重庆大学本科学生毕业设计（论文）

基于ARM的EPOS操作系统的移植



学 生：冯志敏

学 号：20121955

指导教师：洪明坚

专 业：软件工程专业

重庆大学软件学院

二O一六年六月

**Graduation Design (Thesis) of Chongqing University**

**Port of EPOS operating system to ARM**



**Undergraduate: Feng Zhimin**

**Instructor: Associated Prof.Hong Mingjian**

**Major: Software Engineering**

**School of Software Engineering**

**Chongqing University**

**June 2016**

# 摘 要

随着开源硬件在嵌入式领域的广泛使用，越来越多基于ARM体系结构的开源开发板产品也随之诞生，其中比较流行的一款开源硬件—Raspberry Pi(树莓派)，其体积仅有普通信用卡大小，搭载ARM架构处理器，运算性能与智能手机相仿。本课题研究的内容就是将EPOS操作系统移植到Raspberry Pi开发板上。

本课题是将X86体系结构下的EPOS操作系统移植到以Broadcom BCM2835 ARMv7 为处理器的开源硬件Raspberry Pi B+上，主要是将EPOS操作系统由X86系统架构修改成ARM体系架构，大致工作内容包括系统引导项(Bootloader)、内存管理单元(MMU)、中断处理、系统定时器、多线程的切换以及Raspberry Pi开发板上的UART驱动。

本课题实现的功能为：将EPOS操作系统编译后生成kernel.img文件，将kernel.img文件拷贝到FAT文件格式的SD卡中，将SD卡插入Raspberry Pi B+卡槽内，开启电源后，通过UART串口向PC机发送一条消息，同时初始化多任务。

**关键词：**嵌入式， Bootloader，UART，MMU，Raspberry Pi B+

# ABSTRACT

With the widely use of the open source hardware in the embedded field, more and more based on ARM architecture development board products emerge, which more popular a open source hardware - Raspberry Pi (RPi), the volume of only credit card size, equipped with ARM processor architecture, the computational performance and smart phones similar. The content of this research is to transplant the EPOS operating system to the Pi Raspberry development board.

In this paper, the x86 architecture of EPOS operating system transplantation by Broadcom BCM2835 armv7 processors of open source hardware raspberry Pi B +, mainly is the EPOS by x86 system structure modified to arm architecture, roughly modified content includes system boot item (bootloader), memory management unit (MMU), interrupt processing, system timer, multi thread switching and Pi development board UART driver.

In this paper, the realization of the function is: EPOS operating system compiler generated after kernel.img file, kernel.img files are copied to the SD card fat file format in, SD card into the raspberry Pi B + card slot, after power is turned on, through the UART serial port to the PC sends a message. Meanwhile, the initial of multi task.

**Key words：**Embedded, Bootloader, UART, MMU, Raspberry Pi B+

**目 录**

[摘 要 I](#_Toc452907319)

[ABSTRACT II](#_Toc452907320)

[1 绪论 1](#_Toc452907321)

[1.1 选题背景及意义 1](#_Toc452907322)

[1.2 开发板介绍 1](#_Toc452907323)

[1.3 开发环境搭建 2](#_Toc452907324)

[2 相关技术介绍 4](#_Toc452907325)

[2.1 ARM体系结构介绍 4](#_Toc452907326)

[2.2 Bootloader介绍 6](#_Toc452907327)

[2.2.1 Raspberry Pi启动流程介绍 6](#_Toc452907328)

[2.2.2 操作系统初始化以及加载kernel 7](#_Toc452907329)

[2.3 ARM虚拟内存 7](#_Toc452907330)

[2.3.1 MMU工作原理 7](#_Toc452907331)

[2.3.2 CP15协处理器 12](#_Toc452907332)

[2.3.3 恒等映射 12](#_Toc452907333)

[2.3.4 page fault处理 13](#_Toc452907334)

[2.4 ARM中断处理 14](#_Toc452907335)

[2.4.1 中断机制 14](#_Toc452907336)

[2.4.2 中断向量表 19](#_Toc452907337)

[2.4.3 Raspberry Pi 定时器 19](#_Toc452907338)

[2.5 多任务处理 20](#_Toc452907339)

[2.5.1 多任务工作原理 20](#_Toc452907340)

[2.5.2 多任务切换 21](#_Toc452907341)

[3 系统设计及实现 23](#_Toc452907342)

[3.1 系统引导及其初始化 23](#_Toc452907343)

[3.2 UART串口驱动设计 24](#_Toc452907344)

[3.3 虚拟内存 26](#_Toc452907345)

[3.3.1 内存映射 26](#_Toc452907346)

[3.3.2 激活MMU 27](#_Toc452907347)

[3.3.3 EPOS恒等映射 28](#_Toc452907348)

[3.4 中断处理 29](#_Toc452907349)

[3.4.1 初始化中断向量表 29](#_Toc452907350)

[3.4.2 IRQ中断开关 30](#_Toc452907351)

[3.4.3 中断现场保护 31](#_Toc452907352)

[3.4.4 定时器中断处理 32](#_Toc452907353)

[3.5 多任务 32](#_Toc452907354)

[3.5.1 创建多任务 33](#_Toc452907355)

[3.5.2 任务切换 33](#_Toc452907356)

[4 系统展示 35](#_Toc452907357)

[4.1 源码编译操作 35](#_Toc452907358)

[4.2 系统运行结果 35](#_Toc452907359)

[5 结论及展望 39](#_Toc452907360)

[5.1 结论 39](#_Toc452907361)

[5.2 展望 39](#_Toc452907362)

[致谢 40](#_Toc452907363)

[参考文献 41](#_Toc452907364)

# 绪论

## 选题背景及意义

对于计算机专业的学生来说，操作系统是一门非常重要的主干课程，是计算机专业学生必须掌握的一门专业基础课程，对学生以后的在职业道路上有很大的帮助。在操作系统的课程学习中，不仅需要懂得一个操作系统的基本工作原理，还需自己亲自去做实验，学习操作系统的实现方法，这样有助于学生更加深刻的理解操作系统原理。本课题选择将一个轻量级的操作系统EPOS移植到开源硬件Raspberry Pi上运行，一方面可以帮助学生理解操作系统的工作原理，同时学生可以直接将操作系统的理论知识转化为代码，并且可以在Raspberry Pi上验证自己实验的正确性。另一方面，Raspberry Pi在嵌入式领域非常受欢迎，我们将EPOS移植到Raspberry Pi上，在嵌入式领域上我们可以使用EPOS来完成部分项目的需求。

## 1.2 开发板介绍

树莓派(Raspberry Pi)是由英国的树莓派基金会开发的一款信用卡大小的卡片式电脑，其官方提供的系统是基于Linux的Raspbian操作系统。树莓派简称RPi是为计算机编程教育而设计，目的是提升学校在计算机编程上的教学质量，让计算机编程变得有趣。目前树莓派以及有多个版本，本课题所使用的是Raspberry Pi B+开发板，其采用Broadcom BCM2835 700MHz ARM1176JZFS 处理器，带FPU和VideoCore IV双核GPU。

Raspberry Pi开发板与普通ARM开发板的略有不同，常见的ARM开发板是上电后直接初始化CPU，树莓派不同的是上电后先驱动GPU，当GPU初始化成功后才会跳转到物理内存为0x8000处去驱动CPU，接下来是对ARM模式设置并且初始化ARM各种模式下的堆栈，设置栈指针等等操作。

RPi B+提供4个USB2.0接口、512MB SDRAM、 10/100 BaseT RJ45以太网插座、 HDMI 1.3和1.4视频/音频插座、MicroSD 卡插座以及40个外接引脚。下面是Raspberry Pi B+的外观图：见图1.1(树莓派B+外观图)

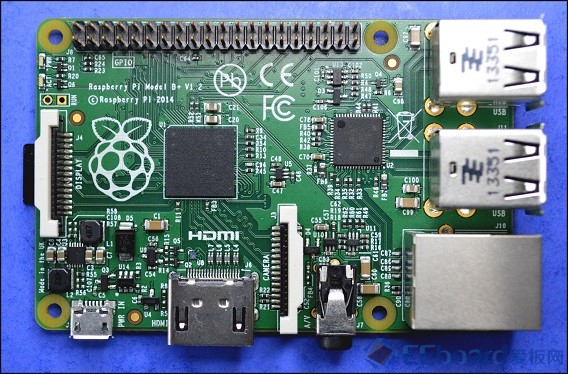


图1.1 树莓派B+外观图

下面是Raspberry Pi B+的40个外接引脚功能图：见图1.2(引脚图)，本课题中使用的UART的引脚是GPIO14和GPIO15。

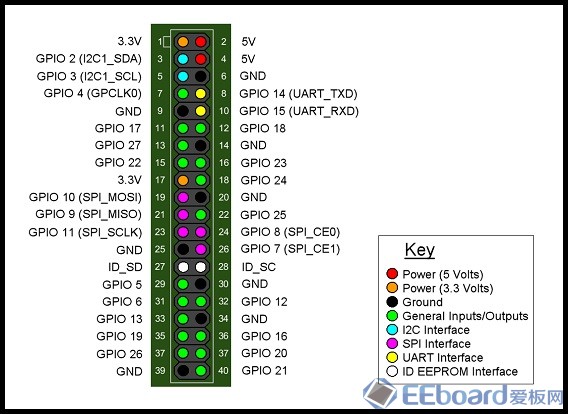


图 1.2 RPi B+引脚图

## 1.3 开发环境搭建

本课题使用的开发平台是Ubuntu 14.04 64位操作系统，采用GNU 推出的的ARM交叉编译工具arm-none-eabi-gcc，可用于交叉编译ARM MCU芯片，如ARM7、ARM9、Cortex-M/R芯片程序，采用GNU make来构建和管理源码。

EPOS的交叉编译环境如图1.3所示，首先通过make将源码编译生成kernel.img文件，然后将kernel.img文件拷贝到FAT32格式的SD卡中，将SD卡插入树莓派卡槽，用串口线将PC与树莓派连接起来(注意：我们使用串口转USB线，将USB接口插入PC机，将串口线与树莓派的外接引脚相连)，开启电源，此时我们就可以在PC机通过minicom软件将串口的数据打印出来。

Ubuntu 主机

RPi B+开发板

串口

图 1.3 交叉编译环境

# 相关技术介绍

## ARM体系结构介绍

计算机指令集可分为精简指令集和复杂指令集两种。精简指令集计算机(RISC:Reduced Instruction Set Computing RISC) 是一种执行较少类型计算机指令的微处理器，复杂指令集计算机(Complex Instruction Set Computing ，CISC)以微程序技术为基础的、具有较复杂指令系统的计算机。X86是采用的CISC指令集，而ARM采用RISC指令集。

目前EPOS操作系统是运行在X86体系结构下的，因为X86结构和ARM结构完全不同，所以本课题主要完成的任务是将EPOS所有与X86相关的底层代码都要修改为符合ARM的要求，主要包括系统引导项、MMU、中断处理以及多任务的切换。

ARM体系的CPU有两种工作状态：

* + - 1. ARM状态：处理器执行32位的字对齐的ARM指令；
      2. Thumb状态：处理器执行16位的、半字对齐的Thumb指令；

在程序运行的过程中，可以在两种状态之间进行相应的转换。处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。CPU上电后默认处于ARM状态。

ARM体系的CPU有以下7种工作模式，分别是：

1. 用户模式(USR)：用于正常执行程序；
2. 快速中断模式(FIQ)：用于高速数据传输模式；
3. 外部中断模式(IRQ)：用于通常的中断处理模式；
4. 管理模式(SVC)：操作系统使用的保护模式；
5. 数据访问终止模式(ABT)：当数据或指令预取终止时进入该模式，可用于虚拟存储以及存储保护；
6. 系统模式(SYS)：运行具有特权的操作系统任务；
7. 未定义指令中止模式(UND)：当未定义的指令执行时进入该模式；

在以上7中模式中，除了用户模式(USR)外，其它6中模式都属于特权模式，在这6中特权模式下，除了系统模式(SYS)外，其它5中模式都属于异常模式，每种异常模式都拥有私有的堆栈指针寄存器R13、返回地址寄存器(R14)以及模式备份寄存器SPSR(Saved Program Status Register)，程序只有在对于的异常模式下才有权访问该模式下的3个私有寄存器。一般我们只用到用户模式(USR)和系统模式(SYS)两种，其它模式是在处理异常的时候用到。

ARM有两种方法进行工作模式切换：

1. 被动切换：在arm运行的时候产生一些异常或者中断来自动进行模式切换。
2. 主动切换：通过软件改变，即软件设置寄存器来经行arm的模式切换，

应为arm的工作模式都是可以通过相应寄存器的赋值来切换的。

ARM体系的CPU有以下7种异常状态，见表 2.1 (ARM异常状态)

所谓的异常是指终止了正常执行的程序而去执行一些特殊的工作，例如：芯片复位、取指失败等等。通常说的中断也是一种异常，这里所说的中断包括两类，一类是由外部硬件产生的外部中断，另一类是由芯片内部硬件产生的内部中断。由中断产生的异常和其它异常，处理方法是一致的。

表 2.1 ARM异常状态

|  |  |  |
| --- | --- | --- |
| **异常类型** | **所属模式** | **说明** |
| 芯片复位 | SVC | 由芯片复位触发 |
| 未定义指令 | UND | 指令不能被识别 |
| 软中断 | SVC | 调用SWI指令触发 |
| 预取异常 | ABT | 没有权限访问寄存器 |
| 数据异常 | ABT | 没有权限访问寄存器 |
| 中断 | IRQ | 由硬件进行中断请求触发 |
| 快速中断 | FIQ | 由硬件进行快速中断请求触发 |

ARM一共有37个通用寄存器，但是在程序中我们不可以同时使用这37个通用寄存器中，因为这些寄存器是按照ARM模式进行分组的，每种模式下都有属于自己的私有寄存器，程序只有在对应模式下才能够使用对应模式的私有寄存器。在ARM体系下只有一组寄存器是共有寄存器，这组共有寄存器是在所有模式下访问都是同一个寄存器，ARM系统中的R0~R7、R15以及CPSR(Current Program Status Register)各寄存器只有唯一一个，这些寄存器就是ARM系统中唯一的一组共有寄存器，例如：在中断模式和用户用户下访问R15寄存器，其实是在访问同一个寄存器。具体寄存器分组见表 2.2 ARM寄存器：

表 2.2 ARM寄存器

ABT

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_abt |
| R14\_abt |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_abt |

UND

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_und |
| R14\_und |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_und |

IRQ

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_irq |
| R14\_irq |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_irq |

SVC

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_svc |
| R14\_svc |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_svc |

FIQ

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8\_fiq |
| R9\_fiq |
| R10\_fiq |
| R11\_fiq |
| R12\_fiq |
| R13\_fiq |
| R14\_fiq |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_fiq |

USR /SYS

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13(SP) |
| R14(LR) |
| R15(PC) |

|  |
| --- |
| CPSR |
|  |

注：R13寄存器别名为SP，该寄存器是栈指针寄存器。R14寄存器别名为LR，该寄存器是链接寄存器。R15寄存器别名为PC，该寄存器是程序计数器。CPSR寄存器全名是：Current Program Status Register，当前程序状态寄存器。除了系统模式和用户模式外，其它的模式都有一个特殊的私有寄存器，称为SPSR(Saved Program Status Register，保存程序状态寄存器)，当异常产生时，CPSR会自动将当前的值保存到SPSR中，从异常返回退出时则可以从SPSR中将CPSR恢复。由于用户模式和系统模式不属于异常模式，它们没有SPSR[1]。

## Bootloader介绍

### Raspberry Pi启动流程介绍

树莓派的启动方式和普通的arm不一样。主要是由树莓派的硬件决定的。树莓派芯片上有一个arm11的CPU，还有一个GPU用来处理图形。树莓派的启动是GPU和CPU协同工作的：

首先：上电后首先运行的是GPU，运行GPU中的一个核心功能就是用FAT32挂载SD卡。

成功挂载SD卡后，首先加载放在SD卡boot分区里面的bootcode.bin文件到内存中运行，这个就是我们通常说的bootloader， 然后再调用start.elf到内存，然后再启动GPU，GPU运行start.elf的程序，开始一系列的配置，最后调用kernel.img即系统内核到内存[2]。

最后，GPU将控制权交给CPU，也就是完成系统内核的启动。

上面提到的bootcode.bin、start.elf是从树莓派官网下载，这两个文件直接放到FAT格式的SD卡中即可。kernel.img文件则是由我们的系统源码编译后生成的镜像文件。CPU开始运行的物理地址不是在0x0处，而是CPU从物理地址的0x8000处开始运行。

### 操作系统初始化以及加载kernel

EPOS中，首先是定义7种异常的服务函数，也就是定义中断向量表，然后EPOS对7种模式进行初始化，首先通过设置CPSR寄存器，将当前模式修改成对应模式，为该模式进行设置栈指针。最后切换当前模式为SVC模式，使用bl指令跳转到C语言函数进行其他操作，到此完成了EPOS的启动。

## ARM虚拟内存

### MMU工作原理

MMU是Memory Management Unit的缩写，MMU负责的是将虚拟地址到物理地址的转换，提供硬件机制的内存访问授权。图2.1 展示了MMU的作用，图中是将虚拟内存0x00008000映射到物理内存0xC0008000处。

虚拟内存 物理内存

图 2.1 MMU作用

0x00008000

0xC0008000

MMU

页表(page table)是MMU中一个非常重要的概念，页表是存储在内存中的一张表，在表中每条数据都是记录着将虚拟内存转化为物理内存的关键信息，MMU就是通过查询该表中的数据，实现了将虚拟地址转化为物理地址。也就是说，MMU每次工作的时候都要去查这张表，从表中找出与虚拟地址相对应的物理地址，然后再进行数据存取操作。页表的作用如图 2.2所示：

条目被称为页表项(page table entry)，一条页表项负责记录一段虚拟地址空间到物理地址空间的映射，这里所说的虚拟地址空间被称为页，一个页对应页表的中的一项。页的大小是可以选择的，通常ARM支持页大小为1K、4K、16K和1M，分别叫做微页、小页、大页和段页。本课题所采用的小页映射，所以接下来我们主要讨论小页映射[1]。

0x00008000

0xC0000000

MMU

TLB

页表

虚拟内存

物理内存

图 2.2 页表作用

当页的大小为4K的时候，MMU采用二级查表的方法，第一步由虚拟地址的高12位索引出第一张表的字段，然后根据虚拟地址的12~19位和这些字段索引出第二张表的字段，最后根据虚拟地址的低12位和第二张表的内容确定出该虚拟地址对应的物理地址，这里提到的第一张表称为页目录，第二张表称为小页表。采用二级页表查询物理地址的目的是减少页表自身占用的内存空间，但缺点是进一步减少了内存的寻址效率。

下面介绍页目录表结构和小页表结构以及虚拟地址结构图，图2.3为页目录结构，图2.5为小页表项结构，图2.7位虚拟地址结构图，页目录的高22位存放的是小页表的基地址，而小页表项的高20位存放的是物理地址的基地址。虚拟地址的高12位(20~31位)存放的是页目录的偏移量，第12~19位存放的是小页表项的偏移量，低12位(0~11位)存放的是物理地址的偏移量。

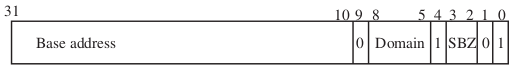


图 2.3 页目录结构

在页目录中，1~0位被设置为01，3~2位是SBZ(should be zero)被设置为0，第4位被设置为1，第8~5位表示该页目录项所映射的内存所属那个域，在ARM中最多运行定义16个域，并且只能运行每个页目录项对应的地址空间属于16个域中的一个，从而我们可以通过改变域来实现批量控制整块地址空间的访问权限。在ARM中这16个域的访问权限是由CP15协处理器中的C3寄存器来控制，

图2.4为C3寄存器结构：



图 2.4 C3寄存器结构

图 2.4中，32位的C3寄存器每两位代表一个域的权限，一共有16个域，每个域有4种权限，这四种权限的取指分别是0b00、0b01、0b10、0b11，表2.3显示不同值代表的权限。

表2.3 域权限值

|  |  |  |
| --- | --- | --- |
| **访问** | **域权限值** | **描述** |
| 管理者 | 0b11 | 不进行权限检查，允许任何访问 |
| 保留 | 0b10 | 结果不可预料 |
| 用户 | 0b01 | 使用段描述符、页描述符中AP进行权限检查 |
| 禁止访问 | 0b00 | 将产生域错误 |

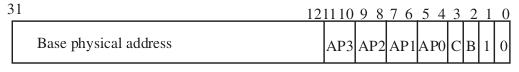


图 2.5 小页表项结构

在小页表项结构中，第1~0位被设置为10，第3位和第2位分别是用来设置cache和write buffer，当将这两位设置为1时，则表示使用cache和write buffer，反而则不使用。第11~4位是分别是该小页表的4个AP(access permision)位，每个AP（AP0-3）控制一个小页（4KB）的1/4内存的访问权限，AP0、AP1、AP2、AP3结合CP15寄存器协处理器中的C1寄存器的R/S位来进行权限的控制，表2.4位AP与R/S位结合的控制权限：

表2.4 AP与R/S位结合的控制权限

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| AP | S位 | R位 | 特权模式 | 用户模式 | 描述 |
| 0b11 | 忽略 | 忽略 | 可读写 | 可读写 | 所有模式进行可操作 |
| 0b10 | 忽略 | 忽略 | 可读写 | 只读 | 用户模式是进行写操作产生异常 |
| 0b01 | 忽略 | 忽略 | 可读写 | 不可访问 | 在特权模式下可以读写 |
| 0b00 | 0 | 0 | 不可访问 | 不可访问 | 任何访问都会产生异常 |
| 0b00 | 0 | 1 | 只读 | 只读 | 任何写操作都产生异常 |
| 0b00 | 1 | 0 | 只读 | 不可访问 | 在特权模式下可以进行读 |
| 0b00 | 1 | 1 | 不可预料 | 不可预料 | 不可预料结构 |

图2.6为CP15协处理器的C1寄存器结构：

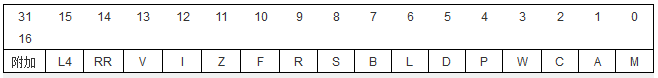


图 2.6 C1寄存器结构

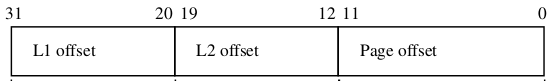


图 2.7 虚拟地址结构

接下来我们用实际的案例来理解MMU工作原理见图2.8。

首先，我们要让MMU知道页目录在物理内存中的基地址，因为页目录是我们通过程序确定的，存储在什么位置自然是知道的，然后CPU需要寻址的时候，MMU就可以自动的将虚拟地址映射为物理地址，具体步骤如下：

* 1. MMU取出虚拟地址的高12位作为页目录的偏移量，结合页目录的基地址取出对应的页目录项，具体操作是将这12位取出，然后左移两位和页目录基地址相加操作，这样就得到页目录项的地址了。例如：页目录的基地址是0x20000000，假如虚拟地址是0x00101001，则高12位是0x001，那么左移两位是0x004，那么得到的页目录项的地址是0x20000000+0x004=0x20000004，接下来从该地址读取的数据是该虚拟地址对应的小页表的基地址。
  2. 找到虚拟地址对应的小页表的基地址后，MMU取出虚拟地址的12~19位，结合小页表的基地址取出对应的小页表项，具体操作是将这8位取出，然后左移两位和小页表的基地址进行相加操作，这样就得到了小页表项的地址了。例如：由1取出的数据是0x30000001，那么我们知道小页表的基地址是0x30000000，此时虚拟地址的中间8位是0x01左移两位后是0x04，那么小页表项的地址是0x30000000+0x04=0x30000004，从该地址读取的数据是该虚拟地址对应的物理地址的基地址。
  3. 从小页表项中读取的32位数据就是包含物理地址的基地址，接下来，将虚拟地址的低12位与物理地址的基地址进行相加操作，最后得到的就是该虚拟地址对应的物理地址。例如：由2中读取的数据是0xC0000001，虚拟地址的低12位是0x001，那么得到的物理地址是0xC0000000+0x001=0xC0000001，那么虚拟地址0x00101001经过MMU查找后对应的物理地址是0xC0000001。

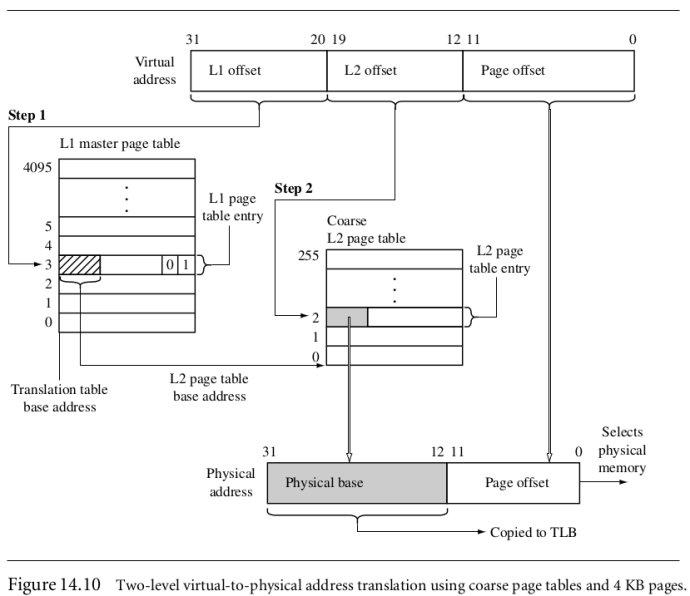


图 2.8 MMU工作原理

注意：虚拟地址的高12位表示页目录项的偏移量，所以一共有2^12=4096项页目录项，每项占4个字节，所以页目录总共占内存16K，其中每一项对应着一张小页表的基地址，虚拟地址的中间8位表示小页表项的偏移量，所以一张小页表包括2^8=256项小页表项，一项占4个字节，所以一张小页表占1K大小。

### CP15协处理器

CP15是ARM的一个协处理器，MMU就是通过配置CP15来激活的，ARM支持16个协处理器，当然并不是所有的ARM体系结构都支持。CP15是负责系统控制的，系统中的cache、write buffer、MMU等等都是通过CP15协处理器来控制的。协处理器的访问不能使用常规的指令。

ARM寄存器到协处理器寄存器的数据传送指令如下：

MCR{<cond>} p15，0，<Rd>,<CRn>,<CRm>{,<opcode\_2>}

协处理器寄存器到ARM寄存器的数据传送指令如下：

MRC{<cond>} p15，0，<Rd>,<CRn>,<CRm>{,<opcode\_2>}

其中，<cond>为指令执行的条件码。当<cond>忽略时指令为无条件执行。

<Rd>作为源寄存器的ARM寄存器，其值将被传送到协处理器寄存器中。

<CRn>作为目标寄存器的协处理器寄存器，其编号可能是C0，C1，…，C15。

### 恒等映射

目前主流操作系统中将虚拟内存空间0~3G划分给用户模式使用，将3G~4G空间划分给内核使用。由于ARM启动的时候是从地址空间0x0处开始执行，要想我们将kernel定位到3G以上，那么我们采用恒等映射来完成，图2.9显示了一个操作系统的恒等映射。

Kernel

MMU

Kernel

3G

Kernel

4G

0

物理内存

虚拟内存

图 2.9 恒等映射

具体操作如下：

首先，我们将kernel的物理地址映射两份，一份映射是将虚拟地址和物理地址一致，另一份是将kernel的物理地址加上3G进行映射。

上电后，CPU从物理地址0x8000处开始执行程序，当激活MMU后，CPU运行在虚拟地址空间上，此时的PC寄存器任然指向3G以下的虚拟空间运行，此时，我们通过修改PC寄存器将CPU跳转到3G以上去运行。

最后，我们将3G以下的kernel虚拟地址空间清除掉，这样就保证虚拟地址控制3G以下留给用户使用，3G以上供内核使用。

### page fault处理

产生Page fault的原因是程序访问一个虚拟地址的时候，经过MMU转化后没有找到对应的物理地址，则此时会产生一个Page fault错误，简单的说就是由于缺页找到Page fault错误。

产生Page fault的方式有两种，一种是通过预取指令异常产生，当程序执行在某一地址加载一个指令的时候，无法获取该指令，则此时向系统发出Page fault错误。另一种是通过数据异常产生，当程序向一个地址写或者读取数据的时候，如果没有权限进行写或者读操作，那么此时向操作系统发出一个Page fault错误。

处理Page fault的大致方法：首先获取触发Page fault的地址，然后获取错误代码，同时保存上下文，进入对应的异常处理函数后，通过调用do\_page\_fault(由操作系统实现处理page fault的函数)函数来进行操作。

ARM获取Page fault地址和错误代码是通过访问IFAR、DFAR、IFSR和DFSR寄存器来获取。IFAR全称Instruction Fault Address Register，该寄存器中存放的是指令预取中止异常时的地址，IFSR(Instruction Fault Status Register)则存放着对应的错误代码值。DFAR全称Data Fault Address Register，该寄存器中存放的是数据中止异常时的地址，DFSR(Data Fault Status Register)则存放着对应的错误代码值。下面是对以上四个寄存器的操作：

MRC p15, 0, <Rt>, c6, c0, 2 ; 读取 IFAR 到 Rt

MCR p15, 0, <Rt>, c6, c0, 2 ; 将 Rt 写入 IFAR

MRC p15, 0, <Rt>, c5, c0, 1 ; 读取 IFSR 到 Rt

MCR p15, 0, <Rt>, c5, c0, 1 ; 将 Rt 写入 IFSR

MRC p15, 0, <Rt>, c6, c0, 0 ; 读取 DFAR 到 Rt

MCR p15, 0, <Rt>, c6, c0, 0 ; 将 Rt 写入 DFAR

MRC p15, 0, <Rt>, c5, c0, 0 ; 读取 DFSR 到 Rt

MCR p15, 0, <Rt>, c5, c0, 0 ; 将 Rt 写入 DFSR

## ARM中断处理

### 中断机制

中断处理是嵌入式系统的关键部分，通过有效利用中断管理与外部系统的相互作用可以显着提高系统的效率和使用处理资源的。ARM中断类型大致可以分为以下几种：

* + - 1. 非嵌套中断处理
      2. 嵌套中断处理
      3. 可重入嵌套中断处理
      4. 优先级中断处理

嵌入式系统必须处理实时的事件，比如键盘被按下、视屏同步输出和处理通讯设备数据的接受和发送，这些事件必须及时的处理，意思就是在特定的时间段内处理该事件，例如：在嵌入式系统中，当键盘的键被按下后，那么系统必须快速的响应，以便于用户可以在屏幕上看到一个字符，并且没有使用户感到有明显的延时。如果出现了过度的延时，那么用户就会认为系统没有响应。

一个嵌入式系统必须处理很多的事件，为了便于说明，事件可分为两种类型，计划和非计划。计划型事件例如有按键被按下、时钟定期的产生中断和软件中断。非计划型例如有数据的异常、指令异常和未定义指令异常。从物理角度看，当ARM处理器上的IRQ引脚被设置为高的时候，此时中断产生。中断源的时间可以跟随处理器的时钟或不。当中断源跟随着处理器时钟叫做同步中断源，反而叫做异步中断源。(见图2.10 同步和异步中断源)[3]

注：在ARM内部，所有的中断都是同步的。

异步中断源

同步中断源

图 2.10 同步和异步中断源

有两种方法触发一个中断(边沿和电平)，这两种都依赖于电压的变化(见图 2.11)，这种改变可以在上升沿也可以在电压平衡后。

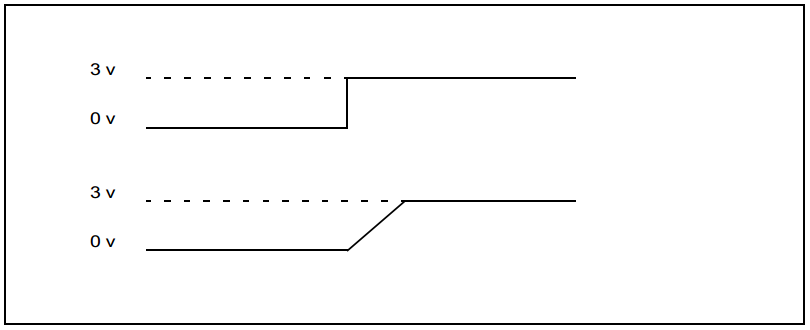


图2.11 中断触发(上-电平 下-上升沿)

从软件的角度来评论以上两种方法优点和缺点：

① 上升沿：当信号变为高的时候，中断将会被触发，但是不会被重新触发除非信号再次变高或者变低。

② 电平：当信号为高的时候，中断持续被触发，所以将会不断的进入中断处理函数除非信号被清除。即使处理器没有被启动，中断也会发生。

注：大多数ARM微型控制器是用过软件配置来触发的。

图2.12 显示了一个有3个中断源(按钮、串行设备和时钟)的简单嵌入式系统。当按钮被按下的时候，按钮信号将会被触发。这些信号将会被送到中断控制器，如果中断控制器屏蔽了这个信号，那么它将不会被通过处理器执行。一旦一个中断出现软件处理函数将会通过读取中断控制寄存器去判断那个中断，这个中断将会被中断服务函数处理(ISR)。







图2.12 简单中断系统例子

中断处理函数是当中断发生时被执行的程序，ISR是一个执行特殊中断的程序。例如按键的ISR是处理那个键被按下，同时将一个字符放入到键盘中断中。

EPOS中使用的中断时非嵌套中断，所以接下来我们只讨论非嵌套中断处理方法，非嵌套中断时最简单的中断方式，非嵌套中断言外之意就是一旦进入某一中断后，其他任何中断都不可以在此期间产生，除非完成了当前中断后，才可以进行其他中断。所以非嵌套中断不适合于拥有优先级的复杂嵌入式系统。

当IRQ中断产生的时候，ARM处理器首先会屏蔽进一步的IRQ中断产生，然后处理器设置PC指向一个正确的中断向量表并且指向对应的指令，这些指令就是修改PC寄存器使其指向中断处理函数。一旦进入中断处理程序后，首先第一步必须先保存当前上下文内容，目的是为了当处理完中断后可以正确的返回中断前的程序继续执行。图 2.13 展示了一个简单的非嵌套中断程序。

禁止中断

开启中断

保存上下文

恢复上下文

中断程序

中断服务程序

中断

1

6

2

5

3

4

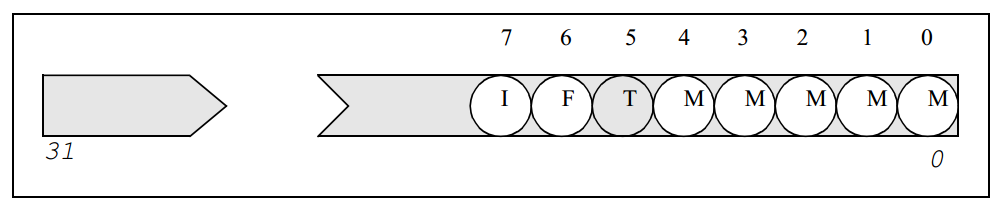
返回

图 2.13 非嵌套中断程序

图 2.13每个阶段的详细描述如下：

* 1. 外部中断源设置中断标记，处理器屏蔽进一步的中断并且在中断向量表中查找中断入口。
  2. 进入中断处理程序后，首先要保存当前程序的上下文。
  3. 在中断处理程序中标记出中断源并且跳转到中断服务函数开始执行。
  4. 进入中断服务函数进行服务中断。
  5. 恢复上下文。
  6. 开启中断，同时返回中断的程序继续执行。

上电后ARM处理将会禁止所有的中断，除非它们通过初始化代码进行开启，中断的开启或者禁止是通过设置程序状态寄存器(PSR或者CPSR C代表当前)来实现。PSR的前4位是保留条件执行的标志，在特权模式下，程序可以对CPSR寄存器进行读写操作，但是在非特权模式下程序仅仅只能读取CPSR寄存器。当中断或者异常产生的后，处理器将会进入相应的中断或者异常模式。



M = 处理器模式位

T = Thumb位(1 = Thumb状态 0 = ARM状态)

I = IRQ (1 = 禁止 0 = 开启)

F = FIQ (1 = 禁止 0 = 开启)

图 2.14 PSR寄存器

从图2.14可以看到，低6位和7位是开启或者禁止中断的屏蔽位。ARM处理器有两个外部中断输入都可以被认为是通用的中断。第一个叫做中断请求(IRQ)第二个叫做快速中断请求(FIQ)。

注意：IRQ中断和FIQ中断的主要区别就是FIQ通常用于快速响应的中断处理，因为FIQ产生时，只会保存部分寄存器，而IRQ则会保存所有寄存器，所以FIQ中断比IRQ中断快。

下面是介绍IRQ和FIQ中断发生时候，PSR寄存器控制位的变化情况。详细见图2.15 IRQ产生的状态机和图2.16 FIQ产生的状态机。

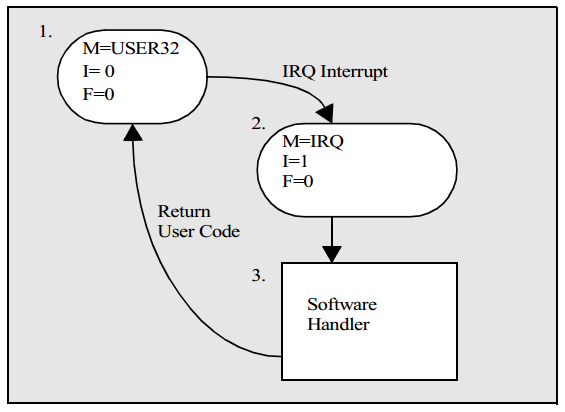


图 2.15 IRQ产生的状态机

图2.15显示了当IRQ产生的时候状态的变化，注对于这个例子状态1的处理器是出于用户模式。在CPSR中的IRQ位(I位)设置为0运行IRQ中断。当IRQ产生的时候，处理器将会自动设置I位为1，屏蔽进一步的IRQ中断产生，见状态2。F位任然保存为0，那么将运行FIQ可以产生，FIQ是比IRQ有更高的优先级中断，并且它不会被屏蔽。当模式设置为IRQ模式的时候，那么先前的模式，对于该例子中是用户模式CPSR将会被自动拷贝到IRQ的SPSR寄存器中。软件中断处理是在状态3。

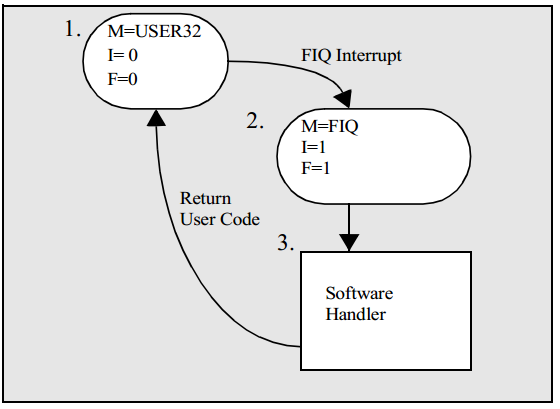


图 2.16 FIQ产生的状态机

图2.16显示了FIQ产生的状态变化图。处理器经过与IRQ相同的步骤进行处理，不过不仅要屏蔽进一步的IRQ中断位(I位)，同时也会屏蔽FIQ中断位(F位)，这就意味着当进入软件处理程序(状态3)时，I位和F位都会被设置为1。在FIQ模式下没必要保存r8和r12。这就意味着这些寄存器可以被用于存储临时数据，例如缓冲区指针或者计数器。

### 中断向量表

当中断或异常发生的时候，CPU自动将PC指向一个特定的地址，这个地址就是中断向量表。中断向量表的开始地址是0x00000000(ARMx20处理器可以选择定为向量表的地址为0xffff0000)。中断向量表包含了一系列的ARM指令操作PC寄存器（即B，MOV，LDR）, 这些指令会导致CPU跳转到一个特定的位置，可以处理特定的异常或中断。FIQ向量应该避免使用B或者LDR指令，因为它在向量表的末尾，这意味着，FIQ处理程序可以在FIQ向量位置开始。FIQ可以节约处理器通过不强制刷新管道当PC指令被修改时。表 2.5显示了中断向量表和处理器模式的关系图。

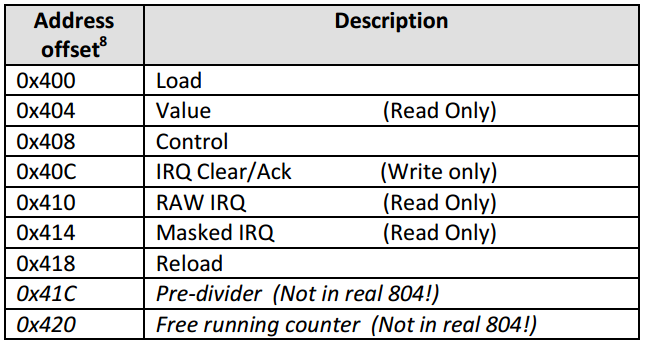
表 2.5 中断向量表和处理器模式的关系

|  |  |  |
| --- | --- | --- |
| **中断/异常/复位** | **模式** | **地址** |
| 复位 | SVC | 0x00000000 |
| 未定义指令 | UND | 0x00000004 |
| 软件中断(SWI) | SVC | 0x00000008 |
| 预取异常 | ABT | 0x0000000C |
| 数据异常 | ABT | 0x00000010 |
| 保留 |  | 0x00000014 |
| IRQ中断 | IRQ | 0x00000018 |
| FIQ中断 | FIQ | 0x0000001C |

### Raspberry Pi 定时器

树莓派B+提供一个硬件的定时器(Timer)，其物理地址是0x2000B000，该定时器包括以下控制寄存器，见表2.6 RPi Timer。

表2.6 RPi Timer



以上寄存器的详细功能介绍请参考BCM2835-ARM-Peripherals手册，本文只介绍EPOS中使用的寄存器，在EPOS中，我们使用了Load寄存器，该寄存器的功能是存放一个32位的数字，当定时器开始运行的时候，每隔一定时间就会将Load中的值自动减一，知道该数据减为0的时候，此时触发一个IRQ中断，然后将Reload寄存器的值拷贝到Load继续进行减，Control寄存器设置定时器的一些参数，具体设置参考图2.17 (Timer Control)。IRQ Clear/Ack寄存器是清除Timer中断，当触发Timer中断后，该寄存器自动设置为0x1，此时在中断服务函数中首先将该寄存器设置为0x0，这样做的目的是防止陷入Timer循环中断。Pre-divider寄存器是用于设置Timer Clock，Timer\_Clock = sys\_Clock/Pre-divider。

注意：树莓派B+的系统频率为250000000。

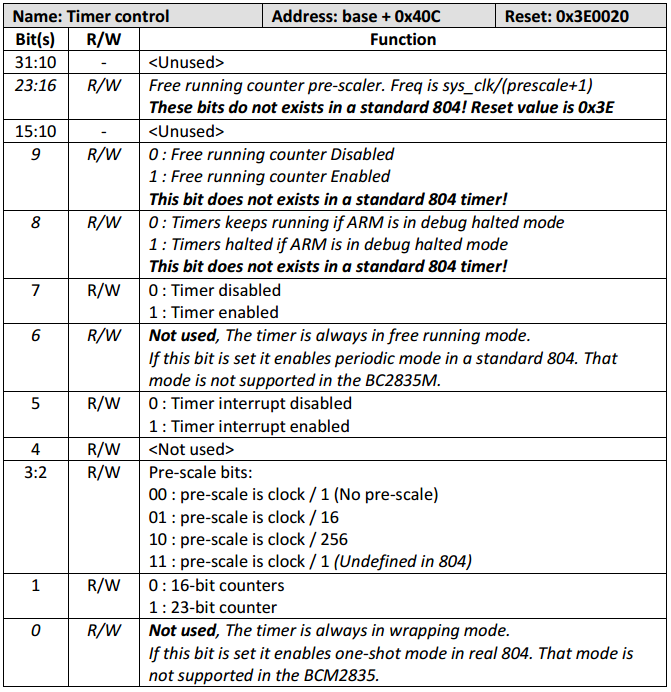


图 2.17 Timer Control

## 多任务处理

### 多任务工作原理

随着计算机硬件的发展，现在的主流计算机系统都是多任务系统，例如Window是单用户多任务操作系统，Unix/Linux是多用户多任务操作系统。简单的说，多任务系统是指在同一时刻计算机可以同时执行多个程序，例如，我们的电脑一边听歌一边玩游戏，听歌和玩游戏互不影响。

我们这里所说的多任务其实不是计算机同时既可以执行任务A也可以执行任务B，我们知道单核CPU在同一时刻只能执行一个程序，那么想要实现多任务，我们可以采用任务的切换来实现，也就是每一个任务不能长久使用CPU，我们为每个任务分时间段去执行，当正在执行的任务时间片段用完后就会被强行从CPU上移去，这时将下一个任务调出来进行继续运行，直到所有的任务运行结束。这样给用户的感觉是所有的任务都在运行。下面是多任务工作原理图，见图2.18。

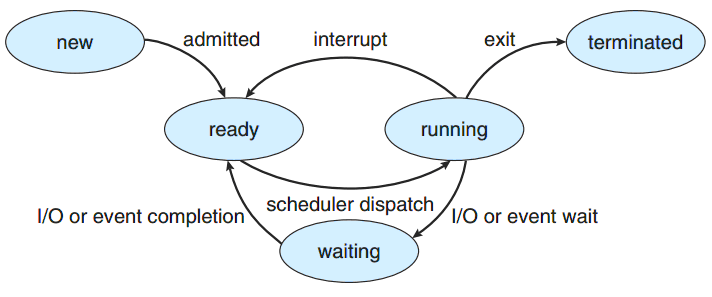


图2.18 多任务切换

多任务操作系统的优点就是可以同时运行多个任务，可以提高工作效率，不过多任务的缺点就是CPU的利用率不如单任务CPU利用率高，因为每个任务的切换需要CPU的执行时间。

### 多任务切换

多任务的切换在操作系统中非常重要，任务的切换主要是保存当前正在执行任务的内容，然后选择下一个将要被执行的任务，将其从就绪状态恢复到运行状态。

常见的调度算法有：先来先服务调度算法（FCFS）、短作业优先调度算法、高优先权优先调度算法和基于时间片的轮转调度算法等等。在EPOS中我们使用基于时间片的轮转调度算法，该算法基本原理是：系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间[5]。

# 系统设计及实现

## 系统引导及其初始化

由于RPi的CPU启动是从物理内存0x8000处开启执行，同时EPOS要求将kernel加载到3G以上运行，而3G一下是留给用户空间使用，所以我们使用的链接脚本中将kernel定位到0xC0000000 + 0x8000 = 0xC0008000处(3G=0xC0000000)，紧接着是EPOS的文本段、只读数据段、数据段和BSS段的初始化，这些段都是以4K为边界对齐，以上就是EPOS内核的内存布局，见图3.1 EPOS 内核内存布局图。

用户空间

0xC0008000

text

rodata

data

bss

Kernel end

虚拟内存

图 3.1 EPOS内核内存布局图

注意：链接脚本就是程序链接时参考的文件，其目的就是为了规定程序运行时的内存布局。

在entry.S汇编文件中首先定义ARM的7中模式对应CPSR寄存器的值和IRQ/FIQ中断的值(详见图3.2 ARM模式的CPSR值)，接下就是初始化中断向量表，最后跳转到系统启动代码执行，启动代码首先为EPOS使用的模式进行设置栈大小以及栈指针，最后将模式切换为SVC模式，使用bl指令进行跳转到C语言的入口函数进行下一步硬件的其它的初始化设置(详见图3.3 EPOS启动初始化)[4]。

.equ CPSR\_MODE\_USER, 0x10

.equ CPSR\_MODE\_FIQ, 0x11

.equ CPSR\_MODE\_IRQ, 0x12

.equ CPSR\_MODE\_SVR, 0x13

.equ CPSR\_MODE\_ABORT, 0x17

.equ CPSR\_MODE\_UNDEFINED, 0x1B

.equ CPSR\_MODE\_SYSTEM, 0x1F

.equ CPSR\_IRQ\_INHIBIT, 0x80

.equ CPSR\_FIQ\_INHIBIT, 0x40

图 3.2 ARM模式的CPSR值

*// We're going to use interrupt mode, so setup the interrupt mode*

mov r0, #(CPSR\_MODE\_IRQ | CPSR\_IRQ\_INHIBIT | CPSR\_FIQ\_INHIBIT )

msr cpsr\_c, r0

mov sp, #0x8000

mov r0, #(CPSR\_MODE\_ABORT | CPSR\_IRQ\_INHIBIT | CPSR\_FIQ\_INHIBIT )

msr cpsr\_c, r0

mov sp, #0x7000

mov r0, #(CPSR\_MODE\_SYSTEM | CPSR\_IRQ\_INHIBIT | CPSR\_FIQ\_INHIBIT )

msr cpsr\_c, r0

mov sp, #0x6000

mov r0, #(CPSR\_MODE\_UNDEFINED | CPSR\_IRQ\_INHIBIT | CPSR\_FIQ\_INHIBIT )

msr cpsr\_c, r0

mov sp, #0x5000

*// Switch back to supervisor mode (our application mode) and*

*// set the stack pointer towards the end of RAM. Remember that the*

*// stack works its way down memory, our heap will work it's way*

*// up memory toward the application stack.*

mov r0, #(CPSR\_MODE\_SVR | CPSR\_IRQ\_INHIBIT | CPSR\_FIQ\_INHIBIT )

msr cpsr\_c, r0

mov sp, #0x4000

sub sp, sp, #0x4

*/\* Now enter the C main function... \*/*

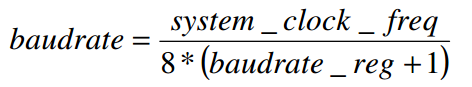
bl cstart

图3.3 EPOS启动初始化

## UART串口驱动设计

RPi B+提供两个UART设置，一个是mini UART另一个是PL011 UART，在EPOS中我们使用的是mini UART，所以我们主要讨论mini UART的驱动设计。

RPi B+中mini UART的物理地址是0x20215040，详细的mini UART介绍请参考BCM2835-ARM-Peripherals开发手册。我们这里只介绍AUX\_MU\_IO\_REG和AUX\_MU\_BAUD\_REG两个寄存器设置，AUX\_MU\_IO\_REG寄存器主要是用于读取数据和发送数据，AUX\_MU\_BAUD\_REG这个寄存器是用于设置mini UART的波特率，设置波特率公式见图 3.4 mini UART波特率设置公式。



baudrate：mini UART波特率

system\_clock\_freq：RPi系统时钟频率

baudrate\_reg：AUX\_MU\_BAUD\_REG寄存器的值

图3.4 mini UART波特率设置公式

在EPOS中我们设置mini UART的波特率是9600，图 3.5是EPOS中mini UART中初始化代码。

mmio\_write(IRQ\_DISABLE1, 1<<29);

unsigned int ra;

mmio\_write(AUX\_ENABLES,1);

mmio\_write(AUX\_MU\_IER\_REG,0);

mmio\_write(AUX\_MU\_CNTL\_REG,0);

mmio\_write(AUX\_MU\_LCR\_REG,3);

mmio\_write(AUX\_MU\_MCR\_REG,0);

mmio\_write(AUX\_MU\_IER\_REG,0x5); *//enable rx interrupts*

mmio\_write(AUX\_MU\_IIR\_REG,0xC6);

mmio\_write(AUX\_MU\_BAUD\_REG, 3254); *//(250000000/(8\*9600))-1*

ra=mmio\_read(GPFSEL1);

ra&=~(7<<12); *//gpio14*

ra|=2<<12; *//alt5*

ra&=~(7<<15); *//gpio15*

ra|=2<<15; *//alt5*

mmio\_write(GPFSEL1,ra);

mmio\_write(GPPUD,0);

mmio\_write(GPPUDCLK0,(1<<14)|(1<<15));

mmio\_write(GPPUDCLK0,0);

mmio\_write(AUX\_MU\_CNTL\_REG,3);

mmio\_write(IRQ\_ENABLE1, 1<<29);

图3.5 EPOS中mini UART中初始化代码

## 虚拟内存

### 内存映射

在EPOS中，首先执行启动代码，然后执行C语言函数，在C语言函数中，我们首先进行内存映射，EPOS初始化20张小页表，每张小页表可以映射1M物理内存，我们使用第17、18、19张小页表进行对I/O外设地址隐射，将I/O外设的物理地址[20000000, 20220000] 映射到虚拟地址[0xC1000000 0xC1220000]处，同时我们将kernel进行恒等映射。图3.6 显示了EPOS内核映射布局。

虚拟内存

物理内存

图 3.6 EPOS内核映射布局

20张小页表

Kernel

页目录

Kernel

Kernel

3G

Kernel

一下代码为EPOS初始化内存映射功能：首先分别20张小页表，其次对EPOS进行恒等映射，最后将I/O外设进行映射。详细代码见图 3.7 EPOS MMU初始化。

pgdir=(uint32\_t \*)physfree;

physfree += 1<<14;*//physfree += PAGE\_SIZE;*

pte = (uint32\_t \*)physfree;

memset(pgdir, 0, 1<<14);

*/\*分配20张小页表，并且填充页目录\*/*

**for**(i = 0; i < NR\_KERN\_PAGETABLE; i++)

{

pgdir[i] = pgdir[i+(KERNBASE>>PGDR\_SHIFT)] = (physfree)|PTE\_V;

memset((void \*)physfree, 0, (1<<10));

physfree += (1<<10);

}

*/\*设置恒等隐射，填充小页表 隐射物理地址为[0, &\_end] 虚拟地址[0, &\_end] [KERNBASE, R(&\_end)] \*/*

**for**(i = 0; i < (PAGE\_ROUNDUP(physfree)>>12); i++)

{

pte[i] = (i<<12)|PTE\_W|PTE\_V;

}

*/\*设置IO端口隐射，物理地址为[20000000, 20220000] 虚拟地址[0xC1000000 0xC1220000]从第17张小页表开始隐射IO设备地址 第17张小页表的第一项index是0x1000 \*/*

**for**(i = 0; i < 0x220; i++)

pte[0x1000+i] = ((0x20000000+(i<<12))&0xFFFFF000)|PTE\_V|PTE\_W;

3.7 EPOS MMU初始化

### 激活MMU

在EPOS中，首先进行MMU初始化，在初始化MMU以前所有使用的地址都是物理地址，只有MMU激活以后使用到的所有地址都是虚拟地址，必须经过MMU转化后才能定位到虚拟地址对应的物理地址位置。

激活MMU使用到的是CP15协处理器，一下代码是激活MMU，详细见图3.8 激活MMU。

注意：GCC内联汇编的一般格式如下[5]：

asm(

代码列表

：输出运算符列表

：输入运算符列表

：被更改资源列表

);

**asm**(

"mcr p15,0,%0,c2,c0,0**\n**"

"mvn r0,#0**\n**"

"mcr p15,0,r0,c3,c0,0**\n**"

"mov r0,#0x1**\n**"

"mcr p15,0,r0,c1,c0,0**\n**"

"mov r0,r0**\n**"

"mov r0,r0**\n**"

"mov r0,r0**\n**"

:

: "r" (pgdir)

: "r0"

);

图3.8 激活MMU

### EPOS恒等映射

在EPOS中，实现恒等映射是通过设置两部分页目录项都共同执行同一个小页表，该小页表里面存放的就是物理内存的基地址。图3.9显示恒等映射原理，图3.10是实现恒等映射的代码，图3.11是清除恒等映射的代码。

图3.9显示恒等映射原理

|  |
| --- |
|  |
|  |
| **……...** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| **……...** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

20张小页表

页目录

0x000

0xC00

4096项

*/\*分配20张小页表，并且填充页目录\*/*

**for**(i = 0; i < NR\_KERN\_PAGETABLE; i++)

{

pgdir[i] = pgdir[i+(KERNBASE>>PGDR\_SHIFT)] = (physfree)|PTE\_V;

memset((void \*)physfree, 0, (1<<10));

physfree += (1<<10);

}

图3.10 实现恒等映射

*/\**

*\* 分页已经打开，切换到虚拟地址运行*

*\*/*

*/\*重定位内核\*/*

asm (

"add sp, sp, #0xC0000000**\n\t**"

"add r11, r11, #0xC0000000**\n\t**"

);

int i;

*/\* 清除恒等隐射 \*/*

**for**(i= 1; i < NR\_KERN\_PAGETABLE; i++)

PTD[i] = 0;

图3.11 清除恒等映射

## 中断处理

### 初始化中断向量表

首先根据ARM中断向量表规定的， 详细规定参考表2.5中断向量表。在EPOS中，我们使用一下代码来进行设置中断向量表。图3.12，EPOS中断向量表[6]。

ldr pc, \_vector\_reset

ldr pc, \_vector\_undefined

ldr pc, \_vector\_swi

ldr pc, \_vector\_prefetch\_abort

ldr pc, \_vector\_data\_abort

ldr pc, \_vector\_reserved

ldr pc, \_vector\_irq

ldr pc, \_vector\_fiq

图3.12 EPOS中断向量表

图 3.13 显示了EPOS操作系统中断向量表的内存布局：

内存 地址 模式

图 3.13 EPOS操作系统中断向量表内存布局

|  |
| --- |
| 复位 |
| 未定义指令 |
| 软件中断 |
| 预取指令异常 |
| 数据异常 |
| 保留 |
| 中断 |
| 快速中断 |
|  |

|  |
| --- |
| 0x0 |
| 0x4 |
| 0x8 |
| 0xC |
| 0x10 |
| 0x14 |
| 0x18 |
| 0x1C |
|  |

|  |
| --- |
| SVC |
| UND |
| SVC |
| ABT |
| ABT |
|  |
| IRQ |
| FIQ |
|  |

### IRQ中断开关

在ARM中，IRQ中断默认是关闭的，所以首先我们在EPOS的初始化代码中开启IRQ中断，具体开启IRQ中断和关闭IRQ中断的汇编代码在entry.S中实现，详细见图3.14开启IRQ中断和图3.15关闭IRQ中断[3]。

.globl \_enable\_interrupts

\_enable\_interrupts:

mrs r0, cpsr

bic r0, r0, *#0x80*

msr cpsr\_c, r0

mov pc, lr

3.14开启IRQ中断

.globl \_disable\_interrupts

\_disable\_interrupts:

mrs r0, cpsr

orr r0,r0,*#0x80*

msr cpsr\_c,r0

mov pc, lr

3.15 关闭IRQ中断

### 中断现场保护

在中断发生时，我们首先要保存R0~R4和R14(LR)，当处理完中断后，我们将先前保存的R0~R4恢复，将LR寄存器不是恢复到LR，而是将LR寄存器恢复到PC寄存器中，因为LR存放的是中断前程序的返回值，所以我们将LR恢复到PC中，以便程序继续向前执行。IRQ中断发生时的PC寄存器如图3.16所示：

|  |
| --- |
| 代码段 |
| 中断发生时正在执行的指令 |
| 从中断返回时要执行的指令 |
| PC寄存器指向的指令 |

3.16 IRQ中断发生时PC寄存器

PC-8

PC-4

PC

图 3.16显示了IRQ中断发生时PC寄存器的状态，中断发生时，CPU自动将PC寄存器的值保存在中断模式下的私有寄存器LR中，一旦中断完成后，我们需要继续跳转到中断前的程序继续执行，所以我们要把LR提前压栈操作，为了能够正确的跳转到中断前的程序执行，所以我们将LR寄存器减4并且将该值压栈操作，等中断完成后，将该值恢复到PC寄存器并且实现程序继续执行。表 3.1列出了其它异常情况下的返回值LR的调整值。

表 3.1 各种异常的LR值

|  |  |  |
| --- | --- | --- |
| **中断** | **地址** | **说明** |
| 复位 | 无 | 未定义复位时的LR |
| 未定义指令 | LR | 指向未定义指令的下一条指令 |
| 软件中断 | LR | 指向软中断的下一条指令 |
| 预取指令异常 | LR-4 | 指向预取异常的下一条指令 |
| 数据异常 | LR-8 | 指向数据异常的下一条指令 |
| 中断 | LR-4 | 指向被中断的下一条指令 |
| 快速中断 | LR-4 | 指向被快速中断的下一条指令 |

下面我们列出EPOS中处理中断时的现场保护和现场恢复情况的代码图(图3.17 EPOS中断现场保护和恢复)。

\_\_vector\_fiq:

sub r14, r14, *#4*

stmfd r13!, {r0-r3, r14}

bl common\_fiq\_handler

ldmfd r13!, {r0-r3, pc}^

图 3.17 FIQ中断现场保护和恢复

### 定时器中断处理

在EPOS我们设置设置定时器的PreDivider寄存器的值为0xF9，这就意味着定时器Load的值每隔1 us(250000000/(0xF9+1))自动减一，然而我们设置定时器Load的值为1000000，也就是每隔1s定时器就会产生IRQ中断，图3.18为EPOS定时器的初始化代码，我们利用定时器中断服务函数来实现多任务的切换，详细的切换请见3.5多任务。

void init\_arm\_timer(unsigned int Load)

{

ArmTimer = (arm\_timer\_t \*)ARMTIMER\_BASE;

irq\_controller\_t \*IRQcontroller = (irq\_controller\_t \*)INTERRUPT\_CONTROLLER\_BASE;

IRQcontroller->Enable\_Basic\_IRQs = BASIC\_ARM\_TIMER\_IRQ;

ArmTimer->Load = Load;

ArmTimer->Reload = Load;

ArmTimer->PreDivider = 0xF9; *//timer\_clock = 250000000/(PreDivide + 1)*

ArmTimer->Control = ARMTIMER\_CTRL\_23BIT |

ARMTIMER\_CTRL\_PRESCALE\_1 |

ARMTIMER\_CTRL\_INT\_ENABLE |

ARMTIMER\_CTRL\_ENABLE;

}

图3.18 EPOS定时器初始化

## 多任务

### 创建多任务

在EPOS中每个任务就是一个TCB(task control block)结构体，EPOS中为每个任务都分配了一个私有的栈空间，每个任务的栈大小为4K。我们将新创建的任务状态设置为就绪状态，并且将其加入到任务列表中，当调度算法选择到该任务的时候，那么该任务就会开始执行。

### 任务切换

当正在运行的任务时间片段到后，EPOS将通过任务调度算法将选择下一个任务来替换当前执行的任务，那么在切换任务的时候，我们首先要保存当前正在运行的任务状态，以便下一次能够正确的恢复该任务到运行状态，然后将选择的任务从就绪状态恢复到运行状态。

首先，我们使用r13\_irq寄存器作为临时栈指针保存r0和r1寄存器的值，然后将当前任务的tcb首地址存放在寄存器r1中，将当前任务的栈指针寄存器存取在寄存器r0中，随后将临时保存的r0和r1寄存器保存在栈上，将r2~r12保存到栈上，以便任务再次被唤醒是将其恢复，将lr寄存器保存在当前任务的pc寄存器上，接下来，我们将模式由irq转化为SVC，将svc模式下的lr、sp和spsc保存到当前任务的栈上，随后使用bl指令跳转到调度函数去执行选择下一个被唤醒的任务，详细的保存代码见图 2.19 任务保存。

sub lr, lr, #4

//保存所有的寄存器状态,要把寄存器保存到内存中.先用sp作为临时的基地址

str r0, [sp, #-4]

str r1, [sp, #-8]

.word task\_global //加载任务状态内存基地址

ldr r1, [pc, #-12]

ldr r0, [r1]

ldr r1, [sp, #-4]

str r1, [r0, #(0 \* 4)] //r0

ldr r1, [sp, #-8]

str r1, [r0, #(1 \* 4)] //r1

str r2, [r0, #(2 \* 4)] //r2

str r3, [r0, #(3 \* 4)]

str r4, [r0, #(4 \* 4)]

str r5, [r0, #(5 \* 4)]

str r6, [r0, #(6 \* 4)]

str r7, [r0, #(7 \* 4)]

str r8, [r0, #(8 \* 4)]

str r9, [r0, #(9 \* 4)]

str r10, [r0, #(10 \* 4)]

str r11, [r0, #(11 \* 4)]

str r12, [r0, #(12 \* 4)] //r12

str r14, [r0, #(15 \* 4)] //r15

mrs r1, spsr

str r1, [r0, #(16 \* 4)] //cpsr

mrs r1, cpsr //从irq切换到svc

bic r1, r1, #0x1F

orr r1, r1, # 0x13

msr cpsr, r1

str r13,[r0, #(13 \* 4)] //13

str r14,[r0, #(14 \* 4)] //14

mrs r1, spsr

str r1, [r0, #(17 \* 4)] //spsr

图 2.19 任务保存

当执行完调度函数后，我们将当前正在执行的任务指针切换成被新选择的任务指针，将该任务栈上的所有保存数据恢复到寄存器中，并且恢复PC寄存器进行继续执行被选择的任务，详细恢复代码见图 2.20 任务恢复。

//==========================================

.word task\_global //加载任务状态内存基地址

ldr r1, [pc, #-12]

ldr r0, [r1]

ldr r2, [r0, #(2 \* 4)] //r2

ldr r3, [r0, #(3 \* 4)]

ldr r4, [r0, #(4 \* 4)]

ldr r5, [r0, #(5 \* 4)]

ldr r6, [r0, #(6 \* 4)]

ldr r7, [r0, #(7 \* 4)]

ldr r8, [r0, #(8 \* 4)]

ldr r9, [r0, #(9 \* 4)]

ldr r10, [r0, #(10 \* 4)]

ldr r11, [r0, #(11 \* 4)]

ldr r12, [r0, #(12 \* 4)]

ldr r13, [r0, #(13 \* 4)]

ldr r14, [r0, #(14 \* 4)] //r14

ldr r1,[r0, #(15 \* 4)] //将pc的值临时放在sp-4中,因为寄存器都恢复后只有sp指向的内存是安全的

str r1, [sp, #-4]

ldr r1, [r0, #(16 \* 4)] //cpsr

str r1, [sp, #-8]

msr cpsr, r1 //这里也就打开中断了

ldr r1, [r0, #(1 \* 4)] //r1

ldr r0, [r0, #(0 \* 4)] //r2

ldr pc,[sp, #-4] //最后恢复pc,跳转

图 2.20 任务恢复

# 系统展示

## 源码编译操作

在EPOS中我们采用Make来管理源代码，我们只需在命令行下执行make命令(首先要将工作区切换到包含Makefile文件的目录下)，那么make就会根据我们所编写的makefile文件内容生成kernel.img文件，随后我们将kernel.img文件拷贝到SD卡中，这样就完成了EPOS的编译流程，详细步骤如下：

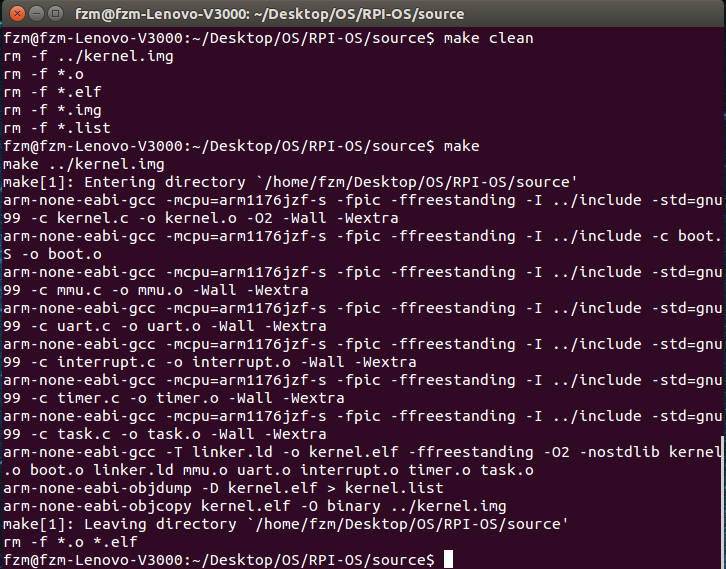


图 4.1 源码编译

1. 执行 cd Desktop/OS/RPI/source 命令，将当前工作区切换到对应源码的目录下。
2. 执行 make clean 将上次编译的中间文件都删除掉。
3. 执行make 生成最新的kernel.img文件。
4. 将生成的kernel.img文件拷贝到SD卡中。

## 系统运行结果

首先，我们在Ubuntu 14.04操作系统上安装minicom串口调试器，安装命令 sudo apt-get install minicom。

接下来，我们要配置minicom，在命令行下输入 sudo minicom –s进行对minicom配置，配置结果如图 4.2 minicom配置，将minicom的设备修改成对应串口的设备号，此处为/dev/ttyUSB0，将波特率设置为9600。

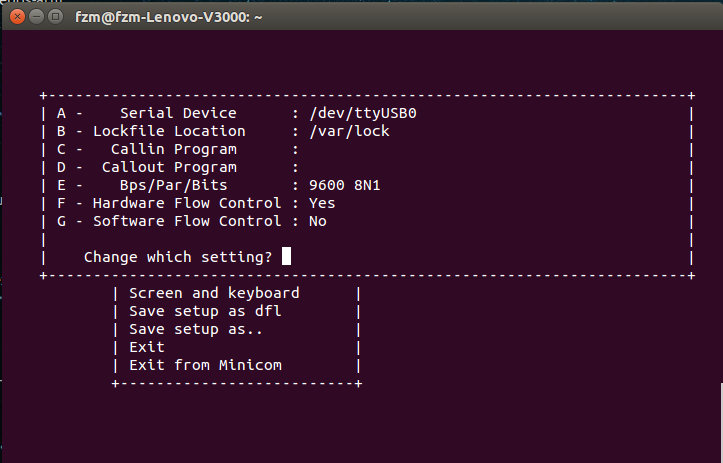


图 4.2 minicom配置

最后，将RPi与PC机连接(见图 4.3)，开启电源，我们可以通过minicom看到RPi的输出结果。



图 4.3 连接图

图4.4为主函数功能图，首先调用md\_startup函数进行虚拟内存初始化，然后初始化uart，通过uart输出3条信息，随后创建3个线程，线程函数见图4.5。最后开启定时器。

*/\**

*\* 机器相关（Machine Dependent）的初始化*

*\*/*

md\_startup(mbi, end-0xC0000000);

uart\_init();

uart\_puts("Welcome to ARM-EPOS**\r\n**");

uart\_puts("Copyright (C) 2005-2015 MingJian Hong<hongmingjian@gmail.com>**\r\n**");

uart\_puts("Author: fengzhimin **\r\n**");

uart\_puts("All rights reserved.**\r\n\r\n**");

task\_init();

unsigned char rank = MAX\_rank ;

unsigned int task\_func = (unsigned int)task\_idle0;

unsigned char TID = task\_create( rank , task\_func);

task\_run(TID);

task\_func = (unsigned int)task\_idle1;

TID = task\_create( rank , task\_func);

task\_run(TID);

task\_func = (unsigned int)task\_idle2;

TID = task\_create( rank , task\_func);

task\_run(TID);

init\_arm\_timer(Kernel\_1Hz);

cli();

*/\**

*\* 分页已经打开，切换到虚拟地址运行*

*\*/*

*/\*重定位内核\*/*

asm (

"add sp, sp, #0xC0000000**\n\t**"

"add r11, r11, #0xC0000000**\n\t**"

);

int i;

*/\* 清除恒等隐射 \*/*

**for**(i= 1; i < NR\_KERN\_PAGETABLE; i++)

PTD[i] = 0;

**while**(1);

图 4.4 主程序代码

图 4.5为线程函数：

void task\_idle0(void)

{

uart\_puts("first task create success!**\r\n**");

**while**(1);

}

void task\_idle1(void)

{

uart\_puts("second task create success!**\r\n**");

**while**(1);

}

void task\_idle2(void)

{

uart\_puts("third task create success!**\r\n**");

**while**(1);

}

图4.5 线程函数

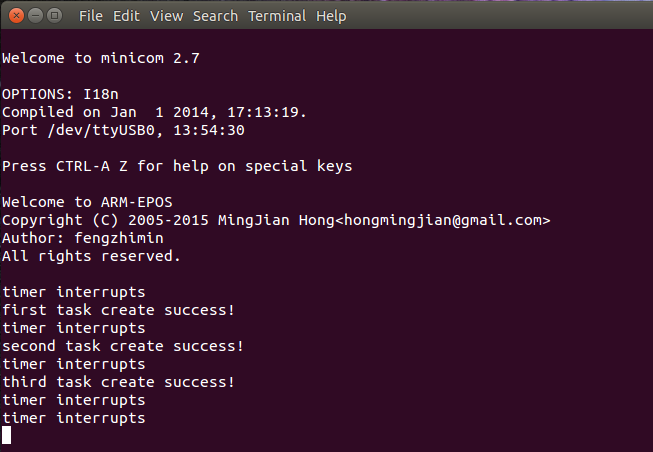


图 4.6 运行结果图

# 结论及展望

## 结论

本课题主要是将EPOS操作系统移植到ARM体系下，使EPOS操作系统可以在真实的设备上运行，摆脱了只能在虚拟机上运行的现状，让操作系统课程的实验变得更加有趣，根据EPOS底层实现的方式不同可以对比X86和ARM的不同之处。

本课题完成了EPOS在ARM体系下的底层开发，让EPOS的内核可以在RPi上运行，具体包括功能包含以下几点：

* + - 1. 系统引导项：将EPOS在RPi上运行起来。
      2. MMU：实现虚拟内存映射。
      3. 中断处理：实现了非嵌套中断处理。
      4. 系统定时器：将RPi的系统定时器驱动成功。
      5. 多任务：实现了多任务。

## 展望

目前EPOS在RPi上可以运行，实现了一个操作系统的基本功能，缺少的功能是没有文件系统和没有将RPi的GPU利用起来，接下来将进一步的对EPOS进行开发，在EPOS中实现一个简单的文件系统，同时将RPi的GPU启动起来，并且实现简单的绘图功能，利用RPi强大的图像处理能力，以后将RPi的摄像头驱动假如到EPOS中，那么到时候就可以利用EPOS来做图像处理等功能。

# 致 谢

大学四年匆匆而过，在大学的生活里，我收获很多，不光是知识的增长，同时也学会如何与同学相处，如何自学一门新的学科技术，在此也非常感谢软件学院的学长们，他们给了我很多在学习上的建议，让我在学习的道路上没有走太多的弯路。

回顾大学四年，感觉在做毕业设计上下了很大的功夫，同时也从中收获很多，在此特别感谢我的指导老师—洪明坚老师，在做毕业设计期间，洪老师在技术上帮助了我很多，同时我也从老师那里学习到了很多关于底层开发的知识。自己一直都非常喜欢底层软件的开发，希望以后的工作方向就是底层开发，虽然在学习这方面知识的时候遇到了很多困难，不过我相信没有一条道路是畅通无阻的。

最后，衷心感谢在百忙之中抽出宝贵的时间对本论文进行评审的各位专家！本科毕业喽 ！！！

# 参 考 文 献

[1] ARM\_architecture\_reference\_manual ARM

[2] BCM2835-ARM-Peripherals 2012 Broadcom Corporation

[3] Andrew N. Sloss. ARM-Interrupts.April 25th, 2001

[4] ARM 汇编语言官方手册（中文） ARM

[5] 李无言.一步一步写嵌入式操作系统--ARM编程的方法与实践.电子工业出版社

[6] 杜春雷.ARM体系结构与编程.清华大学出版社

[7] 周立功.ARM嵌入式系统基础教程.北京航空航天大学出版社

[8] 马忠梅马广云,徐英慧等.ARM嵌入式处理器结构与应用基础[M].北京:北京航空航天大学出版社，2002

[9] Elsner D, Fenlason J. Using as: the GNU Assembler[J]. 2000.

[10] Stevens W R. UNIX 环境高级编程: 英文版[M]. 机械工业出版社, 2002.

[11] Stallman R M. Using and porting the gnu compiler collection[J]. Free Software Foundation, 1989, 59: 02111-1307.

[12] Rusling D A. The linux kernel[J]. 1999.

[13] Tanenbaum A S, Woodhull A S, 陈渝, 等. 操作系统设计与实现: 第三版[M]. 电子工业出版社, 2007.

[14] 文全刚. 汇编语言程序设计. 北京航空航天大学出版社, 2007

[15] 孙红波. ARM与嵌入式技术. 电子工业出版社, 2006

[16] 周立功.张华. 深入浅出ARM 7. 北京航空航天大学出版社, 2005

[17] 毛玉萃.牛玉军.赵宏伟.操作系统教程. 清华大学出版社, 2013

[18] 邱传慧.操作系统原理. 人民邮电出版社, 1992

[19] 于渊.一个操作系统的实现. 电子工业出版社, 2009-6-1

[20] 于渊.自己动手写操作系统. 电子工业出版社, 2005-8

[21] 孙红波. ARM与嵌入式技术. 电子工业出版社, 2006

[22] 三恒星科技. ARM 7易学通. 人民邮电出版社, 2006

[23] 陈赜. ARM9嵌入式技术及Linux高级实践教程. 北京航天航空大学出版社, 2005

[24] 张勇. ARM原理与C程序设计. 西安电子科技大学出版社, 2009

[25] 汪明虎.欧文盛. ARM嵌入式Linux应用开发入门. 中国电力出版社, 2008

[26] 王宇行. ARM程序分析与设计. 北京航空航天大学出版社, 2008

[27] 王小强. ARM处理器裸机开发实战. 电子工业出版社, 2012

[28] 刘同法, 彭继卫, 肖志刚. ARM Cortex-MO最小系统应用与C程序编写. 北京航空航天大学出版社, 2014

[29] 董胡, 刘刚, 钱盛友. ARM 9嵌入式系统开发与应用. 电子工业出版社, 2015

[30] 陈章龙, 唐志强, 涂时亮. 嵌入式技术与系统. 北京航空航天大学出版社, 2004

[31] 王宜怀, 吴瑾, 蒋银珍. 嵌入式系统原理与实践. 电子工业出版社, 2012

[32] 侯殿有. 嵌入式系统开发基础. 清华大学出版社, 2013