摘 要

本文是基于STM32嵌入式开发平台，对四轴飞行器控制系统底层运行平台搭建及设计的描述。主要意义是为了合理运用理论和专业知识解决工程设计中的问题，掌握工程设计的基本方法和步骤，为培养优秀的工程师奠定基础。本文首先介绍了嵌入式操作系统的移植，在上学期的基础上将μC/OS-Ⅱ操作系统移植到STM32开发板上，接着在μC/OS-Ⅱ操作系统的基础上将上学期完成的代码进行移植，增加了串口驱动，并通过GY86中的MPU6050模块完成了初步的飞行状态获取和姿态解算。

**关键词：四轴飞行器，μC/OS-Ⅱ，MPU6050，I2C**

# 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 4

1.3. 针对复杂工程问题的方案实现 7

第二章 系统测试 27

2.1 uC/OS-II操作系统移植工程的建立与测试 27

2.2 MPU6050驱动程序的测试 32

第三章 知识技能学习情况 34

第四章 分工协作与交流情况 35

参考文献 36

致谢 37

# 第一章 针对复杂工程问题的方案设计与实现

* 1. 针对复杂工程问题的方案设计
     1. μC/OS-II操作系统移植的设计

此次要移植的操作系统选择的是μC/OS-II。因为μC/OS-II作为源代码公开的实时内核，能满足目前嵌入式应用的绝大部分要求，开放源代码的特性更是使得整个系统更加透明，减少了系统设计的隐患，加上μC/OS-II系统的可裁剪性，使它可以轻松的嵌入到很小的系统之中，大大的增加了系统的灵活性。使嵌入式系统更易开发、管理和维护，从而大大减少各项成本。在现实中具有重要的意义。为了移植μC/OS-II，首先要了解的是一般嵌入式框架，才能对移植操作系统的工程结构进行详细设计。一般的嵌入式框架如图1-1。



图1-1

可以看到，嵌入式操作系统是介于用户程序和硬件直接的一个系统软件，操作系统对用户隐藏了硬件细节，用户只需要调用操作系统提供的接口便能对各种硬件资源进行访问。以此可设计出移植μC/OS-II操作系统的工程结构如图1-2。



图1-2

ucosii\_source和ucosii\_port是官网所提供的μC/OS-II源码。其中ucosii\_source是与ARM处理器无关的代码，包括任务、信号量、内存管理、队列等功能，ucosii\_port则是与ARM处理器直接相关的代码，也是在移植过程中需要重点修改的地方，在这一部分，主要是一些和处理器相关的函数或者宏定义。接着就是BSP(Board Support Pack)部分，其中包含两个重要部分，一个是BSP.c，里面是我们自己编写的各种驱动初始化的地方，另一个则是启动文件stm32f401xx.s，里面包含了异常向量表的建立、栈堆的定义、系统时钟设定以及由汇编跳转到C语言主函数的代码。然后是用户应用程序部分，这里包含用户自己实现的各种驱动程序以及主函数。图1-1最下层则是我们所使用的开发板STM32F401RE。可以看到，整个工程结构清晰明了，从上电一刻开始经历的流程如图1-3。



图1-3

当给开发板上电之后便进入了启动代码stm32f401xx.s，进行了异常向量表的建立、栈堆的定义、系统时钟设定等操作，然后跳转到主函数main.c，紧接着就是调用BSP.c对各种驱动进行初始化，之后返回主函数对操作系统进行初始化，随后便开始创建多个任务，最后开启操作系统开始执行任务。

* + 1. MPU6050驱动程序的设计

MPU6050 内部整合了 3 轴陀螺仪和 3 轴加速度传感器，并且含有一个第二 IIC 接口 ，可用于连接外部磁力传感器，并利用自带的数字运动处理器（DMP: Digital Motion Processor）硬件加速引擎，通过主 IIC 接口，向应用端输出完整的 9 轴融合演算数据。

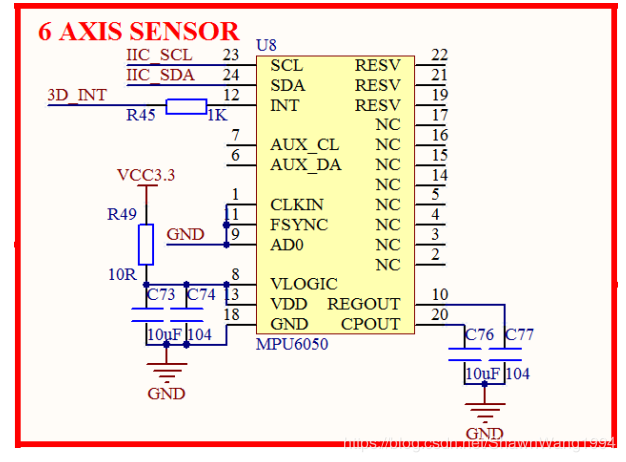


图1-4 MPU6050原理图

**硬件设计**：

循环获取MPU的原始数据，通过DMP的姿态解算得到欧拉角，再通过MPU外接IIC模块，经过SMTM32F401核心板串口输出显示到显示屏上。

所需硬件资源：

1. STM32F401RET6核心板
2. MPU6050
3. IIC模块
4. 杜邦线
5. 计算机
6. 四轴模型

**软件设计**：

编写对应的iic.c、iic.h、mpu6050.c和mpu6050.h。

* 1. 针对复杂工程问题的推理分析
     1. μC/OS-II操作系统的分析

在移植μC/OS-II前，需要对μC/OS-II的代码特点进行分析。μC/OS-II源码结构如表1-1。

|  |  |
| --- | --- |
| ucosii\_source | os\_core.c |
| os\_flag.c |
| os\_mbox.c |
| os\_mem.c |
| os\_mutex.c |
| os\_q.c |
| os\_sem.c |
| os\_task.c |
| os\_time.c |
| os\_tmr.c |
| ucosii.h |
| ucosii\_port | os\_cpu.h |
| os\_cpu\_a.s |
| os\_cpu\_c.c |
| os\_dbg.c |

表1-1

ucosii\_source是与处理器及其他硬件都无关的代码。可以看到，这些代码占了整个μC/OS-II 的绝大部分。作为嵌入式操作系统，易于移植是一个优秀操作系统必不可少的特性之一。为了使μC/OS-II易于移植，它的创始人花费了大量的心血，力求与硬件相关的代码部分占整个系统内核的比例降到最小。ucosii\_port部分，os\_cpu.h主要是定义了一些数据类型、处理器相关代码以及声明函数原型，函数都是需要在os\_cpu\_a.s中用汇编实现，os\_cpu\_c.c主要是一些钩子函数的定义以及任务堆栈初始化函数的定义，而os\_dbg.c是和调试相关的代码，这部分暂时不用去管。

值得一提的是，Cortex-M3、Cortex-M4处理器和一般的ARM9处理器（如三星的S3C2440A）相比，有一个很明显的区别就是上下文切换方式不一样。Cortex-M3、Cortex-M4处理器的上下文切换是发生在PendSV中断的最后，并且PendSV的可编程中断优先级设为最低。这样做是为了防止中断处理杯耽搁的情况。

假如系统使用SysTick异常进行任务切换，则正常情况下如图1-4：

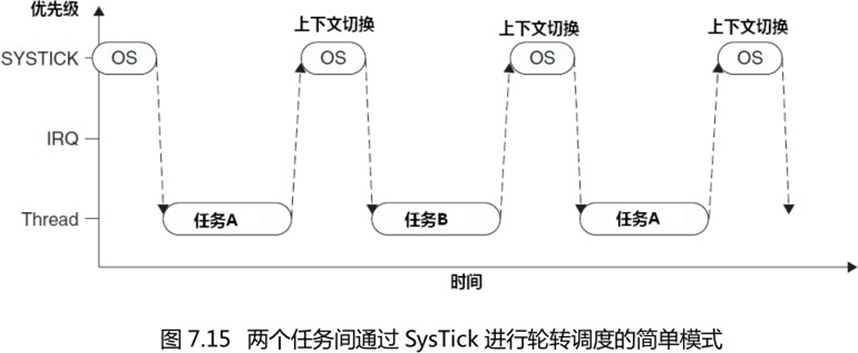


图1-5

但实际上，有时候单片机会进入中断状态响应其他中断，这时如果再产生Sys tick中断，进行任务切换，打断了原来的中断服务，则运行流程如图1-5：

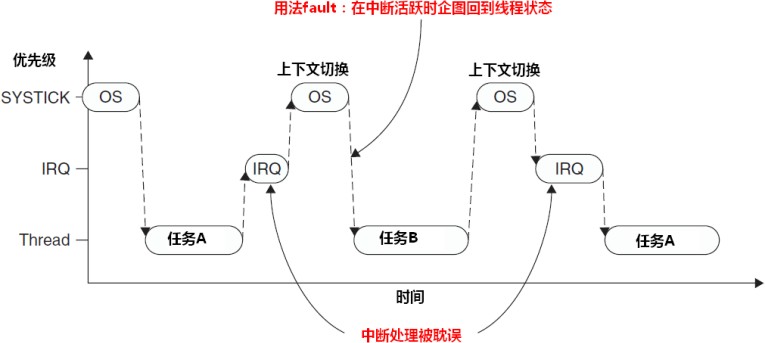


图1-6

显然，中断服务被打断了，间距的时间比较长，这是实时系统所无法忍受的。为此，引入了PendSV来完美解决这个问题，如图1-6：

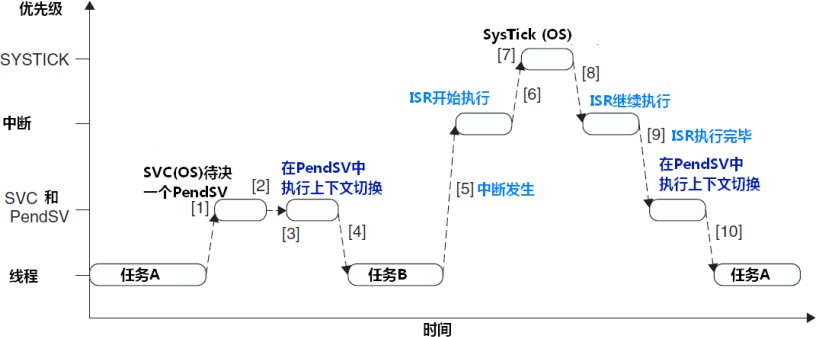


图1-7

PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。

* 1. 针对复杂工程问题的方案实现
     1. μC/OS-II操作系统移植的实现

μC/OS-II操作系统的移植主要体现在os\_cpu\_a.s汇编文件的编写以及os\_cpu.h、os\_cpu\_c.c文件的修改上。

(1) 内核头文件(os\_cpu.h)

在os\_cpu.h中，主要声明了一些与微处理器相关的常量、宏和typedef。

|  |
| --- |
| *typedef unsigned char BOOLEAN;*  *typedef unsigned char INT8U; /\* Unsigned 8 bit quantity\*/*  *typedef signed char INT8S; /\* Signed 8 bit quantity\*/*  *typedef unsigned short INT16U; /\* Unsigned 16 bit quantity\*/*  *typedef signed short INT16S; /\* Signed 16 bit quantity\*/*  *typedef unsigned int INT32U; /\* Unsigned 32 bit quantity\*/*  *typedef signed int INT32S; /\* Signed 32 bit quantity\*/*  *typedef float FP32; /\* Single precision floating point\*/*  *typedef double FP64; /\* Double precision floating point\*/*  *typedef unsigned int OS\_STK; /\* Each stack entry is 32-bit wide\*/*  *typedef unsigned int OS\_CPU\_SR; /\* Define size of CPU status register (PSR = 32 bits)\*/* |

代码1-1

在 STM32 处理器及 keil MDK 或者 IAR 编译环境中可以通过查手册得知 short 类型是 16 位而 int 类型是 32 位，这对于 Cortex-M4 内核是一致的。故这部分代码无需修改。尽管μC/OS-II 定义了 float 类型和 double 类型，但为了方便移植它们在μC/OS-II 源代码中并未使用。为了方便使用堆栈，μC/OS-II 定义了一个堆栈数据类型。在 Cortex-M4 中寄存器为 32位，故定义堆栈的长度也为 32 位。Cortex-M4 状态寄存器为 32 位，定义OS\_CPU\_SR 主要是为了在进出临界代码段保存状态寄存器。

μC/OS-II 为了保证某段代码的完整执行，需要临时的关闭中断，在这段代码执行完成之后再打开中断。这样的代码段称作临界代码段。μC/OS-II 通过定义两个宏OS\_ENTER\_CRITICAL()和OS\_EXIT\_CRITICAL()来分别实现中断的关闭和打开。一般来说，采用方法 3 来实现这两个宏。这两个宏分别定义如代码1-2。

|  |
| --- |
| *#define OS\_CRITICAL\_METHOD 3u*  *#if OS\_CRITICAL\_METHOD == 3u*  *#define OS\_ENTER\_CRITICAL() {cpu\_sr = OS\_CPU\_SR\_Save();}*  *#define OS\_EXIT\_CRITICAL() {OS\_CPU\_SR\_Restore(cpu\_sr);}*  *#endif* |

代码1-2

函数OS\_CPU\_SR\_Save()和OS\_CPU\_SR\_Restore(cpu\_sr) 在os\_cpu\_a.s 中定义。同时得注意，在使用这两个宏之前，必须定义OS\_CPU\_SR cpu\_sr;否则编译时将出错。

尽管μC/OS-II 支持两种方向生长的栈，但对于以 Cortex-M3 为内核的STM32 微处理器来说，它支持向下增长的满栈，故需要定义栈增长方向宏为1。如代码1-3。

|  |
| --- |
| *#define OS\_STK\_GROWTH 1* |

代码1-3

任务级任务切换调用宏 OS\_TASK\_SW()来实现。因为这个宏也是与处理器相关的，因此这个宏在 OS\_CPU\_A.ASM 中描述。在 os\_cpu.h 中，还声明了代码1-4的几个函数，这几个函数均在 os\_cpu\_a.s 中实现。

|  |
| --- |
| *void OSCtxSw (void);*  *void OSIntCtxSw (void);*  *void OSStartHighRdy (void);*  *void OS\_CPU\_PendSVHandler (void);* |

代码1-4

(2) 处理器相关汇编文件(os\_cpu\_a.s)

在 os\_cpu\_a.s 中实现的是下面代码1-5五个与处理器相关的函数。

|  |
| --- |
| *OS\_CPU\_SR\_Save(); L(1)*  *OS\_CPU\_SR\_Restore();*  *OSStartHighRdy();*  *OSCtxSw();*  *OSIntCtxSw(); L(4)*  *OS\_CPU\_PendSVHandler(); L(5)* |

代码1-5

其中代码1-5 L(1)~L(4)是声明在os\_cpu.h中的函数，这里不再赘述。之前提到过Cortex-M4处理器和一般ARM处理器不同的是对上下文切换的实现，Cortex-M4处理器上下文切换统一放在PendSV中断的最后来实现，因此还需要添加代码1-5 L(5)处的OS\_CPU\_PendSVHandler()函数，用以实现上下文切换的功能。

a) 关中断函数(OS\_CPU\_SR\_Save)

OS\_ENTER\_CRITICAL()里进入临界区调用，保存现场。其实现如代码1-6。

|  |
| --- |
| *OS\_CPU\_SR\_Save*  *MRS R0,PRIMASK*  *CPSID I*  *BX LR* |

代码1-6

实现原理为先读取PRIMASK到R0（保存全局中断标记，除了故障中断），然后令PRIMASK=1，关中断，最后返回并且返回值保存在R0。

b) 恢复中断函数(OS\_CPU\_SR\_Restore)

OS\_EXIT\_CRITICAL()里退出临界区调用，恢复现场环境。其实现如代码1-7。

|  |
| --- |
| *OS\_CPU\_SR\_Restore*  *MSR PRIMASK,R0*  *BX LR* |

代码1-7

实现原理为读取R0到PRIMASK中(恢复全局中断标志)，通过R0传递参数。

Cortex-M4处理器有单独的指令来打开或者关闭中断，所以这两个函数实现起来很简单。

c) 启动最高优先级任务运行(OSStartHighRdy)

OSStart()调用OSStartHighRdy()来启动最高优先级任务的运行，从而启动整个系统。OSStartHighRdy()主要完成以下几项工作：

① 为任务切换设置PendSV的优先级

② 为第一次任务切换设置栈指针为0，

③ 设置OSRunning = TRUE,以表明系统正在运行

④ 触发一次PendSV,打开中断等待第一次任务的切换。

其实现如代码1-8。

|  |
| --- |
| *OSStartHighRdy*  *;set PendSV prio*  *LDR R0,=NVIC\_SYSPRI14*  *LDR R1,=NVIC\_PENDSV\_PRI*  *STR R1,[R0]*    *;clear PSP*  *MOV R0,#0*  *MSR PSP,R0*    *;OSRunning = true*  *LDR R0,=OSRunning*  *MOV R1,#1*  *STR R1,[R0]*    *;pend(enable) PendSV interrupt*  *LDR R0,=NVIC\_INT\_CTRL*  *LDR R1,=NVIC\_PENDSVSET*  *STR R1,[R0]*    *;open interrupt*  *CPSIE I* |

代码1-8

d) 上下文切换(OSCtxSw)

当任务放弃 CPU 的使用权时，就会调用 OS\_TASK\_SW()。一般情况下，OS\_TASK\_SW()是做任务切换。但在Cortex-M4中，任务切换的工作都被放到 PendSV 的中断处理服务中去做以加快处理速度，因此OS\_TASK\_SW()只需简单的悬起(允 许)PendSV 中断即可。当然，这样就只有当再次开中断的时候，PendSV中断处理函数才能执行。其实现如代码1-9。

|  |
| --- |
| *OSCtxSw ;only need to enable PendSV interrupt*  *LDR R0,=NVIC\_INT\_CTRL*  *LDR R1,=NVIC\_PENDSVSET*  *STR R1,[R0]*    *BX LR* |

代码1-9

e) 中断退出处理时的上下文切换(OSIntCtxSw)

当中断处理函数退出时，就会调用OSIntExit()来决定是否有优先级更高的任务需要执行。如果有，OSIntExit()会调用OSIntCtxSw()做任务切换。在Cortex-M4里，与OSCtxSw一样，任务切换时，OSIntCtxSw都只需简单的悬起(允许)PendSV 中断即可，真正的任务切换工作放在 PendSV 中断服务程序里，等待开中断时才正在执行任务切换。在这里，OSIntCtxSw的代码是与OSCtxSw完全相同的，其实现如代码1-10。

|  |
| --- |
| *OSIntCtxSw ;the same with OSCtxSw*  *LDR R0,=NVIC\_INT\_CTRL*  *LDR R1,=NVIC\_PENDSVSET*  *STR R1,[R0]*    *BX LR* |

代码1-10

f) PendSV中断服务(OS\_CPU\_PendSVHandler)

PendSV的作用在1.2.1节中已经详细说明了，这里不再赘述。为方便理解，首先来看其伪代码，如代码1-11处。

|  |
| --- |
| *//OS\_CPU\_PendSVHandler 伪代码思路*  *OS\_CPU\_PendSVHandler:*  *if (PSP != NULL) { //当调用 OS\_CPU\_PendSVHandler() 时，*  *//CPU 就会自动保存 xPSR、PC、LR、R12、R0-R3 寄存器到堆栈*  *//保存后，CUP 的栈 SP 指针会切换到使用主堆栈指针 MSP 上*  *//我们只需检测 进入栈指针 PSP 是否为 NULL*  *//就知道是否进行任务切换*  *//因此当我们第一次启动任务是，OSStartHighRdy()*  *//就把 PSP 设为 NULL，避免系统以为已经进行任务切换*  *Save R4-R11 onto task stack; //手动保存 R4-R11*  *OSTCBCur->OSTCBStkPtr = SP; //保存进入栈指针 PSP 到任务控制块*  *//以便下次继续任务运行时继续使用原来的栈*  *}*  *OSTaskSwHook(); //此处便于我们使用钩子函数来拓展功能*  *OSPrioCur = OSPrioHighRdy; //获取最高优先级就绪任务的优先级*  *OSTCBCur = OSTCBHighRdy; //获取最高优先级就绪任务的任务控制块指针*  *PSP = OSTCBHighRdy->OSTCBStkPtr; //保存进入栈指针*  *Restore R4-R11 from new task stack; //从新的栈恢复 R4-R11 寄存器*  *Return from exception; //返回* |

代码1-11

具体汇编代码1-12如下：

|  |
| --- |
| *OS\_CPU\_PendSVHandler ;CPU 会自动保存 xPSR, PC, LR, R12, R0-R3 CPSID I ;关中断 MRS R0, PSP ;PSP 就是栈指针， R0=PSP CBZ R0, OSPendSV\_nosave ;当 PSP==0，执行 OSPendSV\_nosave 函数  SUB R0, R0, #0x20 ;装载 r4-11 到栈 ，共 8 个寄存器，  ;32 位， 4 个字节即 8\*4=32=0x20 STM R0, {R4-R11}   LDR R1, \_\_OS\_TCBCur ;R1=&OSTCBCur LDR R1, [R1] ;R1=\*R1 (R1=OSTCBCur) STR R0, [R1] ;\*R1=R0 (\*OSTCBCur=SP)*  *OSPendSV\_nosave PUSH {R14} ;保存 R14 LDR R0, \_\_OS\_TaskSwHook ;调用钩子函数 OSTaskSwHook() BLX R0 POP {R14} ;恢复 R14  LDR R0, \_\_OS\_PrioCur ;设置当前优先级为最高优先级就绪任务的优先级  ;OSPrioCur = OSPrioHighRdy LDR R1, \_\_OS\_PrioHighRdy LDRB R2, [R1] STRB R2, [R0]  LDR R0, \_\_OS\_TCBCur ;设置当前任务控制块指针 LDR R1, \_\_OS\_TCBHighRdy ;OSTCBCur = OSTCBHighRdy LDR R2, [R1] STR R2, [R0]  LDR R0, [R2] ;R0 是新的 SP  ;SP = OSTCBHighRdy->OSTCBStkPtr;  LDM R0, {R4-R11} ;从新的栈恢复 R4-R11 ADD R0, R0, #0x20 MSR PSP, R0 ;PSP=R0,用新的栈 SP 加载 PSP ORR LR, LR, #0x04 ;确保 LR 位 2 为 1，返回到使用进程堆栈 CPSIE I ;开中断 BX LR ;返回* |

代码1-12

当第一次开始任务切换时时， 而任务刚创建时 R4-R11 已经保存在堆栈中，此时不用再保存，就会跳到 OS\_CPU\_PendSVHandler\_nosave 执行。前面已经说过真正的任务切换是在 PendSV 中断处理函数里做的，由于 Cortex-M4 在中断时会有一些的寄存器自动保存到任务堆栈里，所以在 PendSV 中断处理函数中只需保存 R4-R11并调节堆栈指针即可。 其中 xPSR, PC, LR, R12, R0-R3 已自动保存，不用我们管了。

图1-7为一个任务切换时寄存器的情况：

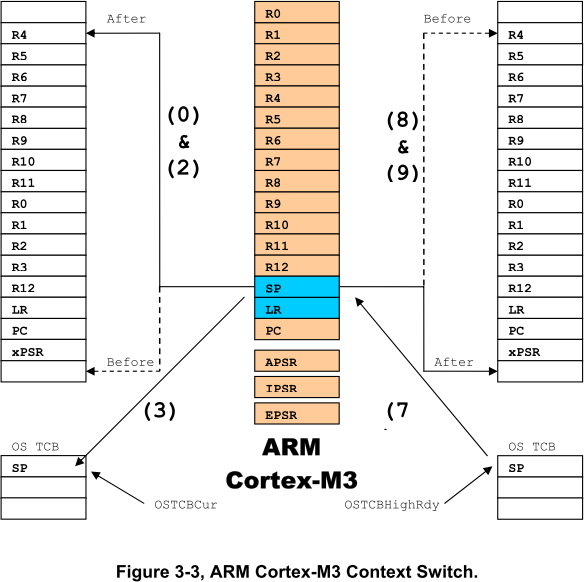


图1-8

(3) 处理器相关C函数和钩子函数(os\_cpu.c)

移植 μC/OS 时，我们需要写 10 个相当简单的 C 函数： 9 个钩子函数和 1 个任务堆栈结构初始化函数。所谓钩子函数，指那些插入到某些函数中为扩展这些函数功能的函数。一般地，钩子函数为第三方软件开发人员提供扩充软件功能的入口点。为了拓展系统功能， μC/OS-II 中提供有大量的钩子函数，用户不需要修改 uC/OS-II 内核代码程序，而只需要向钩子函数添加代码就可以扩充 μC/OS-II 的功能。

a) 钩子函数

尽管 uC/OS-II 中提供了大量的钩子函数，但实际上，移植时我们需要编写的也就 9 个钩子函数，如代码1-13。

|  |
| --- |
| *OSInitHookBegin() //OSIinit() 系统初始化函数开头的钩子函数 OSInitHookEnd() //OSIinit() 系统初始化函数结尾的钩子函数 OSTaskCreateHook() //OSTaskCreate()或 OSTaskCreateExt() 创建任务钩子函数 OSTaskDelHook() //OSTaskDel() 删除任务钩子函数 OSTaskIdleHook() //OS\_TaskIdle() 空闲任务钩子函数 OSTaskStatHook() //OSTaskStat() 统计任务钩子函数 OSTaskSwHook() //OSTaskSW() 任务切换钩子函数 OSTCBInitHook() //OS\_TCBInit() 任务控制块初始化钩子函数 OSTimeTickHook() //OSTaskTick() 时钟节拍钩子函数* |

代码1-13

然而在本次移植操作系统中不需要用到这些钩子函数，所以暂时只把他们定义成空函数体。

b) 任务堆栈结构初始化函数(OSTaskStkInit())

函数定义如代码1-14：

|  |
| --- |
| *OS\_STK \*OSTaskStkInit (void (\*task)(void \*p\_arg), void \*p\_arg, OS\_STK \*ptos, INT16U opt){*  *OS\_STK \*p\_stk;*  *(void)opt; /\* 'opt' is not used, prevent warning\*/*  *p\_stk = ptos + 1u; /\* Load stack pointer\*/*  */\* Align the stack to 8-bytes.\*/*  *p\_stk = (OS\_STK \*)((OS\_STK)(p\_stk) & 0xFFFFFFF8u);*    */\* Registers stacked as if auto-saved on exception\*/*  *\*(--p\_stk) = (OS\_STK)0x01000000uL; /\* xPSR\*/*  *\*(--p\_stk) = (OS\_STK)task; /\* Entry Point\*/*  *\*(--p\_stk) = (OS\_STK)OS\_TaskReturn; /\* R14 (LR)\*/*  *\*(--p\_stk) = (OS\_STK)0x12121212uL; /\* R12\*/*  *\*(--p\_stk) = (OS\_STK)0x03030303uL; /\* R3\*/*  *\*(--p\_stk) = (OS\_STK)0x02020202uL; /\* R2\*/*  *\*(--p\_stk) = (OS\_STK)0x01010101uL; /\* R1\*/*  *\*(--p\_stk) = (OS\_STK)p\_arg; /\* R0 : argument\*/*  */\* Remaining registers saved on process stack\*/*  *\*(--p\_stk) = (OS\_STK)0x11111111uL; /\* R11\*/*  *\*(--p\_stk) = (OS\_STK)0x10101010uL; /\* R10\*/*  *\*(--p\_stk) = (OS\_STK)0x09090909uL; /\* R9\*/*  *\*(--p\_stk) = (OS\_STK)0x08080808uL; /\* R8\*/*  *\*(--p\_stk) = (OS\_STK)0x07070707uL; /\* R7\*/*  *\*(--p\_stk) = (OS\_STK)0x06060606uL; /\* R6\*/*  *\*(--p\_stk) = (OS\_STK)0x05050505uL; /\* R5\*/*  *\*(--p\_stk) = (OS\_STK)0x04040404uL; /\* R4\*/*  *return (p\_stk);*  *}* |

代码1-14

这是初始化任务堆栈函数。 OSTaskStkInit()被任务创建函数调用，所以要在开始时，在栈中作出该任务好像刚被中断一样的假象。在 ARM 内核中，函数中断后， xPSR， PC， LR， R12， R3-R0 被自动保存到栈中的， R11-R4如果需要保存，只能手工保存。 为了模拟被中断后的假象， OSTaskStkInit()的工作就是在任务自己的栈中保存 cpu 的所有寄存器。这些值里 R1-R12 都没什么意义，这里用相应的数字代号（如 R1 0x01010101）主要是方便调试。

完成以上对os\_cpu.h、os\_cpu\_a.s、os\_cpu\_c.c文件的编写与改，操作系统移植的工作就完成了。

* + 1. I2C协议的实现

**I2C简介：**

I2C总线是由Philips公司开发的一种简单、双向二线制同步串行总线。它只需要两根线即可在连接于总线上的器件之间传送信息。

主器件用于启动总线传送数据，并产生时钟以开放传送的器件，此时任何被寻址的器件均被认为是从器件．在总线上主和从、发和收的关系不是恒定的，而取决于此时数据传送方向。如果主机要发送数据给从器件，则主机首先寻址从器件，然后主动发送数据至从器件，最后由主机终止数据传送；如果主机要接收从器件的数据，首先由主器件寻址从器件．然后主机接收从器件发送的数据，最后由主机终止接收过程。在这种情况下．主机负责产生定时时钟和终止数据传送。

**I2C工作原理：**

I2C主要使用两条总线进行通信，一条是SDA即数据总线，另一条是SCL即控制总线，这两条总线都是双向I/O总线，开漏输出，在总线空闲时，电平为高。

在主设备启动总线传输数据时，需要向SDA总线传送要寻址的从设备地址以及设备对应的寄存器地址，地址一般为7位，第8位用于表示读或者写信号，因此，I2C上的数据总是以字节格式传输，其中，最先传输的是字节的最高位。

****

图1-9 I2C数据传输格式

**I2C时序：**

在I2C的使用中，我们可以通过不同的时序完成不同的操作。

**I2C起始时序：**

我们在保持SCL总线为高电平的情况下，拉低SDA总线，即可产生I2C的起始信号。

**I2C终止时序：**

如果在SCL保持高电平时，拉高SDA总线即可终止I2C的数据传输。

**I2C数据传输时序：**

在数据传输过程中，必须保持SCL为高电平才能传输SDA总线上的数据，如果要改变SDA的电平信号，则必须将SCL拉低。

**I2C响应时序：**

在数据传输完成后，接收方需要向发送方发送一个响应信号，表示已经接收到或者没有接受到数据。响应时，SCL线产生一个时钟脉冲，此时接收方如果将SDA拉低，表示接受到数据，即ACK，如果将SDA拉高，则表示NACK。

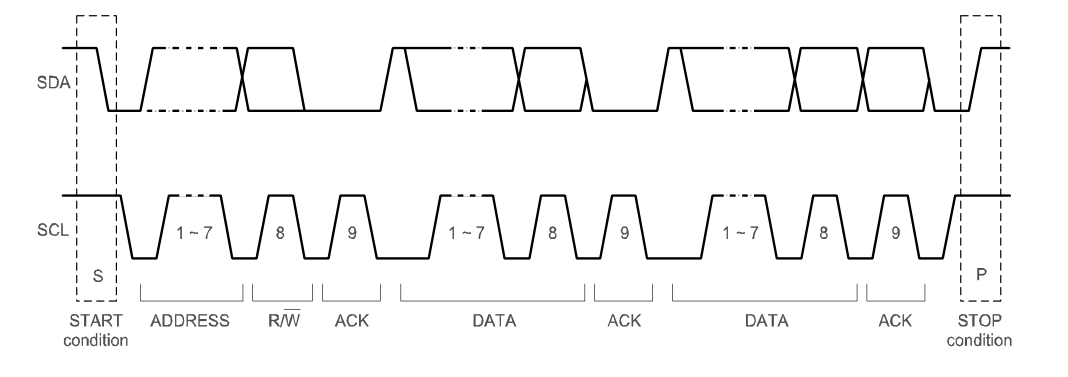


图1-10 I2C时序图

**代码分析（软件模拟）：**

i2c\_gpio\_config()

|  |
| --- |
| *void i2c\_gpio\_config(){*  *GPIO\_InitTypeDef GPIO\_InitStructure;*    *RCC\_AHB1PeriphClockCmd(RCC\_AHB1Periph\_GPIOB,ENABLE);*    *GPIO\_InitStructure.GPIO\_Pin = I2C\_SDA\_GPIO\_PIN;*    *//设置为开漏输出模式*  *GPIO\_InitStructure.GPIO\_Mode = GPIO\_Mode\_OUT;*  *GPIO\_InitStructure.GPIO\_OType = GPIO\_OType\_OD;*    *GPIO\_InitStructure.GPIO\_Speed = GPIO\_Speed\_50MHz;*    *//初始化SDA*  *GPIO\_Init(GPIOC,&GPIO\_InitStructure);*    *//初始化SCL*  *GPIO\_InitStructure.GPIO\_Pin = I2C\_SCL\_GPIO\_PIN;*  *GPIO\_Init(GPIOC,&GPIO\_InitStructure);*  *}* |

代码1-15

i2c\_Start() & i2c\_Stop()

|  |
| --- |
| *void i2c\_Start(){*  *I2C\_SDA\_1();*  *I2C\_SCL\_1();*    *delay\_ms(4);*    *I2C\_SDA\_0();*  *delay\_ms(4);*  *I2C\_SCL\_0();*  *}*  *void i2c\_Stop(){*  *I2C\_SDA\_0();*  *I2C\_SCL\_1();*  *delay\_ms(4);*  *I2C\_SDA\_1();*  *}* |

代码1-16

i2c\_Ack() & i2c\_NAck()

|  |
| --- |
| */\*\* 产生应答信号*  *\* 在SCL总线为高电平时*  *\* 将SDA置低表示产生应答信号*  *\*/*  *void i2c\_Ack(){*  *I2C\_SDA\_0();*  *delay\_ms(4);*  *I2C\_SCL\_1();*  *delay\_ms(4);*  *I2C\_SCL\_0();*  *delay\_ms(4);*  *I2C\_SDA\_1();*  *}*  */\*\* 产生非应答信号*  *\* 在SCL总线为高电平时*  *\* 将SDA置高表示产生应答信号*  *\*/*  *void i2c\_NAck(){*  *I2C\_SDA\_1();*  *delay\_ms(4);*  *I2C\_SCL\_1();*  *delay\_ms(4);*  *I2C\_SCL\_0();*  *}* |

代码1-17

i2c\_SendByte()

|  |
| --- |
| *void i2c\_SendByte(uint8\_t byte){*  *uint8\_t i;*  *for(i = 0;i<8;i++){*  */\* 确保SCL总线处于低电平 \*/*  *I2C\_SCL\_0();*    */\* 如果数据最高位为1，则发送高电平信号，否则为0，发送低电平信号 \*/*  *if(byte&0x80){*  *I2C\_SDA\_1();*  *}else I2C\_SDA\_0();*    */\* 在此处是否需要DELAY一下，存疑*  *\*\* 似乎有必要，在此处DELAY，可以确保数据准备完成，再产生时钟信号，确保数据发送的稳定*  *\*/*  *delay\_ms(4);*    */\* 传输完最高位（最左边）后，左移一位，传输下一位数据 \*/*  *byte<<=1;*    */\* 产生一个上升沿，表示开始接受数据 \*/*  *I2C\_SCL\_1();*  *delay\_ms(4);*  *I2C\_SCL\_0();*    */\* 当一个字节数据发送完成后*  *\*\* 需要将SDA总线释放（拉高），因为存在最后一位是0的情况*  *\*\* 这之后主机可以产生一个重复起始条件或停止信号*  *\*/*  *if(i==7){*  *I2C\_SDA\_1();*  *}*  *}*  *}* |

代码1-18

i2c\_ReceiveByte()

|  |
| --- |
| *uint8\_t i2c\_ReceiveByte(void){*  *uint8\_t i;*  *uint8\_t ReceiveByte = 0;*    *I2C\_SCL\_0();*  *delay\_ms(4);*  *I2C\_SDA\_1();*  *for(i = 0;i<8;i++){*  */\* SCL上升沿读取数据 \*/*  *I2C\_SCL\_1();*  *delay\_ms(4);*  *ReceiveByte <<= 1;*  *if(I2C\_SDA\_READ()){*  *ReceiveByte |= 0X01;*  *}*  */\* 将SCL拉低，使发送端将数据放在SDA总线上 \*/*  *I2C\_SCL\_0();*  *delay\_ms(4);*  *}*  *return ReceiveByte;*  *}* |

代码1-19

* + 1. 通过I2C读取MPU6050数据

**MPU6050初始化**：

1. 初始化IIC接口
2. 复位MPU6050，让MPU6050内部的所有寄存器恢复默认值
3. 设置角速度传感器和加速度传感器的满量程范围
4. 设置其他参数，关闭中断，关闭AUX IIC接口，禁止FIFO，设置陀螺仪采样率和设置数字低通滤波器等
5. 配置系统时钟源并使能角速度传感器和加速度传感器

**涉及MPU寄存器**：

**电源管理寄存器1**

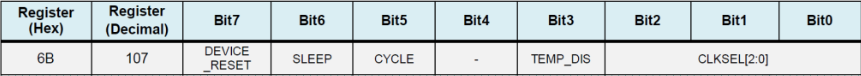


图1-11 电源管理寄存器1

DEVICE\_RESE=1，复位MPU6050，复位完成后，自动清零

SLEEP=1，进入睡眠模式，SLEEP=1，正常工作模式

TEMP\_DIS用于设置是否使能温度传感器，为0则使能

CLKSEL[2:0]用于选择系统时钟源



陀螺仪配置寄存器

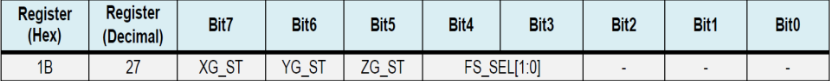


图1-12 陀螺仪配置寄存器

该寄存器中，FS\_SEL[1:0]两个位用于设置陀螺仪满量程范围，一般设置为3

加速度传感器配置寄存器

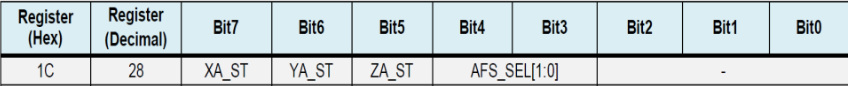


图1-13 加速度传感器配置寄存器

该寄存器中，AFS\_SEL[1:0]两个位用于设置加速度传感器满量程范围，一般设置为0

FIFO使能寄存器

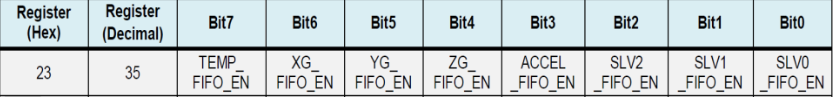


图1-14 FIFO使能寄存器

该寄存器用于控制FIFO使能，在简单读取传感器数据的时候，可以不用FIFO，设置对应位为0，即可进制FIFO，设置为1则使能

**陀螺仪采样率分频寄存器**

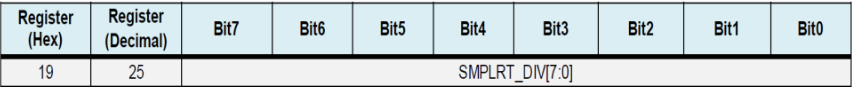


图1-15 陀螺仪采样率分频寄存器

该寄存器用于设置MPU6050的陀螺仪采样频率，计算公式为采样频率=陀螺仪输出频率/（1+SMPLRT\_DIV），陀螺仪的输出频率与数字低通滤波器（DLPF）的设置有关，当DLPF\_CFG=0/7时，频率为8KHz，其他情况是1KHz。且DLPF滤波频率一般设置为采样率的一半。

**配置寄存器**

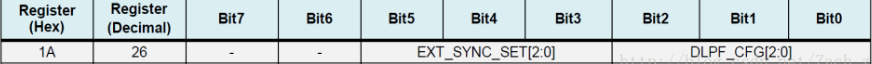


图1-16 配置寄存器

数字低通滤波器（ DLPF）的设置位，即： DLPF\_CFG[2:0]，加速   
度计和陀螺仪，都是根据这三个位的配置进行过滤的。

电源寄存器2

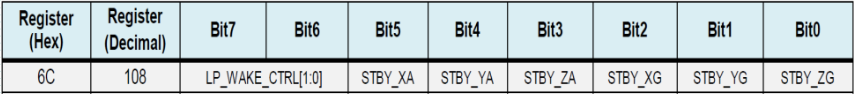


图1-17 电源寄存器2

LP\_WAKE\_CTRL 用于控制低功耗时的唤醒频率剩下的 6 位，分别控制加速度和陀螺仪的x/y/z轴是否进入待机模式。

**陀螺仪数据输出寄存器**

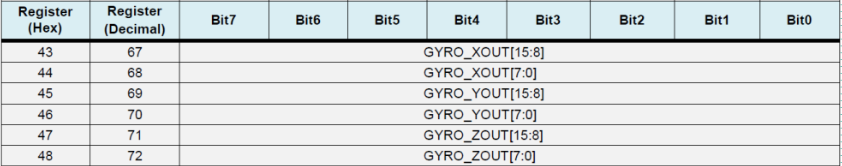


图1-18 陀螺仪数据输出寄存器

通过读取这6个寄存器，就可以读到陀螺仪 x/y/z 轴的值，其中，[15:8]为数据高8位，[7:0]为数据低8位

**加速度传感器数据输出寄存器**

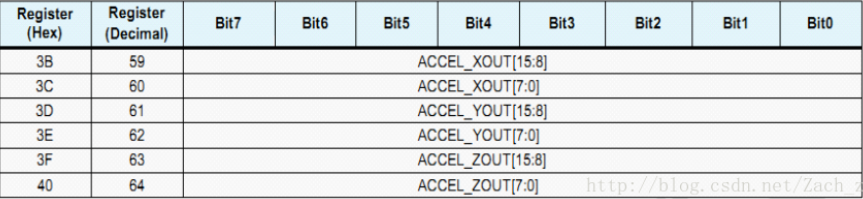


图1-19 加速度传感器数据输出寄存器

通过读取这6个寄存器，就可以读到加速度传感器x/y/z 轴的值，其中，[15:8]为数据高8位，[7:0]为数据低8位

**代码分析**：

MPU6050\_Init()

|  |
| --- |
| *void MPU6050\_Init(void)*  *{*  *I2C\_ByteWrite(MPU6050\_SlaveAddress,PWR\_MGMT\_1,0x00);//解除休眠状态*  *delay\_ms(20);*  *I2C\_ByteWrite(MPU6050\_SlaveAddress,SMPLRT\_DIV,0x07);//发送数据，设定陀螺仪采样率*  *delay\_ms(20);*  *I2C\_ByteWrite(MPU6050\_SlaveAddress,MPU6050\_CONFIG,0x06);//设置低通滤波频率*  *delay\_ms(20);*  *I2C\_ByteWrite(MPU6050\_SlaveAddress,GYRO\_CONFIG,0x08);//陀螺仪自检 +-500度/秒*  *delay\_ms(20);*  *I2C\_ByteWrite(MPU6050\_SlaveAddress,ACCEL\_CONFIG,0x08);//加速度自检、测量范围 +-4G*  *}* |

代码1-20

该函数对MPU6050进行初始化，该函数成功执行之后，即可读取传感器数据。

MPU\_Get\_Gyroscope()

|  |
| --- |
| *u8 MPU\_Get\_Gyroscope(short \*gx,short \*gy,short \*gz)*  *{*  *u8 buf[6],res;*  *res=MPU\_Read\_Len(MPU\_ADDR,MPU\_GYRO\_XOUTH\_REG,6,buf);*  *if(res==0)*  *{*  *\*gx=(((u16)buf[0]<<8)|buf[1])-GYRO\_OFFSET.X;*  *\*gy=(((u16)buf[2]<<8)|buf[3])-GYRO\_OFFSET.Y;*  *\*gz=(((u16)buf[4]<<8)|buf[5])-GYRO\_OFFSET.Z;*  *}*  *return res;;*  *}* |

代码1-21

该函数用于获取陀螺仪值（原始值），其中，gx、gy、gz表示陀螺仪x、y、z轴的原始数据（带符号），返回值为0表示成功，返回值为其他值表示失败。

MPU\_Get\_Accelerometer()

|  |
| --- |
| *u8 MPU\_Get\_Accelerometer(short \*ax,short \*ay,short \*az)*  *{*  *u8 buf[6],res;*  *res=MPU\_Read\_Len(MPU\_ADDR,MPU\_ACCEL\_XOUTH\_REG,6,buf);*  *if(res==0)*  *{*  *\*ax=(((u16)buf[0]<<8)|buf[1])-ACC\_OFFSET.X;*  *\*ay=(((u16)buf[2]<<8)|buf[3])-ACC\_OFFSET.Y;*  *\*az=(((u16)buf[4]<<8)|buf[5])-ACC\_OFFSET.Z;*  *}*  *return res;;*  *}* |

代码1-22

该函数用于获取加速度值（原始值），其中，gx、gy、gz表示陀螺仪x、y、z轴的原始数据（带符号），返回值为0表示成功，返回值为其他值表示失败。

MPU\_write\_Len()

|  |
| --- |
| *u8 MPU\_Write\_Len(u8 addr,u8 reg,u8 len,u8 \*buf)*  *{*  *u8 i;*  *MPU\_IIC\_Start();*  *MPU\_IIC\_Send\_Byte((addr<<1)|0);//发送器件地址+写命令*  *if(MPU\_IIC\_Wait\_Ack())//等待应答*  *{*  *MPU\_IIC\_Stop();*  *return 1;*  *}*  *MPU\_IIC\_Send\_Byte(reg);//写寄存器地址*  *MPU\_IIC\_Wait\_Ack();//等待应答*  *for(i=0;i<len;i++)*  *{*  *MPU\_IIC\_Send\_Byte(buf[i]);//发送数据*  *if(MPU\_IIC\_Wait\_Ack())//等待ACK*  *{*  *MPU\_IIC\_Stop();*  *return 1;*  *}*  *}*  *MPU\_IIC\_Stop();*  *return 0;*  *}* |

代码1-23

MPU\_Write\_Len 用于指定器件和地址，连续写数据，可用于实现 DMP 部分的： i2c\_write函数。

MPU\_Read\_Len()

|  |
| --- |
| *u8 MPU\_Read\_Len(u8 addr,u8 reg,u8 len,u8 \*buf)*  *{*  *MPU\_IIC\_Start();*  *MPU\_IIC\_Send\_Byte((addr<<1)|0);//发送器件地址+写命令*  *if(MPU\_IIC\_Wait\_Ack())//等待应答*  *{*  *MPU\_IIC\_Stop();*  *return 1;*  *}*  *MPU\_IIC\_Send\_Byte(reg);//写寄存器地址*  *MPU\_IIC\_Wait\_Ack();//等待应答*  *MPU\_IIC\_Start();*  *MPU\_IIC\_Send\_Byte((addr<<1)|1);//发送器件地址+读命令*  *MPU\_IIC\_Wait\_Ack();//等待应答*  *while(len)*  *{可以连续读取6个数据而不用写6个寄存器地址*  *if(len==1)\*buf=MPU\_IIC\_Read\_Byte(0);//读数据，发送nACK，直到此时读取完毕*  *else \*buf=MPU\_IIC\_Read\_Byte(1);//读数据，发送ACK，截止读取下一个地址的寄存器中的数据*  *len--;*  *buf++;*  *}*  *MPU\_IIC\_Stop();//产生一个停止条件*  *return 0;*  *}* |

代码1-24

MPU\_Read\_Len 用于指定器件和地址，连续读数据，可用于实现 DMP 部分的： i2c\_read函数。

# 第二章 系统测试

2.1 μC/OS-II操作系统移植工程的建立与测试

下面是在Keil开发环境下建立工程和添加文件并编译测试的具体过程。

建立工程步骤如下：

①新建空白Keil工程文件

②添加μC/OS-II操作系统文件(包含自己修改、编写的文件)

③添加标准库文件

④编写用户程序

⑤编译运行测试

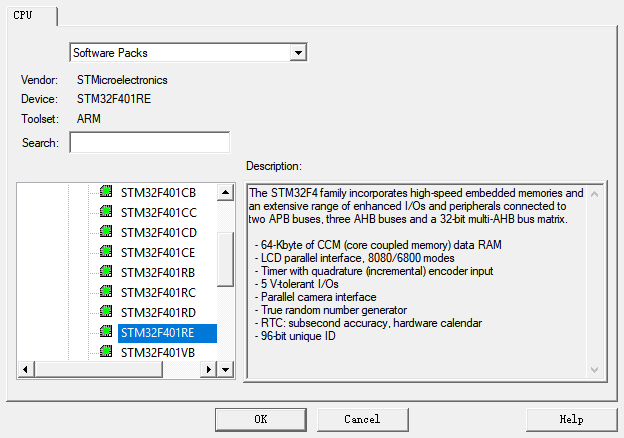
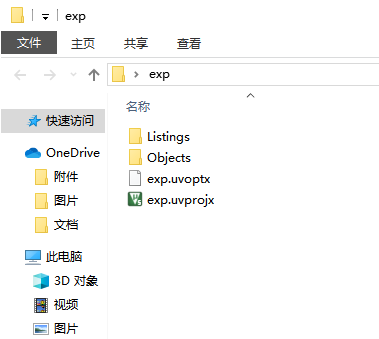
 首先是在一个空文件夹下建立空的Keil工程文件(开发板选择STM32F401RE)，如图2-1：

图2-1

然后再文件夹下建立五个文件夹APP(用户程序)、BSP(板级支持包)、DRIVER(标准库)、UCOS\_PORT(处理器相关μC/OS-II文件)、UCOS\_SOURCE(处理器无关μC/OS-II文件)，如图2-2：

图2-2

接着回到Keil工程，依次将相关文件及文件夹添加到工程里面，如图2-3：

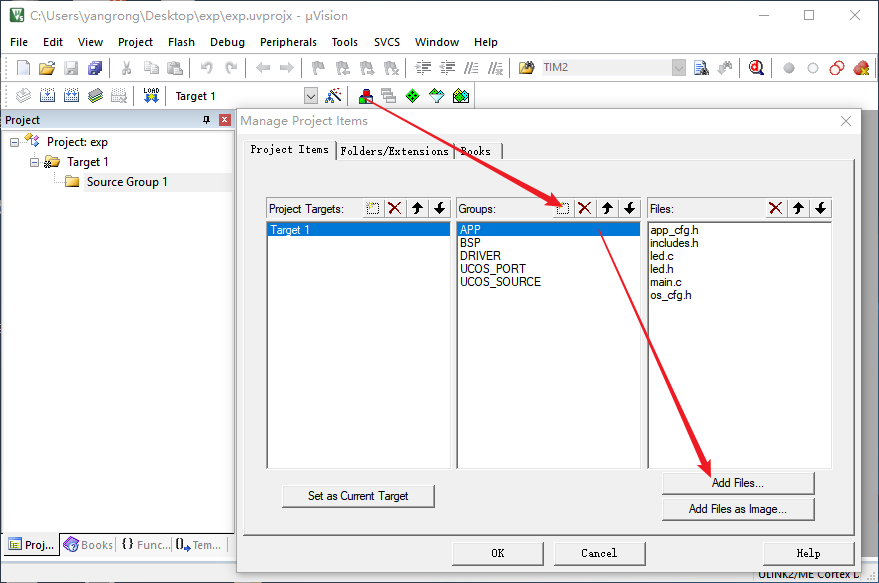
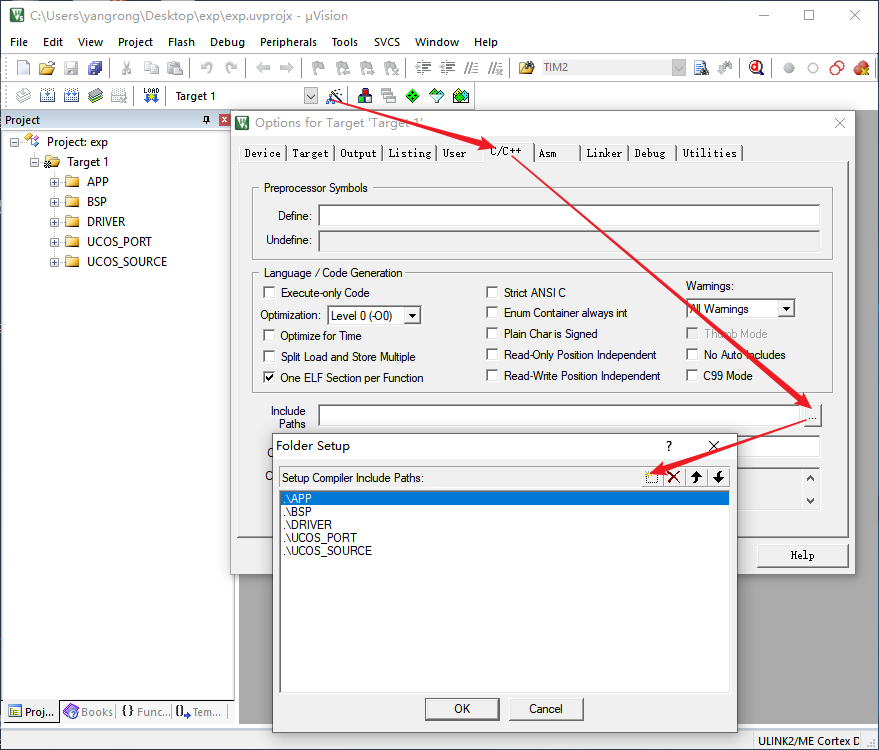


图2-3

接着到Setting->C/C++->Include Paths，把C语言头文件所在的文件夹添加到设置里面，如图2-4：

图2-4

然后转到Setting->C/C++->Preprocessor Symbols添加相关定义，如图2-5：

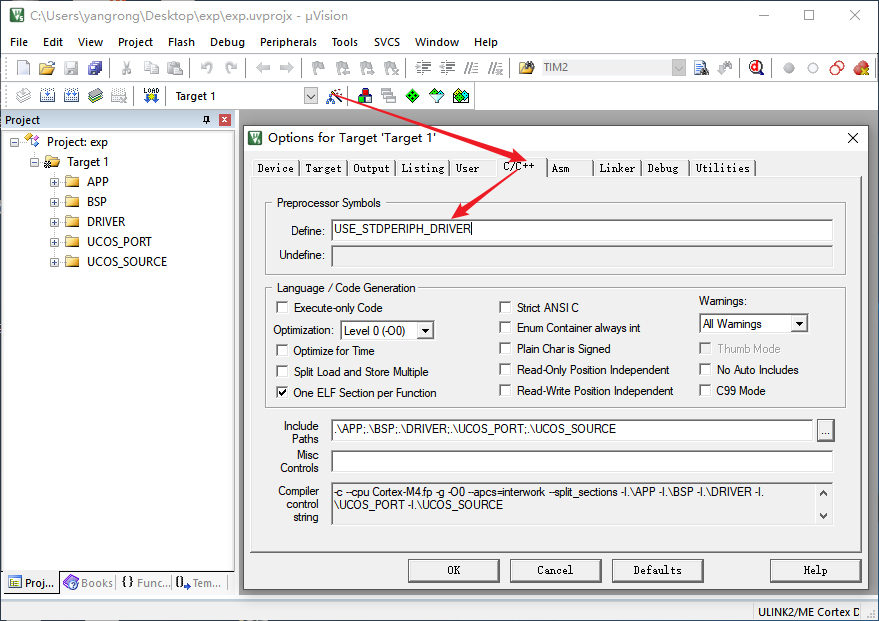


图2-5

然后是编写用户程序，如图2-6：

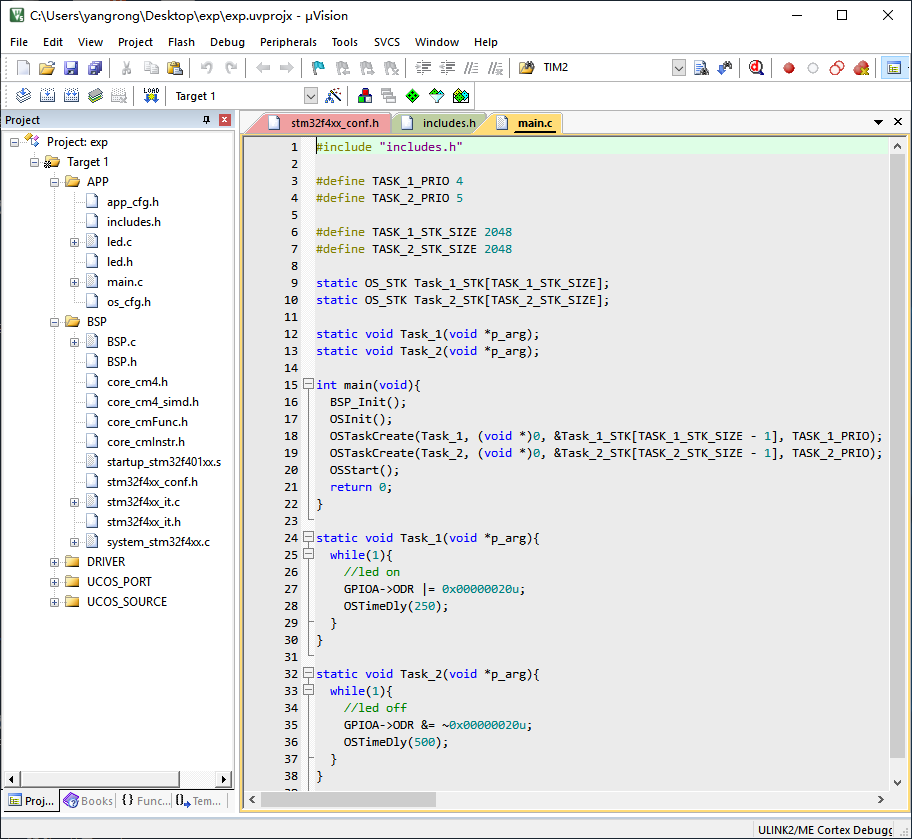


图2-6

最后是编译，如图2-7，显示成功！

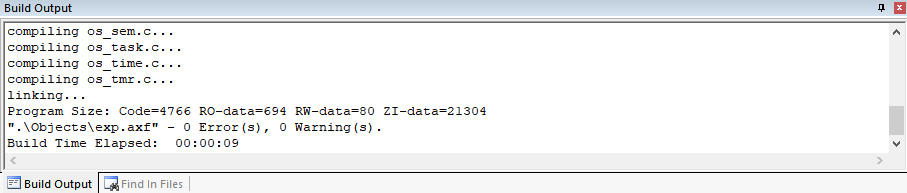


图2-7

实际将工程烧写进开发板，开发板上电复位后成功显示跑马灯程序，说明μC/OS-II操作系统移植成功。

2.2 MPU6050驱动程序的测试

测试工具：串口调试助手、MPU6050、STM32F401RET6

测试步骤：

1. MPU6050与开发板正确连接，开发板连接到PC
2. 打开PC上串口调试助手
3. 检查开发板能否与PC串口通信
4. 将MPU6050读出的数据通过串口打印到PC机上显示

测试结果：

其中ACCEL\_X、ACCEL\_Y、ACCEL\_Z是加速度，TEMP是温度，GYRO\_X、GYRO\_Y、GYRO\_Z是陀螺仪数据。

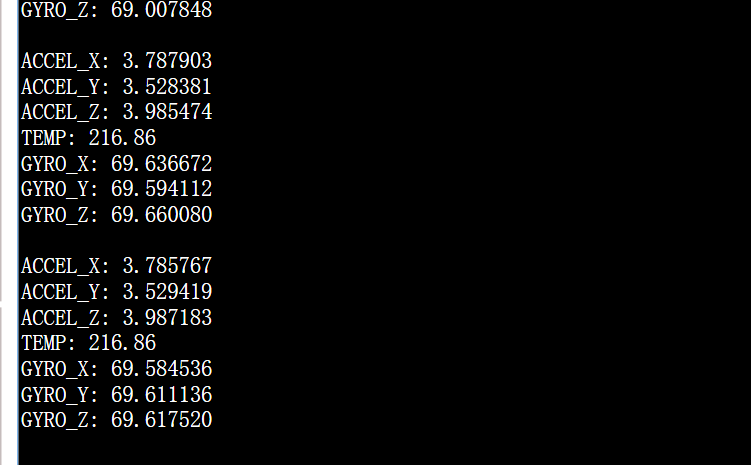


图2-8 串口显示的数据（一）

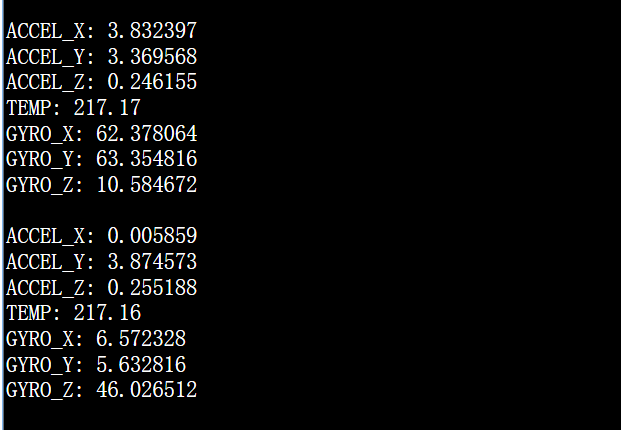


图2-9 串口显示的数据（二）

# 第三章 知识技能学习情况

1.在移植操作系统之前，通过编写裸板驱动，能对stm32开发板的结构及原理有更加深入的理解，并且巩固嵌入式操作系统课上所学的裸板启动的过程。

2.通过对μC/OS-Ⅱ操作系统的移植，我们充分地运用了计算机操作系统与嵌入式操作系统所学的知识，如通过自己编写OSCtxSw()等函数，巩固了嵌入式操作系统所学的任务切换的知识；而操作系统对各部件的控制，则让我们熟练地运用了任务的创建及调度，中断的响应等知识。

3.在裸板驱动的编写过程中，运用ARM处理器体系结构及应用课上所学的知识进行代码的编写，巩固ARM编程基础。同时在其他模块的C语言编程过程中，能够对指针、链表等知识有更加深刻的理解，巩固C语言的高级编程技术。

# 第四章 分工协作与交流情况

表4-1 任务分工

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 姓名 | 学号 | 分工 | 完成情况 |
| 1 | 尧松 | 2017220301012 | 代码编写、报告撰写 | 完成 |
| 2 | 廖新语 | 2017220302012 | 测试、报告撰写 | 完成 |
| 3 | 杨嵘 | 2017220304011 | 代码编写、报告撰写 | 完成 |
| 4 | 卓子豪 | 2017220305017 | 代码编写、报告撰写 | 完成 |

分工协作：本次综合课程设计，我们小组有明确的分工，杨嵘负责裸板驱动与μC/OS-Ⅱ操作系统的移植以及串口驱动代码的编写，卓子豪和尧松负责mpu代码的编写。报告则有每个组员各自完成自己所负责的部分。

交流情况：裸板驱动与μC/OS-Ⅱ操作系统的移植是在课上经组员内部讨论，杨嵘负责编写代码的情况下共同完成的。并且在编写串口驱动、mpu等部分的代码时，组员内部会进行详细的交流，共同解决问题。

# 参考文献

1. NUCLEOF401RE原理图
2. STM32F4xx参考手册
3. STM32F4开发指南-寄存器版本\_V1.1
4. STM32F4开发指南-库函数版本\_V1.1
5. UM1724：STM32 Nucleo 中文用户手册
6. mpu6050手册
7. i2c官方中文手册

# 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，衷心感谢他在我们的课程设计中给予的悉心指导和鼓励，正是他悉心的指导、严谨的治学风格和孜孜不倦的教诲，给了我们无穷的启发和指引。