 的值，如果 PSP 为 0 说明是第一次进行任务切换，而任务创建时会调用堆栈初始化函数 OSTaskStkInit()函数来初始化堆栈，在初始化的过程中已经做了入栈处理，所以此处无需再次入栈，直接跳转到 PendSV_Handler_Nosave。L(2)通过判断 EXC_RETURN 的 bit4 即 LR 的 bit4 来确定任务是否使用 FPU，如果任务使用 FPU 的话保存 S16-S31 寄存器。L(3)保存剩余的 R4-R11 寄存器。

在 PendSV_Handler_Nosave 中，首先获取系统中最高优先级任务与任务 TCB，然后从新任务的堆栈中恢复 R4-R11，LR。L(4)判断任务是否使用 FPU，如果使用 FPU 的话将 S16-S31 从堆栈中恢复出来。L(5)用新任务的 SP 加载 PSP。然后跳转到 OSTaskASwHook 函数，可以用于在 SystemView 显示 TCB 信息。

1.3.3 在 STM32 上完成 μ C/OS-II 移植

1.3.3.1 设置滴答定时器 SysTick

滴答定时器是一个倒计时定时器，通过从 RELOAD 寄存器中自动重装载定时器初值来实现定期产生异常请求作为系统的时基。移植 μ C/OS-II 需要使用 SysTick 来作为系统时钟。设置 SysTick 主要通过编写 SysTick 的中断服务函数 SysTick_Handler(),函数实现如代码 1-12 所示。

```

void SysTick_Handler(void) {
    OSIntEnter();           //进入中断
    OSTimeTick();
}

```

```
#if OS_TMR_EN > 0u
    OSTmrSignal();
#endif
#ifdef USE_HAL_DRIVER
    HAL_IncTick();
#endif
    OSIntExit();
}
```

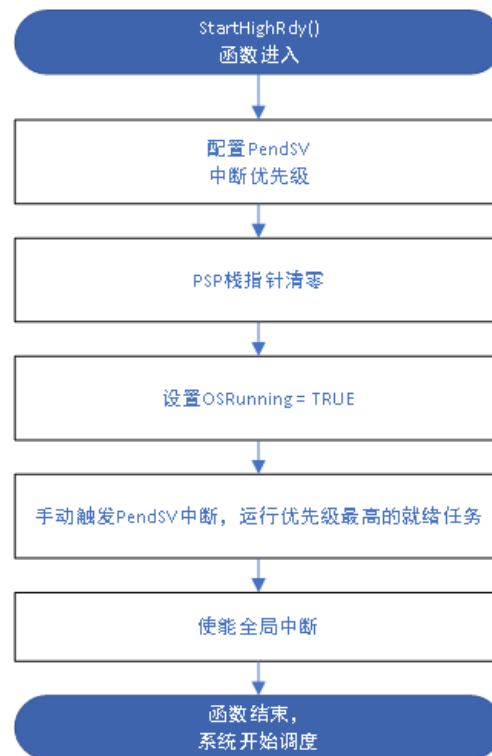
代码 1-12 SysTick_Handler 函数实现

在函数实现中，使用到了条件编译来实现对创建定时器任务、任务切换软中断的选择调用。在 SysTick_Handler 函数中主要是调用 ucos 的时钟服务程序，最后退出中断，触发任务切换软中断。在 ucos 的时钟服务程序通过遍历 TCB 任务链表，修改 TCB 中的延时参数，如果 TCB 中延时时间为 0，那就修改 TCB 状态为就绪态。

1.3.3.2 实现 OSStartHighRdy 函数

μC/OS-II 通过 OSStart()函数开始启动整个系统。在该函数中，当 OS 的运行状态等于错误时，调用 OS_SchedNew()函数查找就绪任务中优先级最高的任务，并将当前运行任务设置为最高优先级任务。同时设置当前 TCB 优先级设置为最高优先级。然后调用 OSStartHighRdy()函数。

OSStartHighRdy()是 OSStart 函数的最关键部分，正是通过这个函数进行任务的启动，实现了系统启动时时运行优先级最高的就绪任务。函数主要实现流程如图表 1-12 所示。



图表 1-12 StartHighRdy 流程图

函数具体实现代码如代码 1-13 所示。

```

OSStartHighRdy
    LDR    R0, =NVIC_SYSPRI14      ; Set the PendSV exception priority
    LDR    R1, =NVIC_PENDSV_PRI
    STRB   R1, [R0]

    MOVS   R0, #0                  ; Set the PSP to 0 for initial
context switch call
    MSR    PSP, R0

    LDR    R0, =OS_CPU_ExceptStkBase ; Initialize the MSP to the
OS_CPU_ExceptStkBase
    LDR    R1, [R0]
    MSR    MSP, R1

    LDR    R0, =OSRunning          ; OSRunning = TRUE
    MOVS   R1, #1
    STRB   R1, [R0]
    
```

第一章 针对复杂工程问题的方案设计与实现

```
LDR    R0, =NVIC_INT_CTRL
LDR    R1, =NVIC_PENDSVSET
STR    R1, [R0]

CPSIE  I
```

代码 1-13 OSStartHighRdy 函数实现

在 OSStartHighRdy()函数中, 首先设置 PendSV 的中断优先级为最小优先级, 然后将任务堆栈指针 PSP 设置为 0。同时修改系统启动标志变量 OSRunning 为 1, 表示系统正在运行。挂起 PendSV 中断, 用于执行系统任务的调度与切换。最后开启中断, 一般情况下, 在启动时系统不会有其他中断, 会立即执行 PendSV 中断处理函数, 进行任务的调度。如果此时有中断, 则在执行外中断后执行 PendSV 中断, 运行就绪队列中优先级最高的任务。

1.3.3.3 实现进出临界区功能代码

在 μ C/OS-II 中需要禁止中断后再访问代码临界区, 并在访问完毕后重新允许中断。在 μ C/OS-II 定义了 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()两个宏来禁止和允许中断。OS_ENTER_CRITICAL/OS_EXIT_CRITICAL 一共实现了三种实现方式, 我们选择使用第三种实现方法, 在关中断前, 使用局部变量来保存中断状态, 函数实现如代码 1-14 所示。

```
#define OS_ENTER_CRITICAL() {cpu_sr = OS_CPU_SR_Save();}
#define OS_EXIT_CRITICAL() {OS_CPU_RestoreSR(cpu_sr);}

OS_CPU_SR_Save:
    MRS R0, PRIMASK
    CPSID I
    BX LR
    NOP

OS_CPU_RestoreSR:
    MSR PRIMASK, R0
    BX LR
    NOP
```

代码 1-14 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()

首先在宏定义 OS_ENTER_CRITICAL()中使用 cpu_sr 保存 OS_CPU_SR_Save 函数的返回值, 在 OS_CPU_SR_Save 函数中保存系统中断状态后关闭中断, 返回

中断状态。在宏定义 `OS_EXIT_CRITICAL()` 中，调用 `OS_CPU_RestoreSR`，传入之前保存的关中断前的中断状态 `cpu_sr`，将系统中断状态恢复为 `cpu_sr`，实现中断的打开。

1.3.4 在 STM32 上实现任务级调试

我们在 STM32 上移植操作系统后，会将各个模块的代码拆分成任务，为了直观观察每个任务的运行时间和运行情况，我们选择了使用 SystemView 进行任务级调试。

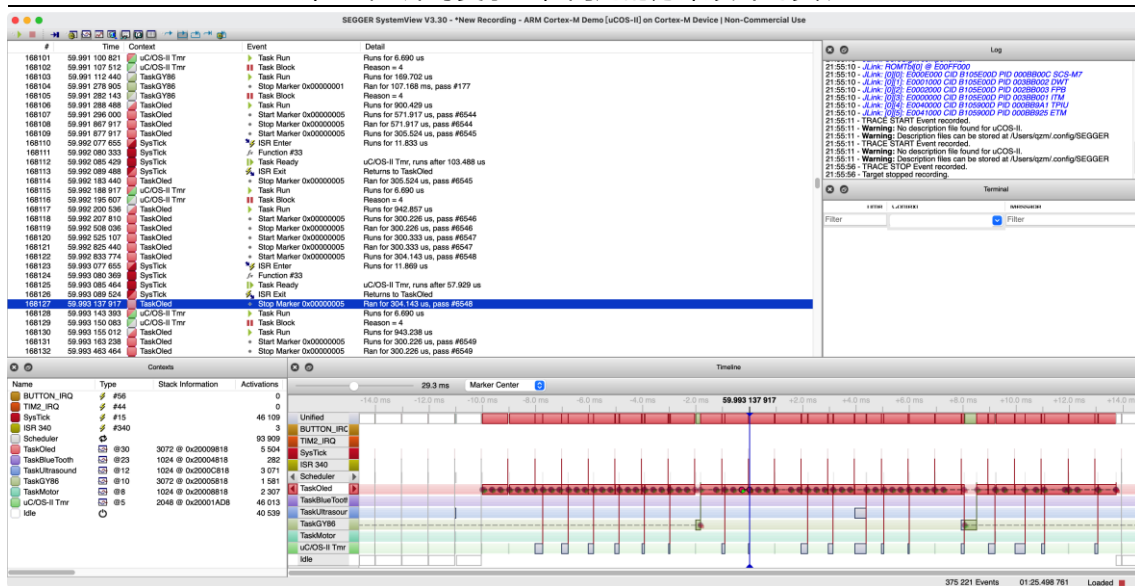


图表 1-13 SystemView 原理图

SystemView 是一个用于对任何嵌入式系统进行可视化分析的工具包。SystemView 能够深入记录分析任务的运行行为，有利于多个任务和事件的复杂系统中的开发和工作。

SystemView 主要包含了两部分：在 PC 上运行的可视化软件和在目标嵌入式系统上运行的代码。在我们 $\mu\text{C}/\text{OS-II}$ 的源代码中，包含了如 `OS_TRACE_TASK_CREATE` 等跟踪函数，在 `os_trace_events.h` 中，通过宏定义将跟踪函数定义为 `SEGGER_SYSVIEW_OnTaskCreate` 等处理函数。在 SystemView 的处理函数中，将关键数据写成数据帧的形式，再通过 `jlink` 调试线发送给上位机，上位机解析收到的数据帧进行可视化显示。

第一章 针对复杂工程问题的方案设计与实现



图表 1-14 SystemView 上位机可视化效果

1.3.5 用 Makefile, HAL 库重构工程

在这个部分我们主要需要完成的工作是为已经移植好的 $\mu\text{C}/\text{OS-II}$ 工程编写一个 makefile 文件并且在需要加入其它文件时修改 makefile 文件以达到将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 makefile 工具就可以轻而易举的理顺各个源文件之间纷繁复杂的相互关系，能够更加有效的帮助我们编译文件。

makefile 的关键就是变量的申明和赋值，比较重要的一些代码部分将在下面举例。

将需要用到的.c 文件的路径用 C_SOURCES 变量的形式声明出来。(如下)

```
C_SOURCES = \
Core/Src/main.c \
Core/Src/gpio.c \
Core/Src/stm32f4xx_it.c \
Core/Src/stm32f4xx_hal_msp.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ramfunc.c \
```

信软学院进阶式挑战性综合项目 II 报告

```
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_gpio.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma_ex.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr_ex.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_cortex.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_exti.c \  
Core/Src/system_stm32f4xx.c \  
Ucos/ucos_ii.c \  
Ucos/os_cpu_c.c \  
Core/Src/dma.c \  
Core/Src/usart.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_uart.c \  
Core/Src/tim.c \  
Core/Src/i2c.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_i2c.c \  
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_i2c_ex.c \  
Drivers/CMSIS/DSP/Source/CommonTables/CommonTables.c \  
Drivers/CMSIS/DSP/Source/BasicMathFunctions/BasicMathFunctions.c \  
Drivers/CMSIS/DSP/Source/FastMathFunctions/FastMathFunctions.c  
  
C_SOURCES += $(wildcard $(patsubst %, %/*.c, $(C_PATH)))
```

由于我们的 μ C/OS-II 的代码中不仅有.c 文件还有.s 文件.asm 文件，我们需要按照上面的方式声明。需要注意的是一般来说对于汇编语言我们只会声明 ASM_SOURCES 这一个变量，但是由于我们的文件中有多种汇编文件所以在 makefile 编写时我们将它分为了两个。

```
# ASM sources  
S_SOURCES = \  
startup_stm32f401xe.s \  
Systemview/SEGGER/SEGGER_RTT_ASM_ARMv7M.s  
  
ASM_SOURCES = \  
Ucos/os_cpu_a.asm
```

还有一些诸如此类的变量的声明都是用于之后文件的编译。

```
C_INCLUDES = \          AS_INCLUDES =
```

Makefile 文件的起到文件编译的部分，这里只举两个例子。通过代换完成 make 的完整形式，可以看到我们在编译后将结果作为\$(BUILD_DIR)输入。

```
$(BUILD_DIR)/%.o: %.c Makefile | $(BUILD_DIR)
    $(CC) -c $(CFLAGS) -Wa,-a,-ad,-alms=$(BUILD_DIR)/$(notdir
$(<:.c=.lst)) $< -o $@
$(BUILD_DIR)/%.o: %.s Makefile | $(BUILD_DIR)
    $(AS) -c $(CFLAGS) $< -o $@
```

文件中比较重要的部分就如上述，下面要叙述一个 makefile 文件的优化，我们在开发的时候需要将本次工程的代码分成多个子目录来编写，但是在 Makefile 的编写上却是个问题，如果一个一个的去添加编译文件，这是一件极其麻烦的事情，所以我们选择在 makefile 文件中构建带有子文件夹的源代码目录的自动扫描编译。这里仍然举两个例子，有这两行代码就能实现对于 C_SOURCES, C_INCLUDES 包含文件的文件夹进行自动扫描编译。

```
C_SOURCES += $(wildcard $(patsubst %,/*.*c,$(C_PATH)))
C_INCLUDES += $(patsubst %, -I%, $(C_PATH))
```

1.3.6 在 μ C/OS-II 上移植外设代码

在移植了操作系统后，我们将外设相关代码从轮询方式修改为任务的方式运行，并使用信号量+定时器让任务以一定的频率被调度。

这里以 OLED 任务为例，任务开始时会创建信号量和以 200 毫秒为周期的定时器，每成功申请一次信号量，OLED 显示屏会刷新一次显示内容。定时器每计时 200 毫秒，释放一次信号量，从而实现 OLED 以 5Hz 的频率刷新显示内容。

```
OS_EVENT *sem_oled_task;

void app_oled_tmr_callback(void *ptmr, void *parg) {
    OSSemPost(sem_oled_task);
}

void *app_oled_task(void *args) {
    uint8_t err;
    OS_TMR *tmr_oled_task;
    sem_oled_task = OSSemCreate(0);
    tmr_oled_task = OSTmrCreate(0, 200, OS_TMR_OPT_PERIODIC,
app_oled_tmr_callback, (void *) 0, "OLED_TASK_Tmr", &err);
    OSTmrStart(tmr_oled_task, &err);

    while (1) {
```

```

        OSSemPend(sem_oled_task, 0, &err);
        app_oled_Show_Data();
    }
}

```

代码 1-15 OLED 任务代码

1.3.7 系统优化

1.3.7.1 外设代码可裁剪设计

在移植 $\mu\text{C}/\text{OS-II}$ 的过程中,我们小组体会到 $\mu\text{C}/\text{OS-II}$ 代码可裁剪设计的好处,使用宏定义的方式,选择是否编译部分代码进行可裁剪设计,我们决定将这一特性运用到我们的外设代码中。在我们第三学期的外设代码中,包含有 GY86、接收机、电机等四轴飞行器飞行必需外设,也包括 OLED、蓝牙、RGB 指示灯等非必要外设。在调试阶段,可以打开 OLED、蓝牙等功能方便调试,在最终使用环境中,可以方便的关闭这部分功能,让有限的 CPU 资源更集中的用于核心功能。我们决定在 app_cfg.h 文件添加如 DEVICE_XX_EN 的宏定义,选择是否编译外设代码与创建任务。

```

#ifdef DEVICE_OLED_EN > 0
    OSTaskCreate(app_oled_task,
                (void *) 0,
                TaskOledStk + LARGE_STK_SIZE - 1,
                TASK_OLED_PRIO);
    OSTaskNameSet(TASK_OLED_PRIO, (INT8U *) "TaskOled", &err);
#endif

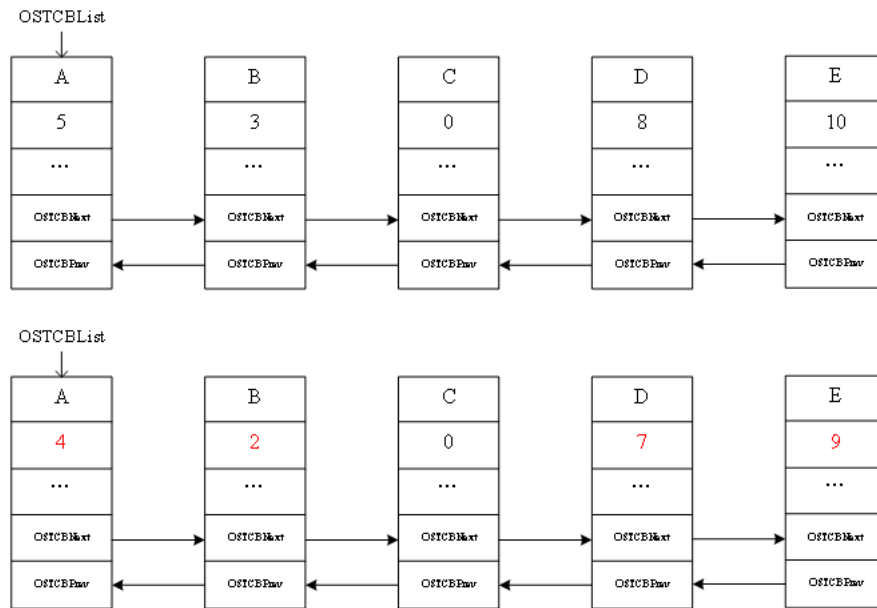
```

代码 1-16 使用了可裁剪设计的 OLED 创建任务代码

1.3.7.2 新增差分时间等待链结构

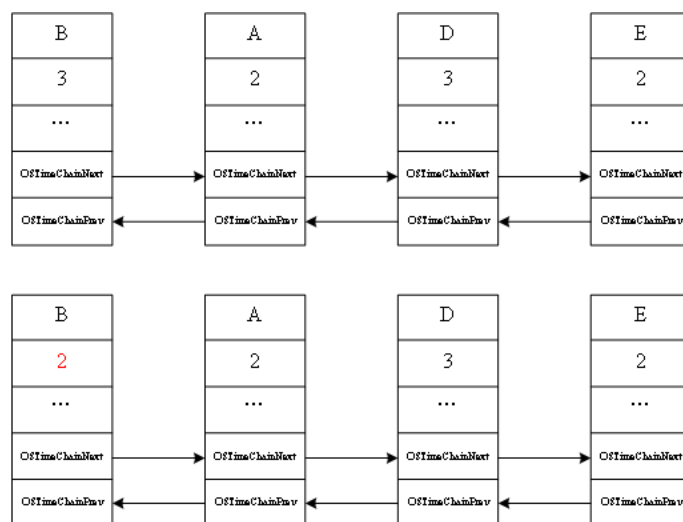
我们在嵌入式操作系统的课程中学习了:为提高实时任务响应性,可以使用差分时间等待链的数据结构对延迟队列进行优化。

在 $\mu\text{C}/\text{OS-II}$ 源码中,每个时钟中断会遍历一遍 OSTCBLList 中所有任务,若 OSTCBDly 不为 0,则对其进行-1 操作,随着任务数的增加,延迟操作所需的时间也会相应延长。



图表 1-15 $\mu\text{C}/\text{OS-II}$ 原任务队列中时钟中断对延迟操作示意图

如果我们使用差分时间等待链，每个时钟中断中仅需对时间链中的第一个任务延时进行-1 操作，延迟操作所需时间确定，在多任务环境下能缩短时钟中断的执行时间。



图表 1-16 使用差分时间等待链时时钟中断对延时操作示意图

因此，我们定义了 `os_timechain` 结构体，通过构建双向链表来保存差分时间等待链，其中 `tcb` 参数存放对应的任务控制块指针，`dly` 参数用于存放差分后的等待时间，`OSTimeChainNext` 和 `OSTimeChainPrev` 参数用于构建双向链表，`tcb` 块中原

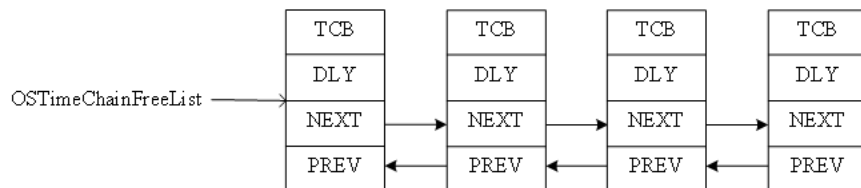
有的 OSTCBDly 参数用于存放原延迟时间。

```
typedef struct os_timechain {
    OS_TCB *tcb;
    INT32U dly;
    struct os_timechain *OSTimeChainNext;
    struct os_timechain *OSTimeChainPrev;
} OS_TIMECHAIN;
```

代码 1-17 os_timechain 结构体

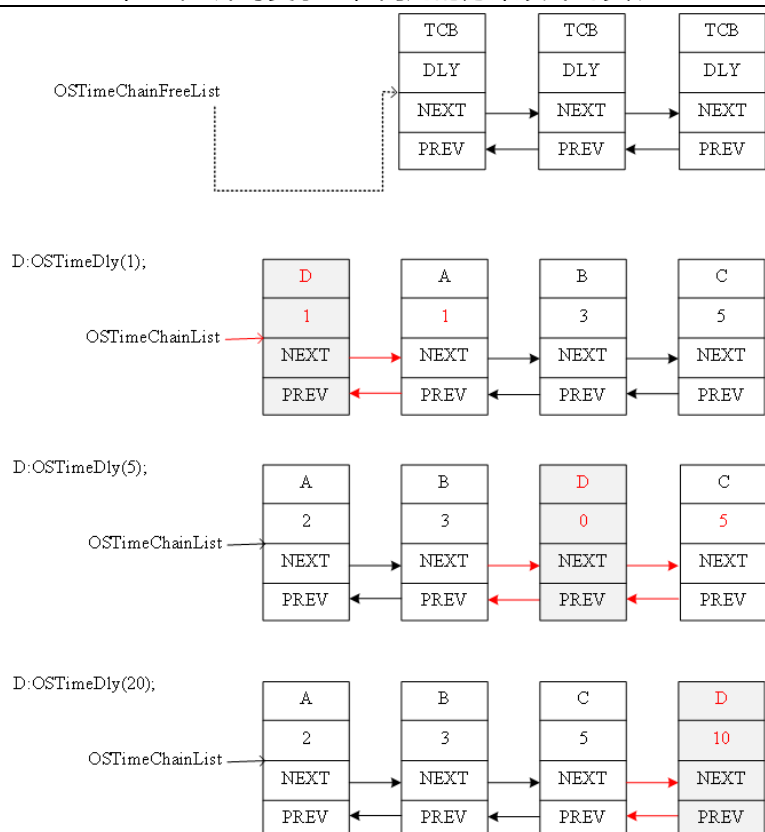
其中，dly 参数保存的差分延迟时间用于在时钟中断中进行-1 操作，当差分时间链头结点的 dly 值为 0，则代表该任务延迟时间已到。OSTCBDly 参数中保存的原延迟时间作为该 TCB 是否在延迟队列中的标志使用，仅标识该任务是否处于延时状态，并不进行实际的延迟操作。

我们还参考 $\mu\text{C}/\text{OS-II}$ 的思想，使用了延时块数组，以在编译链接时分配好对应的空间大小。使用 OSTimeChainFreeList 记录空闲延时块的链表头，使用 OSTimeChainList 记录差分时间链链表头。



图表 1-17 OSTimeChainFreeList 完成初始化后示意图

我们发现，除了在 OSTimeDly 让任务延时，还有互斥量等待、信号量等待中需使用到延时操作。因此，我们在 os_time.c 中，我们实现了 OSTimeDlyLink 函数，将任务添加到差分时间等待链，需要使用延时功能的地方调用此函数即可。在 OSTimeDlyLink 函数中，首先判断当前任务是否处于延时状态，若是则相当于重设等待时间，先从 OSTimeChainList 中去除该任务。再判断等待时间是否为 0，为 0 则不进行操作。判断完成后，将从 OSTimeChainFreeList 中取出一个延时块，设置 TCB 并在 TCB 中的 OSTCBDly 标记当前处于延时状态，若 OSTimeChainList 为空或等待时间小于 OSTimeChainList 中第一个任务的等待时间，则将延时块插入到链表头，否则遍历链表将延时块插入到恰当位置。

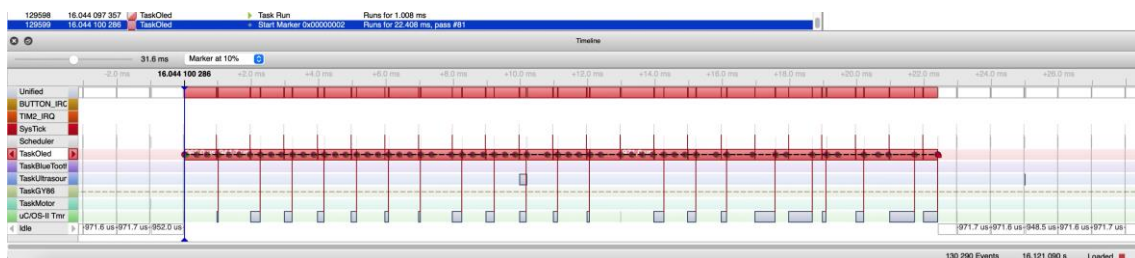


图表 1-18 三种不同延时时间的任务插入到差分时间链的情况示意图

同样，我们还编写了 `OSTimeDlyUnLink` 函数，用于取消某个任务的延时，将任务从差分时间链去除，实现方法与添加任务到差分时间链类似，本文不再描述。

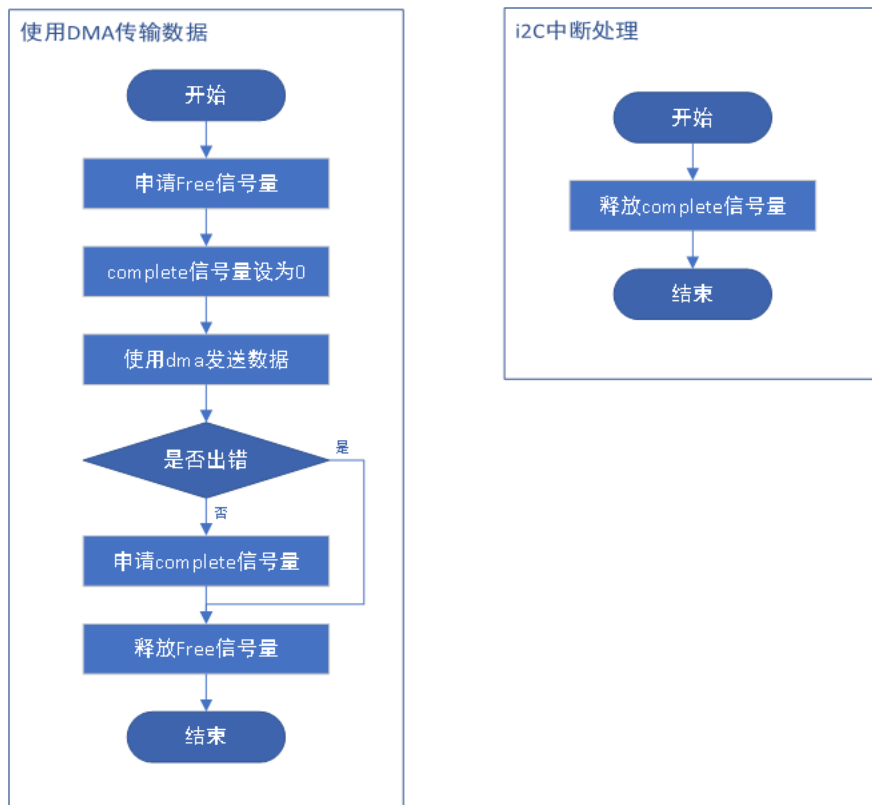
1.3.7.3 使用 DMA 传输数据并支持传输过程任务调度

在实际使用测试过程中，我们发现 OLED 显示任务占用 CPU 资源较多，而任务中大部分时间都用在与 OLED 进行 I2C 通信上，我们决定使用 DMA 进行 I2C 数据传输，并在 DMA 发送数据的同时，操作系统进行任务调度，最大化的利用 CPU 资源。



图表 1-19 SystemView 上观测的 OLED 任务运行情况

我们编写了 I2C 的 DMA 发送函数，首先申请 hi2cFreeSem 空闲信号量，确认 I2C 目前处于空闲状态，然后设置 hi2cCompleteSem 完成信号量为 0，避免上一次 I2C 发送异常后信号量未清空的情况，随后使用 DMA 发送数据，申请 hi2cCompleteSem 信号量，申请成功即为数据发送完成，随后释放 hi2cFreeSem 空闲信号量，完成 DMA 发送。我们在 I2C 完成中断和错误中断中释放 hi2cCompleteSem 信号量，表示数据通过 DMA 发送完毕。



图表 1-20 使用 DMA 传输数据流程图

第二章 系统测试

本章主要从测试环境、 $\mu\text{C}/\text{OS-II}$ 移植测试、SystemView 移植测试、差分时间链测试四个方面对项目进行分析和介绍。

2.1 测试环境

本章在操作系统运行环境下进行详细测试，为防止四轴飞行器在操作系统运行工程中出现故障，本章通过构建使用场景、构造使用用例和制定合理的测试方案对系统来进行测试，以保证系统在实际运行中的正确性、可用性、可靠性和有效性，避免错误，并尽可能的降低延迟。

测试环境参数及测试工具如下表所示。

测试环境类型	型号或版本号	参数	
硬件环境	STM32F401RET6	内存	512KB
软件环境	SystemView	V3.32	

表格 2-1 测试环境参数表

2.2 $\mu\text{C}/\text{OS-II}$ 移植测试

我们将移植 $\mu\text{C}/\text{OS-II}$ 分为三个步骤，首先关闭 STM32 的浮点运算功能，任务上下文切换时仅保存通用寄存器，测试 $\mu\text{C}/\text{OS-II}$ 初步移植的准确性，随后加入上下文切换时保存浮点寄存器的功能，测试带有浮点任务的切换，最后进行所有外设功能的检验，保证所有功能任务创建成功且能正常运行。

2.2.1 任务调度测试

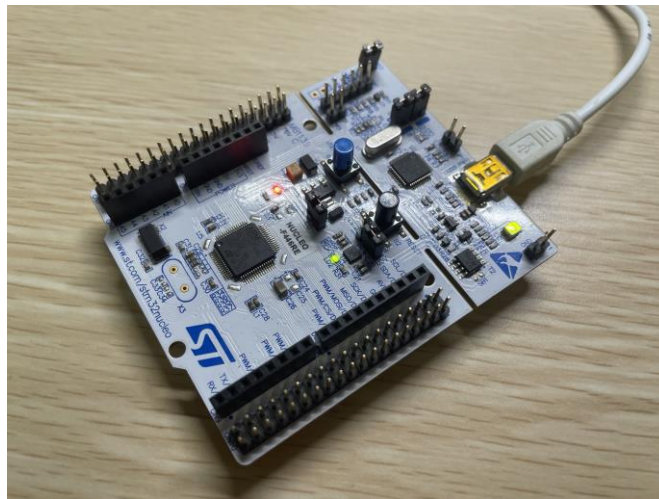
初步移植完成后，我们设计了两个任务，一个是亮灯任务，延时 100 毫秒，一个是灭灯任务，延时 150 毫秒，观测板载 LED 灯是否正常闪烁，若正常闪烁则 $\mu\text{C}/\text{OS-II}$ 移植正确。

```
void led_on_task(void){
    while (1){
```

```
        led_on();
        OSTimeDly(100);
    }
}

void led_off_task(void){
    while (1){
        led_off();
        OSTimeDly(150);
    }
}
```

代码 2-1 任务调度测试时的任务代码



图表 2-1 板载 LED 灯正常闪烁

2.2.2 含浮点数任务测试

任务调度测试完成后，我们添加了保存浮点寄存器的部分，在测试中会生成等待时间不同的 10 个浮点数运算任务，每个浮点数运算任务时间执行时间均大于 1 毫秒，确保会发生一个任务被另一个任务打断的情况。在任务中会计算 10000 个 $\sin 90^\circ$ 的和，考虑到浮点误差，若结果大约为 10000 左右，则计算正确，否则计算错误，点亮 LED 灯，代表含浮点数的任务上下文切换代码出现问题。

第二章 系统测试

```
void app_floatpoint_correctTest_task(void *args) { //输入参数为等待时间
    INT16U delayMSecond = 20;
    if (args != 0) {
        delayMSecond = (INT16U) args;
    }
    while (1) {
        float a, res = 0;
        uint32_t i = 0;
        for (i = 0; i < 10000; ++i) {
            a = sinf(3.14f / 2);
            res += a;
        }
        if (res < 9999.95f || res > 10000.05f) {
            led_on();
        }
        OSTimeDlyHMSM(0, 0, 0, delayMSecond);
    }
}
```

代码 2-2 含浮点数任务测试相关代码

程序运行 5 分钟，LED 灯未被点亮，测试通过，含浮点寄存器的上下文切换代码编写正确。

2.2.3 外设功能测试

对于外设功能，本章主要为 3 大功能模块：电控模块、数据传输模块、OLED 显示模块设计了相应的测试用例进行测试，以保证每个功能模块的正确性和可靠性。

用例描述	对电控模块相应功能进行测试		
用例目的	测试遥控、电机功能能否正确运行		
前提条件	操作系统运行正常		
用例编号	输入/动作	期望的输出/响应	测试结果
DK001	遥控前进	对应电机以正确方向旋转	测试通过
DK002	遥控后退	对应电机以正确方向旋转	测试通过
DK003	遥控左转	对应电机以正确方向旋转	测试通过
DK004	遥控右转	对应电机以正确方向旋转	测试通过

表格 2-2 电控模块功能测试

信软学院进阶式挑战性综合项目 II 报告

用例描述	对数据传输模块相应功能进行测试		
用例目的	测试 GY86、蓝牙等功能能否正确运行		
前提条件	操作系统运行正常		
用例编号	输入/动作	期望的输出/响应	测试结果
SC001	GY86 自动数据采集	数据在 OLED 正确显示	测试通过
SC002	蓝牙数据传输到上位机	数据在上位机正确显示	测试通过

表格 2-3 数据传输模块功能测试

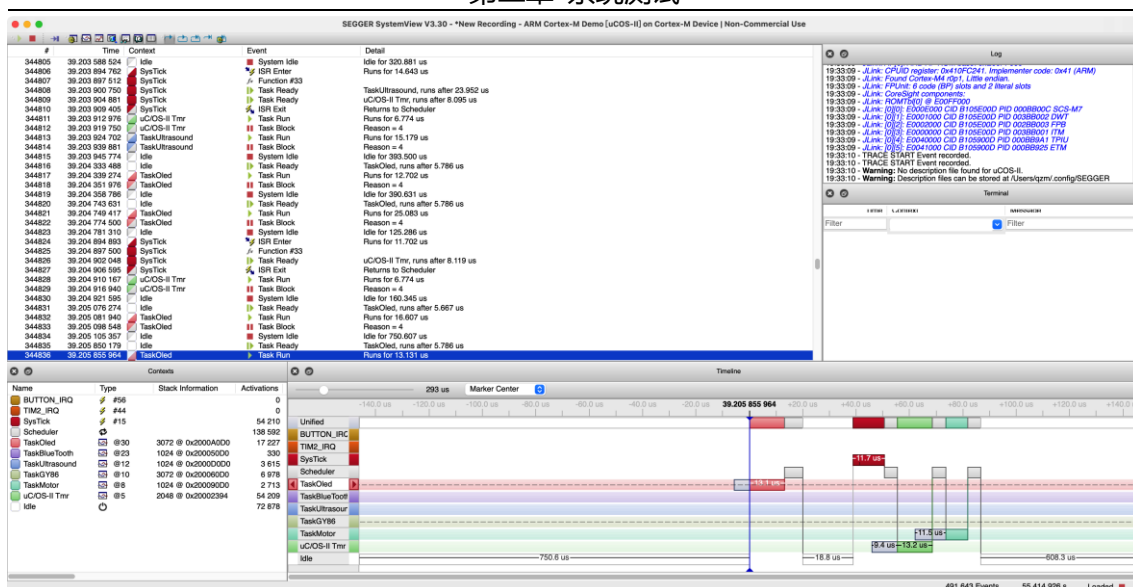
用例描述	对 OLED 显示模块相应功能进行测试		
用例目的	测试 OLED 显示功能能否正确运行		
前提条件	操作系统运行正常		
用例编号	输入/动作	期望的输出/响应	测试结果
XS001	按下刷新按钮	OLED 数据刷新	测试通过
XS002	切换显示界面	对应数据显示	测试通过

表格 2-4 OLED 显示模块功能测试

2.3 SystemView 移植测试

在移植 SystemView 源码到评估板后，将评估板连接电脑，使用 SystemView 成功监测各任务运行状态，获取重要代码运行时间及打印指定信息。

第二章 系统测试



图表 2-2 SystemView 运行截图

2.4 差分时间链测试

编写完差分时间链相关代码后，我们对其进行了正确性测试和性能测试，确保其能达到我们的预期效果。

2.4.1 正确性测试

我们使用了多种方式对其进行测试，以保证其正确性。

2.4.1.1 GDB 调试法

我们使用 GDB 对差分时间链代码进行单步调试，在将任务添加到差分时间等待链后，通过打印等待链、延时块等相关参数数据，观察数据是否符合预期，成功排查一处错误，再次测试后无异常。

2.4.1.2 Systemview 观察法

对于差分时间链，一个任务进入时间链后，可能时间链为空，也可能被插入到链首、链中、链尾等位置，情况复杂，如果代码出现错误，则可能出现任务无法再次被调用、任务等待时间与设置时间不符等问题。

根据差分时间链的特点，我们组采用了动态测试的方法对代码正确性进行测试，我们设计了 10 个 LED 闪灯任务，对于 task1，闪灯后延时为 2 毫秒，对于

task2-task9, 闪灯延时取值范围从 10 毫秒到 24 毫秒, 对于 task10, 闪灯延时为 100 毫秒。通过 systemView 观察每个任务临近几次执行的开始时间间隔, 如果间隔与设置的延时一致, 则差分时间链设置正确, 否则差分时间链代码出现错误。因为闪灯代码的操作时间极短, 若多个任务在同一时间转换为就绪状态, 多个任务在 0.1 毫秒内都能执行完成转换回等待队列, 对系统整体延时和调度观察无影响。

任务编号	延时时间 (毫秒)	实际运行间隔时间 (毫秒)
1	2	2
2	10	10
3	12	12
4	14	14
5	16	16
6	18	18
7	20	20
8	22	22
9	24	24
10	100	26

表格 2-5 各任务设置的延时时间与实际延时时间记录表

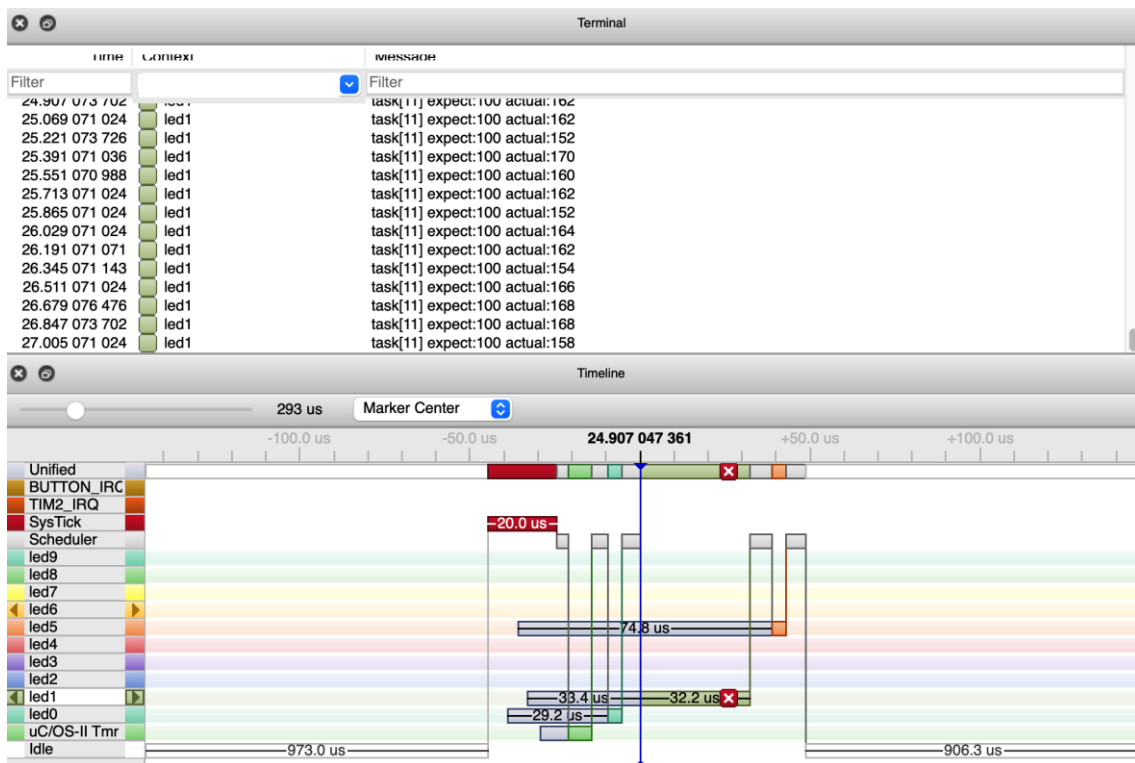
2.4.1.3 嵌入测试代码自动监测法

为了进一步提升测试效率与准确性, 我们在 led 闪烁任务中添加了时间戳间隔监测功能, 每次执行后, 使用局部变量记录当前时间戳, 等待再次执行时, 计算时间戳的差值, 判断实际等待时间与期望等待时间是否相等, 如果不相等则向上位机打印错误信息。

第二章 系统测试

```
void app_led_toggle_monitor_task(void *args) {
    INT16U delayMSecond = 20;
    uint32_t lastTick = 0;
    if (args != 0) {
        delayMSecond = (INT16U) args;
    }
    while (1) {
        if (lastTick != 0 && HAL_GetTick() - lastTick != delayMSecond) {
            SEGGER_SYSVIEW_ErrorfTarget("task[%d] o:%d n:%d", OSPrioCur,
            delayMSecond, HAL_GetTick() - lastTick);
        }
        lastTick = HAL_GetTick();
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
        OSTimeDlyHMSM(0, 0, 0, delayMSecond);
    }
}
```

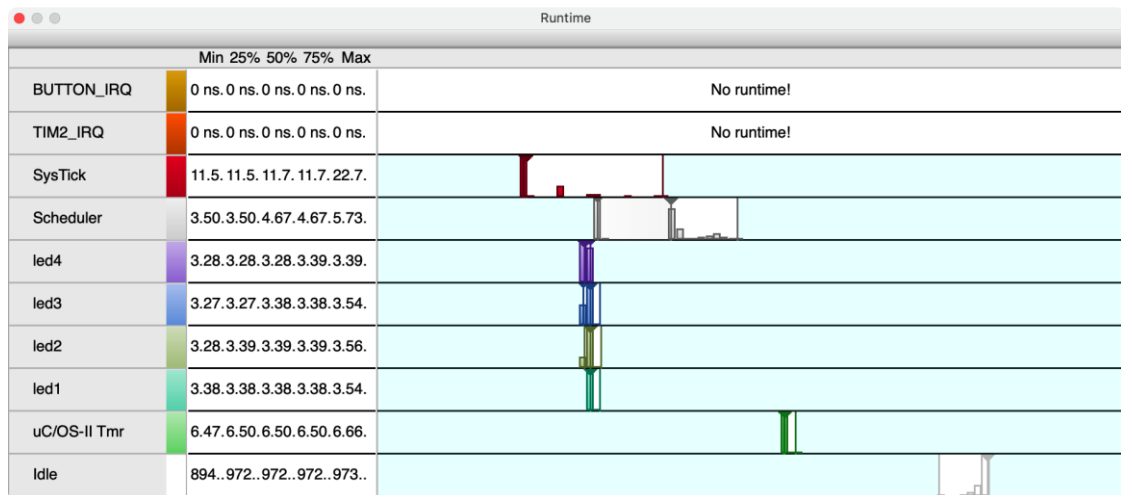
代码 2-3 自动监测等待时间的相关代码



图表 2-3 当差分时间等待链出现错误时，可以在 SystemView 中监测出错误信息

2.4.2 性能测试

我们使用差分时间链和原延时队列测试比较了在 4 个任务和 10 个任务的情况下，使用 SystemView 观察时钟中断的运行时间。



图表 2-4 四个任务使用差分时间链条件下，SystemView 监测的中断运行时间截图

时钟中断执行时间	4 任务	10 任务
原延时队列	11.7ms	13.2ms
差分时间链	11.7ms	15.4ms

表格 2-6 对于 50%统计数据时钟中断执行时间对比表

时钟中断执行时间	4 任务	10 任务
原延时队列	11.7ms	13.2ms
差分时间链	14.4ms	18.0ms

表格 2-7 对于 75%统计数据时钟中断执行时间对比表

通过上述图表可以发现，对于超过 50%的情况，随着任务数的增加，使用差分时间链时，在时钟中断中对于任务延时的处理开销基本相同，而使用原延时队列，在时钟中断中对于任务延时的处理开销逐渐增长。

针对四轴飞行器这种多任务的工程，使用差分时间链可以减少调度所用时间，实现更加快速的调度，进一步保证实时性。

第三章 知识技能学习情况

本章将由每位组员谈谈自己的学习情况。

贺奕嘉

在完成综设任务期间我遇到了一些问题，如：修改 $\mu\text{C}/\text{OS-II}$ 源代码时，有时候会遇到较难解决的问题，现在汇总如下：

1) 某些编译报错信息第一次碰到后看不懂错误原因，不会改，于是把相关错误原因在网上和书中（主要是《嵌入式操作系统》和《难点标记-嵌入式实时操作系统 $\mu\text{C}/\text{OS-II}$ 原理及应用》这两本书中）查找，按照查到的修改方法修改，但是发现还会报错，错误并没能成功解决。这种问题出现的原因是，网上或书中的项目代码和我们的项目代码不完全相同，修改其他项目代码的方法不一定适用于我们的工程。

2) 某些错误信息在网上和书中根本查不到，网上和书中没有这些错误的记录。

面对这两种问题，由于没有可靠参考资料，只能想办法自己解决，我的做法是：

第一步，由于错误信息的描述方式是英文，所以错误信息中每一个不认识的英文单词都要查询，准确把错误信息翻译成中文，然后结合自己的计算机知识，思考为什么编译器会抛出这种错误；

第二步，把与问题有关的代码段中的一些关键词放到 `ucos` 的多个文件中搜索，并想办法弄明白 `ucos` 的各个文件的作用和联系（包括询问同学，上网查找，查阅书籍），思考这一段代码的作用，以及程序文档中哪些代码段与这段问题代码相关联；如果只有错误信息，编译器没有给出发生错误的代码段，则把大部分精力放在思考 `ucos` 的各个文件的作用和联系上，以及哪个功能模块可能出现了问题。

第三步，独立思考这个错误具体该怎么修改，方法和网上的、书中的可能不一样，并尝试多种修改方法，直到问题解决。

3) 最后 0 error，但是有 warning，而且由于 warning 的存在，最终链接失败，没能生成可执行文件。而且在网上或书中查询 warning 的描述信息后，查到的解决方案很多，但是都不能解决遇到的问题，甚至按照参考方法进行修改后，error 和 warning 更多了！看来 warning 并不能不管。

具体操作和解决问题 1、2 的方法类似，先把描述信息翻译成中文，结合不同项目代码文档的作用和联系，思考解决办法。出现问题 3 的原因大概率是：每一个

文件都没有语法错误，都能通过编译，但是整合到一起就出问题了，文件之间不能很好地相互配合。

米锦江

在进行 $\mu\text{C}/\text{OS-II}$ 移植过程中，首先遇到的问题便是不知如何下手，从哪个方面开始移植。通过课上老师与同学的分享，对操作系统底层原理和基本移植工作有了初步的了解。然后在查询 $\mu\text{C}/\text{OS-II}$ 的基本内容后，明白系统启动用到的各文件的基本作用，如 `os_cpu_a.asm` 文件是实现了操作系统所需要的汇编层的功能函数，以及需要结合 STM32 知识对各功能函数进行改写，以实现在 STM32 环境上运行 $\mu\text{C}/\text{OS-II}$ 操作系统。遇到的第二个问题是，需要知道对处理器架构有一定的了解，才能对各文件中需要移植的函数进行改写。通过查询 `cortex-m4` 手册和《嵌入式实时操作系统 $\mu\text{C}/\text{OS-II}$ 原理及应用》，对 `cortex-m4` 的通用寄存器、堆栈和 `PendSV` 异常等基本知识有一个初步的掌握。在对所要做的工作内容以及基本知识有一定的了解后，就依据 $\mu\text{C}/\text{OS-II}$ 源码，一个个编写所要改写的函数。其中在编写汇编层代码时，参考了 $\mu\text{C}/\text{OS-II}$ 开发手册，进一步了解底层原理的实现。

屈子铭

在编写含浮点数的上下文切换代码时，对 32 的浮点数寄存器还不了解，中断中 LR 的保存的 `EXC_RETURN` 值还未理解透彻，导致一开始编写代码时总是无法正常运行。在查阅了《ARM Cortex-M3 与 Cortex-M4 权威指南》和芯片手册后，了解到我们的板子上是含有 32 个单精度浮点寄存器或两两组合成 16 个双精度浮点寄存器，进入中断时，CPU 会在堆栈中预留 16 个单精度浮点寄存器的空间，当中断中使用到了浮点数，硬件会将 16 个单精度浮点寄存器入栈，而剩下 16 个浮点寄存器需要手动入栈。我还通过 GDB 单步调试跟踪寄存器的值发现，当运行到浮点相关的语句时，`control` 寄存器的浮点位会被自动置 1，此后进入中断时，会触发上述的保存浮点寄存器机制，LR 保存的 `EXC_RETURN` 中浮点位会被置位，因此在中断中可以通过该位判断上文任务是否使用了浮点运算，从而决定是否保存剩余 16 个浮点寄存器。

厉浩然

在进行移植的过程中，遇到的比较大的问题是不知道如何将学到的知识技能转化到移植的项目中去。在嵌入式操作系统和 ARM 处理器原理与实践的课程中，我们学习了 2440 等处理器搭载操作系统所需的函数，但在进行 $\mu\text{C}/\text{OS II}$ 移植时，却不知道该如何应用。通过查阅数据手册及各处理器之间的对比，我们了解了对应不同的堆栈架构和寄存器架构应采用不同的函数编写，实现了知识的化

用，一通百通。

杨坤

在整个操作系统的移植过程中遇到了一些问题，首先就是关于移植操作系统需要添加的文件，有部分同学是添加了库文件有一部分同学没有添加库文件，这导致了移植成功的评判标准不一致，在研读了嵌入式操作系统中关于 uc0s 移植和 csdn 上的一些文章后成功的解决了这个问题，最后的结论应该是相应的库文件应该是需要使用时在加入。

第四章 分工协作与交流情况

本章将讲述我们小组的分工协作情况以及每位组员的组内组外交流情况。

4.1 分工情况

为了让每位组员都能在 STM32 上通过移植,了解操作系统的原理及移植方式,我们小组每位同学都会完整的移植一次 ucOS-II 到评估板上,并主攻其中一个任务。

姓名	主攻任务
屈子铭	外设代码移植、SystemView 代码移植、系统优化
米锦江	在 STM32 上移植 μ C/OS-II
厉浩然	编写任务堆栈初始化和上下文切换代码
贺奕嘉	重构 μ C/OS-II 新工程
杨坤	使用 Makefile 重构工程

表格 4-1 小组主攻任务分配表

4.2 交流情况

贺奕嘉

贺奕嘉同学完成了对 ucOS-II 的重构。

在这期间,我在改代码的过程中发现自己缺少芯片启动汇编文件和系统底层的 os_cpu_a.asm 文件,其他组员提供给这两个代码文件。

而且,同组组员还给了我部分错误信息的一些修改思路,给我讲解了部分变量的作用,引导我深入学习 C 语言中的各种宏定义。

聆听班上其他同学上台讲解知识时,我有时会提出一些新问题和个人想法,如: PendSv()函数是把整个异常处理程序在这个函数中处理完,还是跳转到相应的异常处理程序入口?那些优先级小于 0 的任务是否只存在于底层芯片系统中,在 ucOS 操作系统中没有?

当同学分享完知识后,偶尔我会去询问他具体是如何学到这些知识的,询问他查询资料时的思路 and 独特方法。

米锦江

在学习过程中,通过与厉浩然同学的交流,初步拆解并分析了移植要解决的问题,将移植工作按照系统底层原理逐步分为一个个小模块。在实现具体细节时,遇到了问题,如对 STM32 堆栈了解不够清晰、PendSV 处理函数的具体实现等问题,通过咨询屈子铭同学与其他组同学,最终完成具体的实现。

屈子铭

在移植过程中,我们小组会一起探讨操作系统运行到哪一步时,他的变量值会怎么变化,他应该要怎么变化,更加细致透彻的了解 RTOS 运行原理。除此之外,我还和其他小组交流了一些模块的实现逻辑,发现我会忽略一些小细节,在我自己的板子上可能跑得起来,但在其他同学的板子上可能就跑死了,通过交流修改得以提高整套系统的健壮性。

厉浩然

在学习过程中,通过与米锦江同学的交流,初步拆解并分析了移植要解决的问题,将移植工作按照系统底层原理逐步分为一个个小模块。在实现具体细节时,遇到了问题,如对进入临界区的三种方法不清楚、PendSV 处理函数的具体实现等问题,通过咨询屈子铭同学与其他组同学,最终完成具体的实现。

杨坤

在整个项目实践中,我主要负责了用 makefile 重构项目,采取的主要方法是在上 csdn 找参考文献,理清了 makefile 的理论基础,在自己的 makefile 文件还不完善的情况下主动向组长请求帮助,研读已经写好的文件,学习后再为自己的工程创建了 makefile 文件,并从组长接收到了优化 makefile 的方法,遍历文件夹的自动化编译,在网上学习相关知识后对于自己的 makefile 文件进行的更新和优化。

参考文献

- [1] 廖勇, 杨霞. 嵌入式操作系统[M]. 1. 高等教育出版社, 2017.
- [2] 任哲. 嵌入式实时操作系统 $\mu\text{C}/\text{OS-II}$ 原理及应用[M]. 1. 北京航空航天大学出版社, 2009.
- [3] Yiu J, 吴常玉, 曹孟娟. ARM Cortex-M3 与 Cortex-M4 权威指南[M]. 1. 清华大学出版社, 2015.
- [4] STM32 Cortex®-M4 MCUs and MPUs programming manual[M]. 10. STMicroelectronics, 2020.

致谢

本报告的工作是在我们的指导教师廖勇老师的悉心指导下完成的。导师渊博的专业知识，严谨的治学精神，诲人不倦的高尚师德都让我们印象深刻。从课题的选择到项目的最终完成，廖勇老师都始终给予了我们悉心的指导，是他带领我们把大任务分解成小目标，让我们有了明确的进度安排，明白了每一个小步骤具体应该做些什么，当我们表示看不懂某个小目标时，老师马上会做出详细的解释，帮助我们搞明白具体应做些什么，引导我们一步步把最初看似不可能完成的任务变成了可能！感谢廖老师带领我们一行一行地分析核心代码，让我们的能力和对系统、底层的理解都有了很大提升，体会到了操作系统课程所讲的一个个概念如何用代码实现！

感谢每一位做过知识分享的同学！是每一堂课上的每一位讲解知识和上台交流的同学，让我们领略到了不同人具有创造力的独特思想，在讲解与交流的过程中，我们通过学习他人的想法，开阔了自身眼界，让自身能力得到了提高。

感谢同组成员之间合作过程中的齐心协力，在百忙的课程学习中，组员之间依然坚持抽出时间共同交流合作，共同完成了移植 $\mu\text{C}/\text{OS-II}$ 到 stm32 芯片上，并基于 $\mu\text{C}/\text{OS-II}$ 运行四轴飞行器的各个功能模块的艰巨任务！