

uC/OS-II 在 Cortex 上的移植

李 杰

(中国电力科学研究院 北京 100192)

摘要

随着 ARM 的推广应用和发展,ARM9、ARM11 和 Cortex 系列逐步取代了以往的 ARM7 和单片机,可视化多任务操作系统逐步取代了原有的非可视化多任务系统,以往不带操作系统的嵌入式系统也逐步采用多任务系统。利用开放源代码移植多任务操作系统后可以用户编程更为简单和便捷,系统也能实现同时处理多个任务,更加快速实时地响应系统请求。本文详细完整地介绍了如何将 uC/OS-II 移植到 Cortex M0 和 M3 上,完整地列出移植中涉及的关键源代码,并在项目中成功移植和得到验证。

关键词 uC/OS;uCOS;ARM;Cortex;移植

1 引言

uC/OS 是一种基于优先级、开放源代码的多任务操作系统,目前被广泛应用于单片机、ARM7 等嵌入式系统。随着 Cortex 的推广和应用,很多 Cortex 应用场合迫切需要使用 uC/OS 等多任务的操作系统,以便于多任务程序开发和多任务处理。本文将详细介绍 uC/OS 的移植步骤和移植细节,希望能够起到抛砖引玉的作用。

2 主要移植步骤

uC/OS 的主要移植步骤分为:第一步,修改配置文件;第二步,修改函数声明,宏定义包括软中断触发指令;第三步,任务堆栈初始化程序;第四步,创建系统时钟中断初始化和中断处理程序;第五步,修改中断上下文切换处理程序及多任务运行启动程序;第六步,在主程序里初始化和启动多任务系统的运行。在移植过程中涉及修改的文件有 os_cfg.h、os_cpu.h、os_cpu.c、os_cpu_asm.s、系统时钟和主程序文件。

3 移植详细步骤

3.1 修改配置文件

根据自身需求修改 os_cfg.h 配置文件里的配置,各种配置的具体含义和作用在配置文件中有简要的注释,在此不作详述。

3.2 修改函数声明、宏定义

修改 os_cpu.h 文件中的宏定义、函数声明和软中断触发指令。在 os_cpu.h 文件中的宏定义包括数据类型、中断方式、堆栈增长方式、软中断触发指令等定义。数据类型 INT8U、INT8S、INT16U 等宏定义比较简单在此不作说明。

```
#define OS_CRITICAL_METHOD 1 /* 选择直接使用 CPU 开关中断指令方式 */
```

```
#define OS_STK_GROWTH 1 /* 堆栈增长方式为自上而下 */
```

```
#define OS_TASK_TMR_PRIO (OS_LOWEST_PRIO - 2) /*TIMER 任务优先级 */
```

```
#define OS_TASK_SW() NVIC_INT_CTRL &= ~bit27;
```

```

/* 软中断触发指令 */
NVIC_INT_CTRL |= bit28; /* 寄存器地址为
0xE000ED04*/
extern void OS_ENTER_CRITICAL (void); /* 关中断函
数声明 */
extern void OS_EXIT_CRITICAL (void); /* 开中断函数
声明 */
extern void OSIntCtxSw(void); /* 硬中断上下文切换函
数声明 */
extern void OSSWICtxSw (void); /* 本文调用了
OSIntCtxSw 函数,可以不用声明 */
extern void OSStartHighRdy(void);/* 启动多任务系统函
数声明 */
OS_EXT INT32U ENTER_Counter; /* 关中断计数变量
声明 */

```

3.3 修改任务堆栈初始化函数

修改 os_cpu.c 文件中的任务堆栈初始化函数,在初始化赋值时应注意 XPSR、PC、R0 这 3 个寄存器堆栈值不能随意赋值,应按照下列源代码进行赋值,其他寄存器堆栈赋值可以随意赋值。

```

OS_STK *OSTaskStkInit (void (*task)(void *pd), void
*pdata, OS_STK *ptos, INT16U opt){
    OS_STK *stk;
    opt = opt; /* 避免编译器警告 */
    stk = ptos; /* 获取堆栈区指针 */
    *stk = 0x01000000L; /* xPSR 线程模式使用进程堆
栈 */
    *--stk = (OS_STK) task; /* PC(r15) */
    *--stk = 0xeeeeee; /* LR(r14)此初始值随意,以下同 */
    *--stk = 0xcccccc; /* r12*/
    *--stk = 0x33333333; /* r3*/
    *--stk = 0x22222222; /* r2*/
    *--stk = 0x11111111; /* r1*/
    *--stk = (INT32U) pdata; /* r0*/
    *--stk = 0xbbbbbbbb; /* r11*/
    *--stk = 0aaaaaaaa; /* r10*/
    *--stk = 0x99999999; /* r9*/
    *--stk = 0x88888888; /* r8*/
    *--stk = 0x77777777; /* r7*/
    *--stk = 0x66666666; /* r6*/

```

```

*--stk = 0x55555555; /* r5*/
*--stk = 0x44444444; /* r4*/
return (stk);
}

```

3.4 创建系统时钟的中断初始化及其中断处理函数

创建系统时钟中断初始化函数、启动系统时钟函数和中断处理函数,利用系统时钟定时产生中断来实现任务的切换。

```

void SYSTickInit(INT32U mS){//系统时钟初始化
    NVIC_ST_CTRL &= ~(bit0 | bit1 | bit2);//寄存器地址为
0xE000E010
    NVIC_ST_RELOAD = (SYSClk*250)*mS - 1; //寄存
器地址为 0xE000E014
    NVIC_ST_CURRENT = 0;// 寄存器地址为 0xE000E018
}
void SysTick_Start(void){ //启动系统时钟
    OS_ENTER_CRITICAL_Counter = 0;
    NVIC_ST_CTRL = bit0 | bit1 ;
}
void SysTick_ISR(INT32U ptr){//系统时钟中断处理程序
    OSTimeTick();
    OSIntExit();
}

```

3.5 修改软中断和硬中断上下文切换程序

修改 os_cpu_asm.s 文件中的软中断和硬中断上下文切换程序、多任务运行启动程序。软中断和硬中断的向量需要挂接到中断向量表中,在此不做详述。

(1)内部函数向外声明和外部函数引入声明

```

EXPORT OS_ENTER_CRITICAL ;关中断
EXPORT OS_EXIT_CRITICAL ;开中断
EXPORT OSIntCtxSw ;寄存器上下文切换
EXPORT OSStartHighRdy ;多任务运行启动
EXPORT SysTickHandler ;系统时钟中断处理
EXPORT PendSVHandler ;软中断处理
IMPORT ENTER_Counter ;开关中断计数
IMPORT OSRunning ;系统运行指示
IMPORT OSTCBCur ;指向当前任务 TCB 的指针
IMPORT OSTCBHighRdy ;指向将要运行的任务
TCB 的指针
IMPORT OSPrioCur ;当前任务的优先级

```

IMPORT OSPrioHighRdy ;将要运行的任务的优先级
IMPORT SysTick_ISR ;系统时钟中断处理 C 语言
部分

(2)汇编代码包括中断处理、上下文切换、中断的开关
以及多任务运行启动程序

Cortex M3 的汇编源代码如下:

```
AREARWdata, DATA, READWRITE
PRESERVE8
AREA OS_cpu_Code, CODE, READONLY
THUMB
PendSVHandler ;Pend 软中断处理程序
PUSH {R0-R2,LR};压栈
MRS R0,PSP
STMFD.W R0!,{R4-R11} ;R11 最先入栈
MSR PSP,R0 ;更新 PSP
BL OSIntCtxSw ;调用上下文切换程序
MRS R0,PSP
LDMFD.W R0!,{R4-R11} ;R11 最后出栈
MSR PSP,R0;清除 PendSV 中断触发位
LDR.W R0, =0xE000ED04
LDR.W R2,[R0]
LDR.W R1, =0x80000000
ORR.W R2,R2,R1
STR.W R2,[R0]
POP {R0-R2,PC};中断返回
SysTickHandler;系统时钟中断处理程序
PUSH {R0,LR}
MRS R0,PSP
STMFD.W R0!,{R4-R11} ;R11 最先入栈
MSR PSP,R0 ;更新 PSP
BL SysTick_ISR ;调用 C 语言中断处理程序
MRS R0,PSP
LDMFD.W R0!,{R4-R11} ;R11 最后出栈
MSR PSP,R0
POP {R0,PC} ;中断返回
OS_ENTER_CRITICAL ;关中断
CPSID I
PUSH {R0-R1,LR}
LDR.W R0, =OS_ENTER_CRITICAL_Counter
LDR.W R1, [R0]
```

```
CMP.W R1, #255
BEQ OS_ENTER_END
ADD.W R1,R1,#1
STR.W R1,[R0]
OS_ENTER_END
POP {R0-R1,PC}
OS_EXIT_CRITICAL;开中断
PUSH {R0-R1,LR}
LDR.W R0, =OS_ENTER_CRITICAL_Counter
LDR.W R1, [R0]
CBZ R1, OS_EXIT_END0
SUBS.W R1,R1,#1
STR.W R1,[R0]
BNE OS_EXIT_END
OS_EXIT_END0
CPSIE I
OS_EXIT_END
POP {R0-R1,PC}
OSIntCtxSw ;上下文切换程序
PUSH {R0-R2,LR};保存 PSP
MRS R0,PSP
LDR.W R1,=OSTCBCur
LDR.W R1,[R1]
STR.W R0,[R1];更新 TCB 的 OSPrioCur
LDR.W R0, =OSPrioHighRdy
LDR.W R1, =OSPrioCur
LDRB.W R0, [R0]
STRB.W R0, [R1];更新 TCB 的 OSTCBCur
LDR.W R1, =OSTCBHighRdy
LDR.W R2, =OSTCBCur
LDR.W R0, [R1]
STR.W R0, [R2];读出新 TCB 里的栈指针并暂存到 R0
LDR.W R0, [R0];更新 PSP
MSR PSP,R0;函数返回
POP {R0-R2,PC}
OSStartHighRdy ;启动优先级最高的任务开始运行 uC/OS
;更新 OSRunning 为 1
LDR.W R0, =OSRunning
MOV.W R1, #1
STRB.W R1, [R0];使用进程堆栈
```

```

MOV.W   R0, #2
MSR      CONTROL, R0;读出 SP
LDR.W    R0, =OSTCBCur
LDR.W    R0, [R0]
LDR.W    SP, [R0];读出第一个参数
ADD.W    SP, SP, #32
POP      {R0};读出 PC
ADD.W    SP, SP, #20
POP      {R1} ;调整堆栈指针
ADD      SP, #4
;转入任务运行
BX       R1
END

```

函数声明部分 Cortex M0 和 M3 的声明是一样,但是汇编代码部分尤其是寄存器保存和恢复部分 M0 和 M3 的方式是不一样的。由于 Cortex M0 不能使用 STMFD 和 LDMFD,所以只能对 Cortex M0 寄存器进行逐个保存和恢复。

R0~R7 的保存和恢复可以使用下列方式:

```

LDR      R5, [R0] ;POP R5
STR      R5, [R0] ;PUSH R5

```

R8~R15 寄存器的保存和恢复不能使用寻址方式访问,需要选用 R0~R7 其中的一个寄存器作为中转。如下所示:

```

;POP R8
LDR      R1, [R0]
MOV      R8, R1
;PUSH R8
MOV      R1, R8
STR      R1, [R0]

```

其中 R0 是堆栈地址指针。汇编代码其他部分 M0 和 M3 只需将 M3 的指令换成 M0 的指令即可。SP 也是不能使用寻址方式保存到内存或从内存读取。

3.6 初始化 uCOS 和启动 uCOS

在主函数里初始化 uCOS 和启动 uCOS 的运行后,至

此 uCOS 整个移植过程完成。

```

OS_STK   StartTaskSTK[128];//分配堆栈区
Void StartTask(void *Pdata){
SysTick_Start();//启动系统时钟
#if OS_TASK_STAT_EN > 0
OSStatInit(); //统计初始化
#endif
/* 添加用户程序 */
}

int main(void){
SYSTickInit(1);//系统时钟初始化
OSInit();//uCOS 初始化
OSTaskCreate(StartTask, NULL, &TaskSTK[128 - 1], 5);//
创建新任务
OSSStart();//uCOS 启动
return 0;
}

```

4 结束语

uC/OS 按照上述方法移植后,在 Cortex M3 和 M0 芯片上经验证均可以可靠稳定地运行。由于使用了通用系统时钟作为定时时钟中断源,所以整个系统的移植与具体的芯片型号无关,只与 Cortex 系列有关,比如 M3 与 M0 系列个别汇编代码是有区别的,但是如果同属于 M3 或 M0 的芯片尽管型号不同但在移植时 uC/OS 代码是一样的。

参考文献

- 1 uC/OS-II application note .uC/OS II and the ARM Processor. www.Micrium.com
- 2 Jean J L. 邵贝贝译.uC/OS-II——源码公开的实时嵌入式操作系统.北京:中国电力出版社,2001

(收稿日期:2012-10-11)