基于 MicroBlaze 的 μC/OS-II 操作系统移植①

常华利[1,2], 尹震宇[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 沈阳计算技术研究所, 沈阳 110171)

摘 要: LT-H10 滚齿机数控系统采用基于 ARM 的系统结构, 其处理器等性能、资源比以往基于 x86 的系统受到很大的限制, 所以CPU的占用率也相对较高. 为了降低ARM CPU的占用率把系统运行的部分主站控制驱动程序迁移到底板 FPGA MicroBlaze 软核处理器上运行, 本文提出了一种基于 MicroBlaze 软核处理器的 μC/OS-II 的移植方案. 测试实验结果表明μC/OS-II系统移植到MicroBlaze之后能稳定的运行. 快速的任务上下文切换更有利于数控系统的实时性. 针对 MicroBlaze μC/OS-II 系统和 ARM Linux 系统设计了两个不同的任务调度算法对任务上下文切换的时间开销进行研究、测量和分析. 此研究方案不仅可以满足基于 ARM 的数控系统的应用需要, 同时适用于基于x86的数控系统, 达到降低系统CPU占用率的目的, 在嵌入式数控系统中具有重要的研究意义与应用价值.

关键词: FPGA; MicroBlaze; μC/OS-II; 移植

μC/OS-II System Porting Based on MicroBlaze Processor

CHANG Hua-Li[1,2], YIN Zhen-Yu[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Shenyang Institute of Computer Technology, Chinese Academy of Sciences, Shenyang 110171, China)

Abstract: The LT-H10 hobbing machine numerical control system is an embedded system based on ARM processor. The CPU occupancy in ARM-based NC system is always higher than that in x86-based systems for some characteristic of ARM processor itself limits. A μC/OS-II system porting scheme based on MicroBlaze soft-core processor is presented in this paper, to reduce the occupancy of ARM processor by executing some driver programs of master station on the MicroBlaze soft-core processor of FPGA chip on the bottom plate. The experimental test results show that the μC/OS-II system ported on the MicroBlaze processor can run stably. The rapid context-switch benefits to the real-time of the NC system. We design two different tasks scheduling algorithm to measure and research the context-switch time in MicroBlaze-based μC/OS-II system and ARM-based Linux system. This study can not only meet the application needs of CNC system based on ARM, but also can been applied to CNC system based on X86, to achieve the purpose of reducing CPU occupancy, which has important research meaning and application value in Embedded CNC system.

Key words: FPGA; MicroBlaze; μC/OS-II; porting

LT-H10 控制系统是针对滚齿机的专用控制系统, 是基于 ARM 处理器的嵌入式数控系统[1], 底层硬件采用了 FPGA 实现系统主站, 完成基本的电平转换、信号采集等功能. 系统主站要实现伺服-脉冲控制, 总线协议, I/O 控制, 数字光栅尺控制, 计数器, 主轴控制等功能. 进一步的这些控制功能还需要实现在驱动程序部分, 运行这些功能会占用大量 ARM 主处理器资源[2], 而且控制指示灯以及超时报警的检测、处理等也会占用 ARM CPU 资源. 此外, GPMC 总线带宽有限, 传输原始数据时, 也会带来总线的大量开销, 并不利

① 基金项目:国家科技重大专项(2014ZX04009031)
收稿时间:2016-08-12;收到修改稿时间:2016-09-23 [doi: 10.15888/j.cnki.csa.005742]

于实时性.

本文以 LT-H10 控制系统为研究对象, 提出了一种基于 MicroBlaze 软核处理器的 μC/OS-II 的移植[3]方案. 把系统运行的部分主站控制驱动程序迁移到底板 FPGA MicroBlaze 软核处理器上运行, 以达到降低 ARM CPU 占用率的目的.

此研究方案不仅适用于基于 ARM 的数控系统, 同时适用于基于 x86 的数控系统, 达到降低系统 CPU 占用率的目的, 在嵌入式数控系统中具有重要的应用价值.

1 μC/OS-II、MicroBlaze概述及系统移植的可行性分析

1.1 实时操作系统 μC/OS-II

μC/OS-II 操作系统[4]目前在工业领域中使用的非常广泛, 作为一种源代码公开的实时嵌入式操作系统, 最多可支持 63 个任务, 其内核为占先式(即执行就绪态中优先级最高的任务)并支持信号量、邮箱、消息队列等多种进程间通信机制; 同时用户可以根据需要对内核中的功能模块进行裁剪[5]; 并且 μC/OS-II 操作系统代码很简洁, 仅有 5500 行代码, 而且其大部分代码是使用 C 语言编写的, 只有与硬件相关的少量代码需要使用目标板支持的汇编语言[6]. 所以说 μC/OS-II 具有内核小巧、可裁剪、实时性强、多任务、方便移植到多种嵌入式平台等特点, 使得各个任务独立工作, 互相不干涉, 从而很容易使得任务被准确执行, 使实时应用程序的设计和扩展变得简单[7,8].

1.2 软核处理器 MicroBlaze

Xilinx 首创了现场可编程逻辑阵列(FPGA)这一创新性的技术, 随着 FPGA 的技术的发展, 其逻辑容量逐步增大而成本越来越低[9]. 这种趋势下, FPGA 可以代替系统中越来越多的器件, 并且发展到 FPGA 片上系统. Xilinx 推出了 32 位软核处理器 MicroBlaze, 用来替代片上的微控制器, 有效的缩小了 PCB 板的面积和器件数量, 降低了整个系统的成本. 该处理器采用 32 位 RISC 哈佛存储结构, 拥有极其灵活的架构以及丰富的指令集. MicroBlaze 软核处理器具有占用资源少、可配置性强的特点, 可以和其他外设 IP 核一起, 完成可编程芯片的设计. MicroBlaze 软核处理器采用 Xilinx 公司提供的嵌入式开发套件 EDK 进行设计开发, 设计套件 DEK 提供了硬件平台开发环境 XPS 和软件平台开发环境 SDK[10].

1.3 μC/OS-II 代码结构分析以及处理器相关核心代码

从 μC/OS-II 移植的过程来看, μC/OS-II 代码逻辑上可以分为三大组成部分: 与处理器无关的内核代码、与处理器有关的核心代码、与软硬件环境设置有关的配置代码[11]. 如图 1 是 μC/OS-II 操作系统的代码结构.

图 1　μC/OS-II 的代码结构

其中与处理器无关的内核代码主要实现是任务管理、信号量、内存管理、消息队列、系统调用等功能(主要包括 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、ucos_ii.h、ucos_ii.c);

与处理器有关的核心代码主要与操作系统的移植相关(主要包括 os_cpu.h、os_cpu_a.s、os_cpu_c.c);

与软硬件环境设置有关的配置代码主要用于裁剪和设置操作系统(主要包括 os_cfg.h、includes.h).

1.4 μC/OS-II 在 MicroBlaze 上移植的可行性分析

移植的目标是使得这个实时内核能够在软核处理器 MicroBlaze 上得到运行. 要使 μC/OS-II 能够正常运行, 处理器必须要满足:

1) 处理器的 C 编译器能够产生可重入代码, 并且使用 C 语言就可以完成开关中断;

2) 处理器支持中断;

3) 处理器支持可以容纳一定数据量的硬件堆栈;

4) 处理器有把堆栈指针和其他寄存器读出和存储到堆栈或者是内存中的指令.

DEK 软件开发套件内置的编译器能够产生可重入代码, 并支持内嵌汇编语言, 在 C 语言环境中可进

行任意的开关中断操作; MicroBlaze 有专门的堆栈指针寄存器, 足够大的内部 BRAM 容量能够完全硬件堆栈; 另外, MicroBlaze 的指令集包含有丰富的指令对堆栈进行操作, 可以方便的对处理器中的寄存器进行堆栈操作. 所以, μC/OS-II 完全可以移植到软核处理器 MicroBlaze 上.

2 系统移植设计和相应测试实验

2.1 使用 ISE14.7 集成软件环境搭建系统移植环境

主要设备和软件: PC 机一台、ISE14.7 集成软件环境以及图 2 所示的 FPGA 测试板.

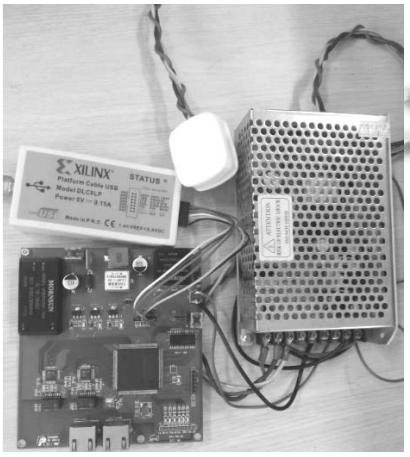

图 2 FPGA 测试板、Xilinx JTAG 下载器, 和 FPGA 测试板电源盒

利用 EDK 的板级开发包向导(BSB Wizard)对硬件平台进行定制, 处理器时钟频率是 50MHZ, 内存约束至 32KB, 添加一个 GPIO IP 核.

搭建系统移植环境的步骤:

1) 使用 ISE14.7 建立工程

根据所用 FPGA 芯片设置对应测试板信息: 架构是 Spartan3e, 设备型号是 xc3s500e, 封装是 pq208, 速度是-4.

2) 向工程添加 VHDL Module, 修改 cpuv01.vhd 文件, 如图 3.

3) 向工程添加 Embedded processor 文件, 自动开启 EDK 板级开发包向导(BSB Wizard)构建硬件平台. 如图 4 所示, 处理器时钟频率是 50MHZ, 内存 32KB.

4) IP 核地址设置

如图 5 所示, 设置 IP 核地址: GPIO 的基地址是 0x90000000, 高地址是 0x900001FF.

```
entity cpuv01_top is
  port (
    fpga_0_clk_1_sys_clk_pin : in std_logic;
    fpga_0_rst_1_sys_rst_pin : in std_logic;
    xps_gpio_0_GPIO_IO_pin : inout std_logic_vector(0 to 5)
);
end cpuv01_top;

architecture STRUCTURE of cpuv01_top is

  component cpuv01 is
    port (
      fpga_0_clk_1_sys_clk_pin : in std_logic;
      fpga_0_rst_1_sys_rst_pin : in std_logic;
      xps_gpio_0_GPIO_IO_pin : inout std_logic_vector(0 to 31)
    );
  end component;

 attribute BOX_TYPE : STRING;
 attribute BOX_TYPE of cpuv01 : component is "user_black_box";

 signal xps_gio_bus:std_logic_vector(0 to 31);

begin

  xps_gio_bus(8 to 31)<=X"FFFFFF";
  xps_gio_bus(7)<='1';
  xps_gio_bus(6)<='1';
  xps_gpio_0_GPIO_IO_pin(0 to 5)<=xps_gio_bus(0 to 5);

  cpuv01_i : cpuv01
    port map (
      fpga_0_clk_1_sys_clk_pin => fpga_0_clk_1_sys_clk_pin,
      fpga_0_rst_1_sys_rst_pin => fpga_0_rst_1_sys_rst_pin,
      xps_gpio_0_GPIO_IO_pin => xps_gio_bus
    );
```

图 3 cpuv01.vhd 文件

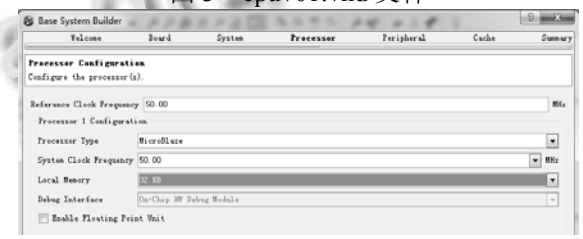

图 4 处理器信息设置

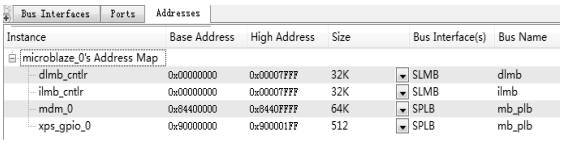

图 5 设置 IP 核地址

5) 添加 UCF 约束文件

UCF 文件用来添加信号的管脚约束与时序约束, 通过文本编辑器来修改.

#Generic Template

Net fpga_0_clk_1_sys_clk_pin TNM_NET = sys_clk_pin

TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 50000 kHz;

Net fpga_0_clk_1_sys_clk_pin LOC=P77;

Net fpga_0_rst_1_sys_rst_pin TIG;

Net fpga_0_rst_1_sys_rst_pin LOC=P54;

Net xps_gpio_0_GPIO_IO_pin<5> LOC=P96;

Net xps_gpio_0_GPIO_IO_pin<4> LOC=P97;

Net xps_gpio_0_GPIO_IO_pin<3> LOC=P98;

Net xps_gpio_0_GPIO_IO_pin<2> LOC=P99;

Net xps_gpio_0_GPIO_IO_pin<1> LOC=P100;

Net xps_gpio_0_GPIO_IO_pin<0> LOC=P102;

把串口的输入输出信号与 FPGA 芯片相应管脚相

连, 时钟信号引脚为 P77, 复位引脚为 P54, 测试板上 P96、P97、P98、P99、P100 和 P102 对应测试板上一组 LED, 并设置输入时钟频率为 50MHZ.

通过以上步骤完成硬件平台的建立, 导出硬件平台文件到 SDK, 在 SDK 中新建 Application Project, 就完成了系统移植环境的搭建, 接下来就可以开始移植 μC/OS-II 操作系统到 MicroBlaze 处理器.

2.2 移植的步骤

1) 使用 SDK 软件开发套件为 μC/OS-II 操作系统建立一个目录, 在该目录下创建工程, 加入 μC/OS-II 内核文件到这个工程.

2) 根据测试板硬件环境移植 os_cpu.h

os_cpu.h 包含和 CPU 相关的不能通用的数据类型的定义、变量声明、函数代码的声明部分. 部分数据类型的定义:

```
typedef unsigned  char  BOOLEAN;
typedef unsigned  char  INT8U;
typedef signed    char  INT8S;
typedef unsigned  short  INT16U;
typedef unsigned  int  INT32U;
typedef signed  int  INT32S;
typedef unsigned  int  OS_STK;
typedef unsigned  int  volatile  OS_CPU_SR;
```

由于不同的处理器有不同的字长, μC/OS-II 的移植包括一系列类型定义, 以确保可移植性. MicroBlaze 是 32 位处理器, 堆栈入口宽度是 32 位, 所以 OS_STK 被定义为 32 位, 所有的任务堆栈也都用 OS_STK 来声明, 状态寄存器也是 32 位的, 所以 OS_CPU_SR 也被定义为 32 位.

```
#define  OS_CRITICAL_METHOD    3
#define  OS_ENTER_CRITICAL()cpu_sr=
OS_CPU_SR_Save();
#define  OS_EXIT_CRITICAL()
 OS_CPU_SR_Restore(cpu_sr);
```

μC/OS-II 在进入系统临界代码段之前要进行关闭中断, 退出临界代码段时要打开中断, 从而保证核心数据不被多任务环境下的其他任务中断破坏. 在 μC/OS-II 操作系统中提供了三种针对 MicroBlaze 的处理中断的方式, 这里用到了第三种方式, 即 OS_CRITICAL_METHOD==3, 用户可以取得当前处理器状态字的值, 保存在 C 函数的局部变量中, 这个变量就是用来恢复程序状态字的值. 另外, MicroBlaze 的堆栈地址由低到高增长, 故 OS_STK_GROWTH==1.

3) 根据测试板硬件环境移植移植 os_cpu_c.c

需要编写 10 个简单的 C 函数, 分别是 OSInitHookBegin()、OSInitHookEnd()、OSTaskCreateHook()、OSTaskDelHook()、OSTaskIdleHook()、OSTaskStatHook()、OSTaskStkInit()、OSTaskSwHook()、OSTCBInitHook()、OStimeTickHook(). 其中唯一必要的函数是 OSTaskStkInit(), 此函数用来初始化任务堆栈, 初始状态的堆栈模拟发生一次中断后的堆栈结构在初始化堆栈寄存器时用与寄存器号相对应的值进行初始化, 目的是便于调试. 其余 9 个必须声明, 但不一定要包含任何代码.

4) 根据测试板硬件环境移植 os_cpu_a.s.

对 OS_CPU_A.S 进行移植, 主要是编写 4 个汇编语言函数, 分别是: 多任务启动函数调用的一个函数 OSStartHighRdy()、任务切换函数 OSCtxSw()、中断任务切换函数 OSIntCtxSw() 和中断服务函数 OS_CPU_ISR() . μC/OS-II 启动时调用 OSStart() , 而 OSStart()又调用 OSStartHighRdy( )运行优先级最高的任务. OSCtxSw()的调用发生在有更高优先级的任务就绪或者当前任务不可执行的情况下. 如果任务执行了某个函数, 其结果改变了当前任务的状态, 或是改变了别的任务状态都会引起新的任务调度. 判断有比中断更重要的任务在运行时, OSIntCtxSw()被 OSIntExit()调用. MicroBlaze 为每一个中断都提供一个单独的中断向量, Xilinx 的 XPS 工具会自动的生成中断向量表, 地址空间为 0x00000010-0x00000014. OS_CPU_ISR()用于判断中断源以及执行器件所对应的中断语句和函数.

2.3 编写应用程序测试操作系统

为了验证 μC/OS-II 操作系统是否成功移植到软核处理器 MicroBlaze 中, 编写应用程序进行测试, 具体思路是:

首先建立主程序, 如 main.c. 主程序中的入口函数 OSInit 执行操作系统初始化函数;

其次是调用 OSTaskCreate()创建控制测试板 LED 的任务 AppTaskFirst, 其优先级为 5 用来控制引脚分别为 P96、 P98、P100 的 LED 的亮灭;

最后 OSStart()开启多任务, 调用 OSStartHighRdy()运行优先级最高的任务, 任务 AppTaskFirst 优先级最高, 会循环执行.

设计测试程序代码如下:

```
int main()
{
INT8U       err;
OSInit();/*初始化 uC/OS-II 操作系统   */
  OSTaskCreate(AppTaskFirst,/* 创 建 第 任 务 AppTaskFirst*/
(void *)0,
 &AppTaskFirstStk[APP_TASK_FIRST_STK_SIZE - 1],/* APP_TASK_FIRST_STK_SIZE 是 1024*/
APP_TASK_FIRST_PRIO /*优先级是 5 */
);
OSStart();/*  开启多任务*/
return 0;
}
void delay(uint cnt) /*定义延迟函数 delay*/
{
      uint i,j;
      for(i=0;i<cnt;i++)
            for(j=0;j<256;j++);
}
void led_on()
{
unsigned int *dport;
dport=0x90000000;  /*GPIO  IP 核对应的内存起始地址*/
XGpio_WriteReg(dport,1,0xA8FFFFFF);  /*点亮引脚分别为 P96、 P98、P100 的 LED*/
delay(M);
  XGpio_WriteReg(dport,1,0xFFFFFFFF);/*  熄  灭 LED */
delay(M);
}
static   void   AppTaskFirst (void *p_arg)
{
      (void)p_arg; /*防止编译器产生警告*/
      led_on();
}
```

编译应用程序, 并通过 Program FPGA 把软件程序生成的可执行可链接格式文件(.ELF)同硬件比特流文件(.BIT)下载到 FPGA 的测试板. 设置断点调试代码, 通过跟踪调试移植的 μC/OS-II 代码和观察 LED 的亮暗情况, 不断修改代码直到确定操作系统正常运行, 至此则 μC/OS-II 操作系统移植成功.

3 测量任务切换时间

3.1 在 ARM 处理器上测量任务切换的时间

实验环境: 安装 CentOS Linux 系统的 ARM Cortex-A8 TQ335X 开发板、安装交叉编译工具链的 CentOS Linux 系统.

设计一个基于优先级的可抢占的任务调度算法: 创建两个具有不同优先级的实时任务 taskA 和 taskB, 其中 taskA 的优先级高于 taskB 的优先级, 两个任务每隔 5 毫秒发生一次调度. taskA 和 taskB 主体部分均可用如下伪代码表示.

```
while (1)
{
clock_gettime();
//延时代码
clock_gettime();
nexttime+=clcyetime;
clock_nanosleep();
}
```

任务开启时, 调用 clock_gettime()函数取得一次系统时间, 接着执行一段延迟代码, 再次调用 clock_gettime()函数取得一次系统时间(两个时间的差值看做任务一次执行的周期), 然后设置任务调度周期(这里设为 5 毫秒, 也可以是其它合理值), 最后调用 clock_nanosleep()将任务挂起直到这一调度周期结束, 将开始下一次循环.

单独运行 taskA 的进程时的部分结果:

```
taskA 0 start time= 781 sec 373724720 nsec
taskA 0 end time= 781 sec 375083470 nsec
taskA 1 start time= 781 sec 378944136 nsec
taskA 1end time= 781 sec 380305595 nsec
taskA 2 start time= 781 sec 383982555 nsec
taskA 2 end time= 781 sec 385328762 nsec
```

可以看出 taskA 每隔 5 毫秒执行一次, 并且使用 30 组数据计算得到 taskA 的平均执行时间是 1334.229

微秒.

单独运行 taskB 的进程时的部分结果:

taskB 0 start time= 788 sec 573482680 nsec

taskB 0 end time= 788 sec 574175930 nsec

taskB 1 start time= 788 sec 578687844 nsec

taskB 1 end time= 788 sec 579380471 nsec

taskB 2 start time= 788 sec 583729893 nsec

taskB 2 end time= 788 sec 584407892 nsec

可以看出 taskB 每隔 5 毫秒执行一次, 并且使用 30 组数据计算得到 taskB 的平均执行时间是 661.337 微秒.

同时运行 taskA 和 taskB 的进程时的部分结果:

taskB 0 start time= 387 sec 585822089 nsec

taskA 20 start time= 387 sec 586094964 nsec

taskA 20 end time= 387 sec 587432131 nsec

taskB 0 end time= 387 sec 587925048 nsec

taskB 1 start time= 387 sec 590983131 nsec

taskA 21 start time= 387 sec 591130049 nsec

taskA 21 end time= 387 sec 592465550 nsec

taskB 1 end time= 387 sec 593067508 nsec

可以看出 taskA 和 taskB 每隔 5 毫秒执行一次且运行时发生 CPU 抢占, taskB 的进程运行时发现有优先级更高的 taskA 的进程就绪时, 发生一次任务切换, taskA 的进程运行结束退出 CPU 发生第二次任务切换, taskB 的进程接着运行直到进程结束, 然后下一调度周期, 开始新的一轮的进程调度. 所以, 两任务共同执行的时间应该包括 taskA 单独执行的时间、taskB 单独执行的时间以及两次任务切换的时间.

表 1　19 组任务切换的时间及平均时间

Num	两任务共同执行时间/微秒	切换时间/微秒
1	2102.959	56.711
2	2084.377	44.420
3	2051.209	27.836
4	2053.583	29.023
5	2051.625	28.044
6	2047.375	25.919
7	2061.417	32.940
8	2048.042	26.253
9	2047.000	25.732
10	2052.041	25.252
11	2047.250	25.857
12	2046.625	25.544
13	2047.292	25.878
14	2046.333	25.398
15	2047.000	25.732
16	2046.750	25.607
17	2046.583	25.523
18	2046.583	25.523
19	2046.958	25.711
平均值	2053.737	29.100

由表 1 可以看出, taskA 和 taskB 的共同执行时间是的平均值是 2053.737 微秒, 任务切换的性能: 任务切换的平均时间 29.100 微秒, 最差情况下是 53.711 微秒.

3.2 在 MicroBlaze 软核处理器上测量任务切换时间

实验环境: ISE14.7 集成软件环境、FPGA 测试板和示波器.

1) 设计基于优先级的任务调度算法

为了测量任务在 MicroBlaze 软核处理器上切换时间, 设计了一个基于优先级的任务调度算法, 具体思路是: 任务创建时调用 OSTaskCreate()创建两个控制测试板 LED 的任务, 任务 AppTaskFirst 优先级为 5 用来控制引脚分别为 P96、P98、P100 的 LED 的亮灭, 任务 AppTaskSecond 优先级为 6 用来控制引脚分别为 P97、 P99、P102 的 LED 的亮灭; OSStart()开启多任务, 调用 OSStartHighRdy()运行优先级最高的任务 AppTaskFirst, 任务 AppTaskFirst 完成控制引脚分别为 P96、P98、P100 的 LED 的亮灭之后调用 OSTaskSuspend 将自身挂起, 进行任务切换, 执行系统里优先级最高的任务 AppTaskSecond, 任务 AppTaskSecond 完成控制引脚分别为 P97、 P99、P102 的 LED 的亮灭之后调用 OSTaskResume 将任务 AppTaskFirst 唤醒, 接下来的时间任务就在 AppTaskFirst 和 AppTaskSecond 之间不断来回切换 .

相关程序代码如下:

```
void delay(uint cnt) /*定义延迟函数 delay */
{
    uint i,j;
    for(i=0;i<cnt;i++)
        for(j=0;j<256;j++);
}
void led1_on()
{
unsigned int *dport;
```

```
dport=0x90000000; /*GPIO IP 核对应的内存起始地址*/
XGpio_WriteReg(dport,1,0xA8FFFFFF); /*点亮引脚分别为 P96、 P98、P100 的 LED */
delay(M); /*延迟参数 M*/
 XGpio_WriteReg(dport,1,0xFFFFFFFF); /* 熄 灭 LED*/
delay(N); /*延迟参数 N*/
}
void led2_on()
{
unsigned int *dport;
dport=0x90000000;
XGpio_WriteReg(dport,1,0x57FFFFFF); /*点亮引脚分别为 P97、 P99、P102 的 LED*/
delay(M); /*延迟参数 M*/
 XGpio_WriteReg(dport,1,0xFFFFFFFF); /* 熄 灭 LED*/
delay(N); /*延迟参数 N*/
}
static void AppTaskFirst (void *p_arg)
{
    (void)p_arg; /*防止编译器产生警告*/
    led1_on();
    OSTaskSuspend(5);
}
static void AppTaskSecond (void *p_arg)
{
    (void)p_arg; /*防止编译器产生警告*/
    led2_on();
    OSTaskResume(5);
}
```

2) 估算任务的切换时间

为了得到任务切换的时间这里使用示波器测量了测试板 P96、P98 两个管脚对应的 I/O 口的电平变换. 对应不同延迟参数 M 和 N 使用示波器来测量 LED 亮和暗的延迟. 如图 6 和图 7 分别是延迟参数 M 和 N 均为 4 时使用示波器测得的 LED 亮的延迟和 LED 暗的延迟. 由图 6 可知 LED 亮的延迟是 252 微秒, 由图 7 可知 LED 暗的延迟是 272 微秒.

图 6 M 和 N 均为 4 时 LED 亮的延迟

图 7 M 和 N 均为 4 时 LED 暗的延迟

由测试程序的代码可知, LED 亮的延迟是调用 delay 的延迟; LED 暗的延迟调用 delay 的延迟与任务切换时间开销之和. 这里 LED 亮的延迟大小和 LED 暗的延迟中调用 delay 的延迟大小应该是相等, 故可得到任务切换时间是 20 微秒.

由于使用示波器测得的数值带一定误差以及电平变换有延迟等原因, 所得到的结果也是有误差的. 所以为了能够更好的分析 LED 亮的延迟和 LED 暗的延迟, 这里使用了统计的办法: 针对延迟参数 M 和 N, 设计了 16 组不同的实验, 每组实验用示波器测得 LED 亮的延迟 T1 和 LED 暗的延迟 T2.

由于 LED 亮的延迟是调用 delay 的延迟和 LED 暗的延迟中调用 delay 的延迟是关于 M 和 N 对应成比例的, 即 T1/M=T2/N, 故 T2-N*T1/M 是任务切换的时间.

表 2 针对 M 和 N 的 16 组不同实验及其结果

M	N/微秒	T1/微秒	T2	T2-N*T1/M
2	2	118	134	16
2	4	116	256	24
2	6	116	376	28
2	8	116	496	32
4	2	236	141	23
4	4	242	272	30
4	6	242	392	29

4	8	249	521	23
6	2	361	153	33
6	4	345	265	35
6	6	345	377	33
6	8	345	489	29
8	2	505	153	27
8	4	473	281	41
8	6	489	393	27
8	8	473	505	32
	平均值			28.875

由表 2 可知, (T2-N*T1/M)在落在一个很小的区间[16 微秒,41 微秒]内, 结果是有意义的. 这里(T2-N*T1/M)的平均值是 28.875 微秒, 最大值 41 微秒, 据此得到任务切换的性能: 任务切换的平均时间 28.875 微秒, 任务切换的最差情况是 41 微秒.

3.3 实验结论

任务在 ARM Linux 系统上切换的平均时间 29.100 微秒, 最差情况是 53.711 微秒; 任务在 MicroBlazeμC/OS-II 系统上切换的平均时间是 28.875 微秒, 与在 ARM Linux 系统下结果一致, 最差情况是 41 微秒较 ARM Linux 系统要好一些. 实验结果表明 μC/OS-II 系统移植到 MicroBlaze 之后, 不仅可以稳定的运行, 而且可以实现快速的任务切换, 符合数控系统的实时性要求, 可以满足 LT-H10 滚齿机控制系统的应用需要.

4 结语

本文以 LT-H10 滚齿机控制系统为课题背景, 针对系统 CPU 占用率高的问题, 为了降低系统 CPU 占用率, 把系统运行的部分主站控制驱动程序迁移到底板 FPGA MicroBlaze 软核处理器上运行, 提出了一种基于 MicroBlaze 软核处理器的 μC/OS-II 的移植方案. 使用 ISE14.7 集成软件环境构建移植环境将 μC/OS-II 移植 FPGA MicroBlaze 软核处理器上, 通过测试 μC/OS-II 在 MicroBlaze 可以稳定运行. 针对 MicroBlazeμC/OS-II 系统和 ARM Linux 系统设计了两个不同的任务调度算法对任务的切换时间进行测量和研究. 实验结果表明移植到 MicroBlaze 上的 μC/OS-II 系统不仅可以稳定的运行, 而且可以实现快速的任务切换, 符合数控系统的实时性要求, 可以满足 LT-H10 滚齿机控制系统的应用需要.

为了降低 ARM CPU 的占用率还有许多工作要做, 首先要实现 ARM 处理器 Linux 操作系统与 FPGA MicroBlaze 软核处理器 μC/OS-II 操作系统的通信设计. 在 LT-H10 滚齿机控制系统中主站要实现伺服-脉冲控制, 总线协议, I/O 控制, 数字光栅尺控制, 计数器, 主轴控制等功能, 这些控制功能是实现在驱动程序部分的, 为了降低 ARM CPU 的占用率, 需要实现这些驱动从 ARM 处理器 Linux 操作系统到 FPGA MicroBlaze 软核处理器 μC/OS-II 操作系统的下载.

参考文献

1 费继友,周茉.基于+的嵌入式数控装置设计.制造技术与机床,2010,12(12):61–64.

2 季照平.基于单片机 ARM 嵌入式技术的数控系统的开发研究.轻工科技,2015,(11):47–48,68.

3 刘建康,付云忠.基于 ARM_FPGA 的嵌入式数控系统硬件设计[硕士学位论文].哈尔滨:哈尔滨工业大学,2013.

4 张开生,陈明,周子超.实时操作系统 μC/OS-II 的实现.电子科技,2013,26(10):36–39.

5 赵伟国,李文军,梁国伟.实时嵌入式操作系统-在上的移植.中国计量学院学报,2005,16(2):137–139.

6 陈果,冯静.系统及其消息队列详析.电子元器件应用,2011,3(3):38–42.

7 王帅,陈金鹰,邹振宇.基于的 μC/OS-II 移植.中国集成电路,2011,20(7):80–84.

8 邵贝贝等译.嵌入式实时操作系统 μC/OS-II.第 2 版.北京:北京航空航天大学出版社,2003.

9 孙丰祥,程玉伟,胡恩俊,等.基于软核的嵌入式最小系统.化工自动化及仪表,2013,8(8):36–39.

10 胡志东,韩晓明,马瑞.嵌入式技术在雷达数字接收机中的应用.信息通信,2016,3(3):63–65.

11 单超,王萍,朱爱民,等.基于 软核的嵌入式系统设计.单片机与嵌入式系统应用,2011,11(3):18–21.