嵌入式系统X210V3

实 验 指 导 书

南京大学 电子科学与工程学院

2014年8月

目录

实验室规章制度

1. 实验室是教学科研重地,实验时要保持安静的环境,不得大声喧闹
2. 学生必须遵守纪律,不得迟到、早退和无故缺席,有事应事先请假
3. 实验前先预习,预习通过才能进行实验;实验结果须经教师检查认可
4. 注意保持实验室环境的卫生整洁,及时清理个人的遗留物品
5. 爱护仪器设备,正确使用实验仪器和设备,违章损坏要酌情赔偿

♠ ♡ ♢ ♣ ♠ ♡ ♢ ♣ ♠ ♡ ♢ ♣ ♠ ♡ ♢ ♣

实验报告格式

1. 实验目的
2. 实验内容与要求
3. 实验设计,包括
   - 硬件结构
   - 软件设计思路
4. 实验记录与分析(包括关键部分的程序清单、说明,实验测量到的数据、波形, 观察到的现象及分析等等)
5. 实验体会、小结及建议
6. 参考文献

其中前三项应在预习中完成,作为预习报告的一部分。

实验涉及到的软件,要求画出程序流程,附上关键部分的程序代码及注释,并说明程序的运行方法和结果。

第一章　实验系统介绍

1.1　性能概括

嵌入式实验系统x210v3 是基于45 nm 工艺S5PV210芯片的开发平台。核心处理器S5PV210为Cortex–A8架构, 主频可到1GHz, 支持1GiB DDR2, 片内32KiB I/D 缓存及512KiB 二级缓存, PowerVR SGX540 图形加速引擎, 支持MPEG4、H.263、H.264 1080P@30fps编码及MPEG4 1080P@30fps解码, HDMI/TV–OUT, 引脚间距0.65mm, 17×17 mm FBGA封装。

x210v3 由核心板,底板和液晶板三大块组成,核心板采用8 层板工艺设计, 底板留有丰富的外设接口,几乎具备210 所有功能的调试,液晶板采用7吋电容触摸屏。

1.2　系统资源

系统硬件配置如下:

- 内核:ARM Cortex-A8,主频1GHz
- 内存:512MB
- Flash:4GB inand-Flash
- 24 位RGB 接口
- 四路USB HOST 接口
- USB OTG 接口
- 两路RS232 接口
- SD 卡接口
- 四LED 指示
- 功能按键(开机/复位等)
- HDMI高清视频接口
- 音频输入/输出接口
- 有线以太网DM9000CEP;
- 电容触摸屏
- 摄像头接口
- 实时时钟
- 支持G-sensor、SPI、I2C、UART、USB/WiFi、GPS、GPRS等多种外围器件扩展

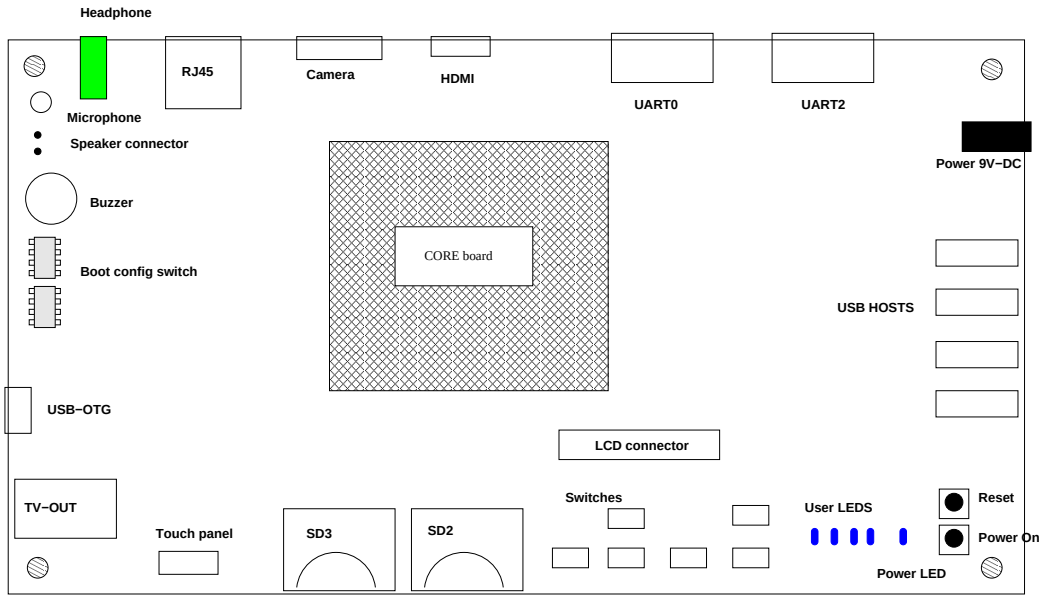


图 1.1: 底板布局

- MPEG2/MPEG4,H.263,H.264 视频编解码1080P@30fps
- 2D/3D 高性能图形加速;

1.3 软件

在x210v3平台上, 可以进行Linux、WinCE等多种操作系统的移植以及裸机实验。本课程主要围绕Linux 操作系统进行开发。

1.3.1 交叉编译工具链toolchain

由于嵌入式系统的局限性，不可能具有很大的存储能力和友好的人机交互开发界面，所以一般开发环境都必须安装在PC 上，再通过 toolchain 生成最终目标文件，将其运行在相应的目标平台上。

由于Cortex–A8 基于ARM 体系结构, 所以在基于Cortex–A8 开发过程中必须使用ARM 的交叉编译。这个编译器环境将使用下面的GNU 工具：

- GNU gcc compilers for c, c++，包括编译器、链接器等;
- GNU binutil，包括归档、目标程序复制和转换、代码分析调试等工具;
- GNU C Library，支持目标代码的c 语言库;
- GNU C header，头文件。

通用的 GNU Tools 都是针对x86 体系结构的，而上述的GNU 交叉编译工具是针对 ARM 的。最终编译后产生的二进制文件只能在ARM 架构的处理器上运行。

1.3.2 工具链安装

GNU 工具链提供完整的源代码，可以在 PC 机上用x86 平台的编译工具编译安装，也可以直接下载二进制代码包解压安装。

本实验使用的交叉编译工具链路径是/usr/local/arm-2009q3, 可执行程序在/usr/local/arm-2009q3/bin 目录下。实验中请将该目录添加到环境变量“PATH” 中, 或给出完整的路径和编译程序名, 如/usr/local/arm-2009q3/bin/arm-none-linux-gnueabi-gcc

1.3.3 嵌入式系统软件

嵌入式系统软件从下到上通常由bootloader、操作系统、文件系统和应用软件等若干层构成。bootloader的目的是加载并引导操作系统运行,有时也会负责文件系统的加载。bootloader还负责核心软件的升级。一旦操作系统启动, bootloader的任务便暂告终结,直到系统重启。

本系统使用的bootloader名为u-boot。根据本实验系统特点, 厂家对u-boot做了针对性的修改, 例如对inand-Flash操作以及支持Android系统升级的fastboot 模式。

厂家在此平台上移植的Linux 操作系统内核版本是2.6.35(Android4.0.4使用的内核版本是3.0.8)。用户可在此基础上进一步剪裁和优化。完成内核正常启动后, 与硬件相关的工作基本上都可以由内核解决, 以后的软件开发可以由用户自由发挥, 例如移植不同版本的根文件系统、不同的图形用户接口、桌面等。

第二章　嵌入式系统开发实验

2.1　实验目的

了解嵌入式系统的开发环境、内核的下载和启动过程

2.2　嵌入式系统开发过程

嵌入式系统与普通计算机系统存在很大区别,主要表现在：

- 嵌入式系统往往不提供BIOS，因此基本输入输出系统需要程序员完成；
- 嵌入式系统开发缺乏友好的人机界面，为程序调试带来困难；
- 嵌入式系统开发能力不如通用计算机系统。通常采用交叉编译的方式，在通用计算机上编译嵌入式系统的程序；
- 嵌入式系统的存储空间有限，对程序优化要求较高。

上述特点决定了在嵌入式系统开发中所使用的工具、方法的特殊性。

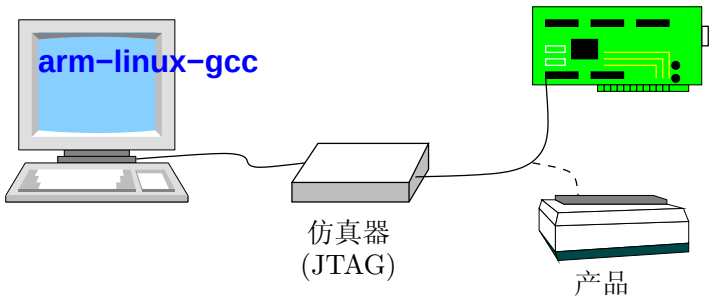


图 2.1: 宿主机/目标机的开发模式

本系统主要基于串行口和网口进行开发。

2.2.1　bootloader

PC 机中的引导加载程序由 BIOS 和位于硬盘的主引导记录 MBR(Master Boot Recorder) 中的 OS Boot Loader 一起组成。 BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader。 Boot Loader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

嵌入式系统中，通常并没有像 BIOS 那样的固件程序，因此整个系统的加载启动任务完全由 bootloader 来完成。用于引导嵌入式操作系统的 bootloader 有 U-Boot、 vivi、 RedBoot等等。 bootloader 的主要作用是：

1. 初始化硬件设备；
2. 建立内存空间的映射图；
3. 完成内核的加载，为内核设置启动参数。

2.2.2 bootloader程序结构框架

嵌入式系统中的 boot loader 的实现完全依赖于 CPU 的体系结构，因此大多数 bootloader 都分两个阶段。依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在阶段一中，且通常都用汇编语言来实现，以达到短小精悍的目的。阶段一通常包括以下步骤：

1. 硬件设备初始化；
2. 拷贝Boot Loader的程序到RAM空间中；
3. 设置好堆栈；
4. 跳转到阶段二的c入口点。

阶段二则通常用c语言来实现，这样可以实现一些复杂的功能，而且代码会具有更好的可读性和可移植性。这一阶段主要包括以下步骤：

1. 初始化本阶段要使用到的硬件设备；
2. 系统内存映射(memory map)；
3. 将kernel映像和根文件系统映像从Flash读到RAM空间中；
4. 为内核设置启动参数；
5. 调用内核。

2.2.3 串口设置(minicom)

多数嵌入式系统都通过异步串行接口 (UART)进行初级引导。这种通信方式是将字符一位一位地传送，一般是先低位、后高位。因此，采用串行方式，双方最少可以只用一对连线便可实现全双工通信。字符与字符之间的同步靠每个字框的起始位协调，而不需要双方的时钟频率严格一致，因此实现比较容易。

RS-232C 是通用异步串行接口中最常用的标准。它原是美国电子工业协会推荐的标准 (EIA RS-232C, Electronics Industrial Association Recommended Standard)， 后被世界各国所接受并应用到计算机的I/O接口中。个人计算机系统中常使用25针或9针接插件 (DB-25/DB-9)连接。例如， DB-25按图2.2 定义了接口信号。

Linux系统用minicom 软件实现串口通信。

运行 minicom，ˆA–o，进入 minicom 的设置界面[1]。在 Serial port setup 项上修改下述设置：

[1] ˆA是minicom 的控制键，可以激活其他功能设置，例如ˆA–z 调出全部功能菜单，ˆA–x 退出minicom。也可以在minicom启动时加上选项“–s”直接进入设置界面

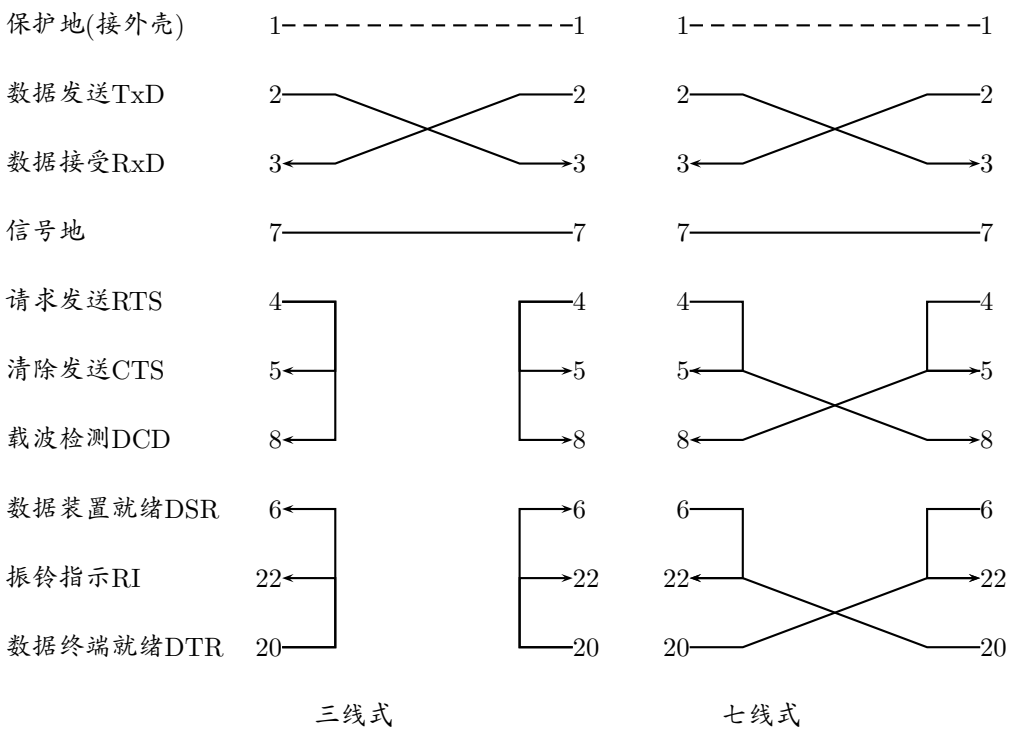


图 2.2: PC机串口引脚信号

```
Welcome to minicom 2.4

OPTIONS: I18n
Compiled on Jan 25 2010, 06:49:09.
Port /dev/ttyS0

Press CTRL-A Z for help on special keys

            +-----[configuration]------+
            | Filenames and paths      |
            | File transfer protocols  |
            | Serial port setup        |
            | Modem and dialing        |
            | Screen and keyboard      |
            | Save setup as dfl        |
            | Save setup as..          |
            | Exit                     |
            +--------------------------+




 CTRL-A Z for help |115200 8N1 | NOR | Minicom 2.4    | VT102 |      Offline
```

图 2.3: minicom界面

- A — “Serial Device”，串口通信口的选择。如果串口线接在PC机的串口1 上，则为/dev/ttyS0，如果连接在串口2上，则为/dev/ttyS1，依此类推。
- E — “Bps/Par/Bits”，串口参数的设置。设置通信波特率、数据位、奇偶校验位和停止位。本实验平台要求把波特率设置为115200bps，数据位设为8位，无奇偶校验，一个停止位；
- F — “Hardware Flow Control”、G — “Software Flow Control”，数据流的控制选择。按 “F” 或 “G” 键完成硬件软件流控制切换(即 “Yes” 与 “No” 之间的切换)。本实验系统都设置为“No”。

配置完成后，选择“Save setup as dfl”保存配置，并返回 minicom 的主界面。以后使用不再需要每次设置。

给开发板加电，这时可以在 minicom 主界面上看到开发板的启动信息。在短时间内(时间可通过bootloader命令设置)通过键盘干预, 将停止加载内核, 进入人机交互方式, 出现提示符“x210 #”。“help”命令可列出所有u-boot的命令；如需了解具体某个命令的使用, 可用“help + 命令”的方式。

以下列出本实验常用的命令:

- setenv: 设置环境变量。主要的环境变量有:
  - ipaddr, serverip, gatewayip，本机和服务器的IP地址，网关
  - bootargs, 启动参数，一般包括监控端口、内核启动参数，加载文件系统等，如setenv bootargs console=ttySAC2,115200 root=/dev/mtdblock2 rw rootfstype=yaffs2 init=/linuxrc, 表示用串口设备ttySAC2作为终端，波特率115200bps，根文件系统在eMMC卡(或FLASH)的第二分区，读写允许，yaffs2文件系统, 内核启动后执行根目录下的linuxrc 命令。
  - bootcmd, 启动命令，上电后或者执行boot 命令后调用。如setenv bootcmd ’movi read kernel 0xc0008000;bootm 0xc0008000’ 表示将eMMC 卡的kernel 分区读到内存以0xc0008000起始的地址中，然后从0xc0008000处开始运行。
- tftp, tftp 命令，如tftp 0xc0008000 zImage，将tftp 服务器目录中的文件zImage 通过tftp 协议读入内存0xc0008000 起始处。
- saveenv 保存环境变量设置。未经保存的环境变量，重启后将恢复原状。

2.2.4 tftp

tftp 是基于UDP协议的简单文件传输协议。目标板作为客户机， bootloader 默认采用 tftp 协议。主机安装 tftp-server ，作为 tftp 服务器。Linux系统的 tftp 服务由超级服务器 xinetd 管理。安装 tftp-server 后， /etc/xinetd.d/tftp 中大致是如下内容: [2]

```
service tftp
{
    socket_type = dgram
    protocol  = udp
```

[2]主机的不同发行版、不同的软件,使用方法和配置文件有所不同, 关键要了解服务的目的、目录、权限及启动方式; 下面的NFS服务亦然

```
    wait = yes
    user = root
    server = /usr/sbin/in.tftpd
    server_args     = -c -s /tftpboot
    disable = yes
    per_source = 11
    cps = 100 2
}
```

将“disable”的选项改为“no”，关闭防火墙，再用如下命令重启tftp服务:

```
# /etc/rc.d/init.d/xinetd restart
```

tftp的配置文件里表明，tftp 服务的主目录是 /tftpboot ，因此只有在这个目录下面的文件才可以通过 tftp 进行下载。我们需要 bootloader 从服务器上下载内核及文件系统。为实现这个目的，需要配置客户端(目标机)的网络 IP 地址，使其和主机处于同一网段，并注意不要和其他系统(包括主机和目标机)的IP发生冲突。u-boot中, 用“setenv ipaddr”和“setenv serverip” 分别设置本机和tftp 服务器的IP地址。

2.2.5 NFS服务器架设

NFS 是 Network File System 的缩写。 NFS 是由 Sun 公司开发并发展起来的一项用于在不同机器、不同操作系统之间通过网络共享文件的服务系统。 nfs-server 也可以看作是一个文件服务器，它可以让 PC 通过网络将远端的 nfs server 共享出来的档案挂载到自己的系统中。在客户端看来，使用 NFS 的远端文件就像是在使用本地文件一样。

NFS 协议从诞生到现在为止，已经有多个版本，如 NFS V2(rfc1094)、 NFS V3(rfc1813)、 NFS V4(rfc3010)等。

NFS 涉及到 portmap 和 NFS 两个服务。需要在主机打开这两个服务:

```
# chkconfig nfs on
# chkconfig portmap on
# service nfs restart
# service portmap restart
```

服务器的共享目录和权限在 /etc/exports 中设定。下面是这个文件的样例:

```
# this is a sample
# /etc/exports
/opt  192.168.1.*(rw,no_root_squash)
```

使用如下命令使修改生效:

```
# exportfs -a
```

目标板上的 Linux 启动后, BusyBox可以提供操作系统人机交互的基本功能。如果需要配置网络, 使用ifconfig, 和主机的用法相同:

```
# ifconfig eth0 192.168.1.xx
```

如果内核和BusyBox都支持NFS, 可以用 mount 命令将主机的 /opt 目录挂在 /mnt 目录下(设主机的IP为192.168.1.100):

```
# mount 192.168.1.100:/opt /mnt -o nolock -o proto=tcp
```

这样当访问目标板的 /mnt 目录时，访问的就是服务器上的 /opt 目录的内容.

2.2.6 使用gcc、g++等工具编译应用程序

1. 编写一个简单的独立程序;
2. 使用如下的 Makefile

```
CC      = /usr/local/arm-2009q3/bin/arm-none-linux-gnueabi-gcc

CFLAGS  =

TARGET  = hello
OBJS    = $(TARGET).o

all: $(TARGET)

$(TARGET) : $(TARGET).o
        $(CC) $(CFLAGS) $^ -o $@
$(TARGET).o : $(TARGET).c
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -f $(OBJS) $(TARGET) *.elf *.gdb
```

3. 将编译生成的可执行程序 hello 拷贝到 NFS 共享目录下，在目标板上运行该程序。

   如果不能运行, 请在gcc编译选项中加上–static

2.3 实验报告要求

归纳总结嵌入式系统下软件开发的一般流程

第三章　Linux内核配置和编译

3.1　实验目的

- 了解Linux 内核源代码的目录结构及各目录的相关内容
- 了解Linux 内核各配置选项内容和作用
- 掌握Linux 内核的编译过程

3.2　相关知识

3.2.1　内核源代码目录介绍

Linux 内核源代码可以从网上下载(http://www.kernel.org/pub/linux)。一般主机平台的Linux内核源码在/usr/src/linux 目录下。内核源代码的文件按树形结构进行组织，在源代码树的最上层可以看到如下一些目录：

- arch：arch 子目录包括所有与体系结构相关的内核代码。 arch 的每一个子目录都代表一个 Linux 所支持的体系结构。例如 arm 目录下就是 ARM 体系架构的处理器目录，包含我们使用的 PXA 处理器。
- include：include 子目录包括编译内核所需要的头文件。与ARM 相关的头文件在 include/asm-arm 子目录下。
- init：这个目录包含内核的初始化代码(不是系统的引导代码)，其中所包含 main.c 和 version.c 文件是研究Linux 内核的起点。
- mm：该目录包含所有独立于CPU 体系结构的内存管理代码，如页式存储管理、内存的分配和释放等。与ARM 体系结构相关的代码在arch/arm/mm 中。
- kernel：这里包括主要的内核代码。此目录下的文件实现大多数 Linux 的内核函数，其中最重要的文件是 sched.c。 Xscale 体系结构相关的代码在 arch/arm-pxa/kernel 目录中。
- drivers：此目录存放系统所有的设备驱动程序，每种驱动程序各占一个子目录：

  1. block：块设备驱动程序。块设备包括IDE 和scsi 设备；
  2. char：字符设备驱动程序。如串口、鼠标等；
  3. cdrom:包含Linux 所有的CD-ROM 代码；
  4. pci：PCI 卡驱动程序代码，包含PCI 子系统映射和初始化代码等；
  5. scsi：包含所有的SCSI 代码以及Linux 所支持的所有的SCSI 设备驱动程序代码；

6. net：网络设备驱动程序；
7. sound：声卡设备驱动程序；

- lib：放置内核的库代码；
- net：包含内核与网络的相关的代码；
- ipc：包含内核进程通信的代码；
- fs：fs 目录是所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统，如jffs2。
- scripts：目录包含用于配置内核的脚本文件等。每个目录下一般都有 depend 文件和一个 Makefile 文件，他们是编译时使用的辅助文件。仔细阅读这两个文件对弄清各个文件之间的相互依托关系很有帮助。

3.2.2 内核的配置的基本结构

Linux 内核的配置系统由四个部分组成：

1. Makefile：分布在Linux 内核源码中的 Makefile， 定义 Linux 内核的编译规则。顶层 Makefile 是整个内核配置、编译的总体控制文件；
2. 配置文件config.in：给用户提供配置选择的功能。“.config”是内核配置文件，包括由用户选择的配置选项，用来存放内核配置后的结果；
3. 配置工具：包括对配置脚本中使用的配置命令进行解释的配置命令解释器和配置用户界面(基于字符界面: make config， 基于 ncurses 界面: make menuconfig， 基于 X-Window 图形界面: make xconfig)；
4. Rules.make：规则文件，被所有的Makefile 使用。

3.2.3 编译规则Makefile

利用make menuconfig(或make config、make xconfig)对Linux 内核进行配置后，系统将产生配置文件“.config”。之前的配置文件备份到“.config.old”，以便用 make oldconfig 恢复上一次的配置。

编译时，顶层Makefile 完成产生核心文件(vmlinux)和内核模块(module)两个任务，为了达到此目的，顶层Makefile 将读取 .config 中的配置选项，递归进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile 进行编译。配置文件(.config)中有许多配置变量设置，用来说明用户配置的结果。例如“CONFIG_ MODULES=y”表明用户选择了 Linux 内核的模块功能。

配置文件(.config)被顶层Makefile 包含后，就形成许多的配置变量，每个配置变量具有四种不同的取值：

- y — 表示本编译选项对应的内核代码被静态编译进Linux 内核；
- m — 表示本编译选项对应的内核代码被编译成模块；
- n — 表示不选择此编译选项；
- 如果根本就没有选择，那么配置变量的值为空。

除了Makefile 的编写外，另外一个重要的工作就是把新增功能加入到 Linux 的配置选项中来提供功能的说明，让用户有机会选择新增功能项。 Linux 所有选项配置都需要在 config.in 文件中用配置语言来编写配置脚本，然后顶层 Makefile 调用 scripts/Configure， 按照 arch/arm/config.in 来进行配置。命令执行完后生成保存有配置信息的配置文件 .config。

3.3 编译内核

3.3.1 Makefile 的选项参数

编译 Linux 内核常用的 make 命令选项包括 config、 dep、 clean、 mrproper、 zImage、 bzImage、 modules、 modules_install等等。

1. config/xconfig/menuconfig：内核配置。调用./scripts/Configure 按照 arch/i386/config.in 来进行配置。命令执行后产生文件 “.config”，其中保存着配置信息。下次在做 make config 时将产生新的 “.config” 文件，原文件 “.config” 更名为 “.config.old”。
2. dep：建立依赖关系，产生两个文件 “.depend” 和“.hdepend”， 其中 ”.hdepend” 表示每个.h 文件都包含其他哪些嵌入文件，而 “.depend” 文件有多个，在每个会产生目标文件 .o 文件的目录下均存在，它表示每个目标文件都依赖于哪些嵌入文件.h。2.6以上版本的内核编译不需要这个步骤。
3. clean：清除以前构核所产生的所有的目标文件、模块文件、核心以及一些临时文件等，不产生任何新文件。
4. mrproper：删除以前在构核过程产生的所有文件，即除了做 make clean 外，还要删除 “.config”、“.depend” 等文件，把核心源码恢复到最原始的状态。下次构核时必须进行重新配置。
5. make,make zImage, make bzImage，编译内核，通过各目录的 Makefile 文件进行，会在各个目录下产生一大堆目标文件。如核心代码没有错误，将产生文件 vmlinux， 这就是所构的核心。同时产生映像文件 System.map 。 zImage 和 bzImage 选项是在 make 的基础上产生压缩的核心映像文件。正常编译完成后, 生成的ARM 内核 zImage 文件在目录 arch/arm/boot 中。需将其移至tftp 服务器目录供下载。
6. modules、modules_install：模块编译和安装。当内核配置中有选择模块时，这些代码不被编入内核文件 vmlinuz， 而是通过 “make modules” 编译成独立的模块文件 .ko (2.4版本的模块扩展名是.o)。 “make modules_install ”将这些文件复制到内核模块文件目录(PC机上通常是 /lib/modules/$VERSION/.....)。

简单的内核配置和编译过程通常是make menuconfig → make clean → make dep(kernel2.4) → make → make zImage。

3.3.2 内核配置项介绍

内核配置主目录有下面这些分支：

1. General setup, 内核配置选项和编译方式等等

2. Enable loadable module support, 利用模块化功能可将不常用的设备驱动或功能作为模块放在内核外部，必要时动态地加载。操作结束后还可以从内存中删除。这样可以有效地使用内存，同时也可减小了内核的大小。

   模块可以自行编译并具有独立的功能。即使需要改变模块的功能，也不用对整个内核进行修改。文件系统、设备驱动、二进制格式等很多功能都支持模块。开发过程中通常都需要选中这项。
3. System Type，处理器架构及相关选项, 根据开发的对象选择。
4. Networking options，网络配置。
5. Device Drivers, 设备驱动。包括针对硬盘、 CDROM 等以 block 为单位进行操作的存储装置和以数据流方式进行操作的字符设备, 还包括网络设备、USB设备、多媒体接口、图形接口、声卡等等。和系统硬件相关的配置主要在这里。
6. File systems，文件系统。对 Linux 可访问的各个文件系统的设置。所有的操作系统都具有固有的文件系统格式。一般为了对不同操作系统的文件系统进行读写操作需安装特殊的应用程序。但是在 Linux 系统中可以同 kernel 模块完成这些操作。

3.4 实验内容

配置一个完整的内核，尽可能理解配置选项在操作系统中的作用；
将编译的内核文件复制到tftp 服务器目录，在目标机中下载并运行:

```
x210 # tftp c0008000 zImage
x210 # bootm c0008000
```

3.5 实验报告要求

- 总结内核映像文件的生成方法及对操作系统的作用；
- 内核配置中，哪些选项对操作系统的正常启动是必须的？

第四章　嵌入式文件系统的构建

4.1　实验目的

- 了解嵌入式操作系统中文件系统的类型和作用
- 了解JFFS2 文件系统的优点及其在嵌入式系统中的作用
- 掌握利用BusyBox 软件制作嵌入式文件系统的方法
- 掌握嵌入式Linux 文件系统的的挂载过程

4.2　Linux 文件系统的类型

4.2.1　ext 文件系统

ext2fs 是Linux 2.4 版本的标准文件系统, 它已经取代了扩展文件系统(extfs)。 extfs 支持的文件最大为 2 GB , 支持的最大文件名长度为255 个字符, 且不支持索引节点(包括数据修改时间标记)。ext2fs 取代extfs 具有以下一些优点：

- ext2fs 支持达4 TB 的内存；
- ext2fs 文件名称最长可以到1012 个字符；
- 在创建文件系统时, 管理员可以根据需要选择存储逻辑块的大小(通常大小可选择 1024、 2048 或 4096 字节)；
- ext2fs 可以实现快速符号链接(类似于 windows 文件系统的快捷方式), 不需为符号链接分配数据块, 并且可将目标名称直接存储在索引节点 (inode) 表中。这使文件系统的性能有所提高, 特别在访问速度上。

由于ext2fs 文件系统的稳定性、可靠性和健壮性, 几乎在所有基于 Linux 的系统(包括台式机、服务器和工作站, 甚至一些嵌入式设备)上都使用ext2fs 文件系统。

4.2.2　NFS 文件系统

NFS 是一个RPC service。它由 SUN 公司开发, 于 1984 年推出。NFS 文件系统能够使文件实现共享。它的设计是为了在不同的系统之间使用, 所以NFS 文件系统的通信协议设计与操作系统无关。当使用者想使用远端文件时, 只要用 “mount” 命令就可以把远端文件系统挂载在自己的文件系统上, 使远端的文件在使用上和本地机器的文件没有区别。NFS 的具体配置可参考第二章的网络文件系统 NFS 的配置。

Linux 内核可以支持NFS 的根文件系统。为实现这项功能, 在内核配置中需要选中“File Systems → Network File Systems → NFS client support → Root file system on NFS”, 而该选项依赖于“Networking support”中的“IP: kernel level autoconfiguration ”。此外，需要在bootloader 中向内核传递NFS 作为根文件系统的信息:

```
x210 # setenv rootfs root=/dev/nfs rw nfsroot=<nfs_server_ip>:<nfs_Root_Dir>
x210 # setenv nfsaddrs nfsaddrs=<client_ip>:<nfs_server_ip>:<gateway>:<mask>
x210 # setenv bootargs console=ttySAC2,115200 $rootfs $nfsaddrs
```

系统启动后, “nfs_Root_Dir”就成为系统的根目录“/”, 它和下面制作的ramdisk 中的根目录结构是相同的。

4.2.3 JFFS2 文件系统

JFFS 文件系统是瑞典Axis 通信公司开发的一种基于 FLASH 的日志文件系统。它在设计时充分考虑了 FLASH 的读写特性和电池供电的嵌入式系统的特点。在这类系统中, 必需确保在读取文件时如果系统突然掉电, 其文件的可靠性不受到影响。对 Red Hat 的 Davie Woodhouse 进行改进后, 形成了 JFFS2。JFFS2 克服了 JFFS 的一些缺点, 使用了基于哈希表的日志节点结构, 大大加快了对节点的操作速度；改善了存取策略以提高 FLASH 的抗疲劳性, 同时也优化了碎片整理性能, 增加了数据压缩功能。需要注意的是, 当文件系统已满或接近满时, JFFS2 会大大放慢运行速度。这是因为垃圾收集的问题。相对于 ext2fs 而言, JFFS2 在嵌入式设备中更受欢迎。

嵌入式系统中采用 JFFS2 文件系统有以下优点：

- 支持数据压缩；
- 提供“损耗平衡”支持；
- 支持多种节点类型；
- 提高了对闪存的利用率, 降低了内存的消耗。

我们只需要在自己的嵌入式 Linux 中加入 JFFS2 文件系统并做少量的改动, 就可以使用 JFFS2 文件系统。通过 JFFS2 文件系统, 可以用FLASH 存储器来保存数据, 将 FLASH 存储器作为系统的硬盘来使用。可以像操作硬盘上的文件一样操作 FLASH 芯片上的文件和数据。同时系统运行的参数可以实时保存到 FLASH 存储器芯片中, 在系统断电后数据不会丢失。

作为一种 EEPROM, FLASH 可分为 NOR FLASH 和 NAND FLASH 两种主要类型。一片没有使用过的 FLASH 存储器, 每一位的值都是逻辑1。对 FLASH 的写操作就是将特定位的逻辑1 改变为逻辑0。而擦除就是将逻辑0 改变为逻辑1。 FLASH 的数据存储是以块 (Block) 为单位进行组织, 所以 FLASH 在进行擦除操作时只能进行整块擦除。

FLASH的使用寿命是以擦除次数进行计算的。一般在十万次左右。为了保证 FLASH 存储芯片的某些块不早于其他块到达其寿命, 有必要在所有块中尽可能地平均分配擦除次数, 这就是“损耗平衡”。JFFS2 文件系统是一种“追加式”的文件系统——新的数据总是被追加到上次写入数据的后面。这种“追加式”的结构就自然实现了“损耗平衡”。

4.2.4 YAFFS2

YAFFS（Yet Another Flash File System）文件系统是专门针对NAND 闪存设计的嵌入式文件系统。与JFFS2 文件系统不同, YAFFS2 主要针对使用NAND FLASH 的场合而设计。NAND FLASH 与NOR FLASH 在结构上有较大的差别。尽管JFFS2 文件系统也能应用于NAND FLASH, 但由于它在内存占用和启动时间方面针对NOR FLASH 的特性做了一些取舍, 所以对NAND来说通常并不是最优的方案。在嵌入式系统使用的大容量NAND FLASH中, 更多的采用YAFFS2文件系统。

4.2.5 Ramdisk

使用内存的一部份空间来模拟一个硬盘分区, 这样构成的文件系统就是ramdisk。将ramdisk 用作根文件系统在嵌入式Linux 中是一种常用的方法。因为在ram 上运行, 读写速度快；用gzip 算法进行压缩, 可节省存储空间。但它也有缺点：由于将内存的一部分用作ramdisk, 这部分内存不能再作其他用途；此外系统运行是更新的内容无法保存, 系统关机后内容将丢失。

4.3 文件系统的制作

4.3.1 Busybox 介绍

Busybox 是 Debian GNU/Linux 著名的Bruce Perens 首先开发, 主要使用在 Debian 的安装程序中。后来又有许多 Debian 开发者对Busybox 贡献力量, 其中包括著名的 Linus Torvalds。Busybox 最终编译成一个叫做 busybox 独立执行程序, 并且可以根据配置, 执行 ash shell 的功能, 包含几十个小应用程序：mini-vi 编辑器、系统不可或缺的 /sbin/init 程序, 以及其他诸如文件操作、目录操作、系统配置等等。这些都是一个正常的系统必不可少的。但如果把这些程序独立编译的话, 其规模在一个嵌入式系统中难以承受。BusyBox 具有全部这些功能, 大小也不过几百 K 左右。而且用户还可以根据自己的需要对BusyBox 的应用程序功能进行剪裁, 使 BusyBox 的规模进一步缩小。

BusyBox 支持多种体系结构, 它可以静态或动态链接glibc 或者uclibc 库, 以满足不同的需要。[1]

4.3.2 BusyBox 的编译

将下载的 BusyBox 软件包解压缩。进入解压目录, 执行“make menuconfig”, 仿照内核配置编译过程：

- 在Build Option 菜单下, 选择静态库编译方式[2], 并设定交叉编译器；
- Installation Options 配置中, 自定义安装路径。编译后的文件系统以这个路径为起点；
- 用户可以根据需要对文件系统的功能选项进行配置, 这样可以减少文件系统的大小, 以节省存储空间。

配置完成后便可对BusyBox 进行编译(make)和安装(make install)。安装完毕, 在安装目录下可以看到 bin、 sbin 和 usr(usr 目录是否存在, 取决于配置的安装选项)这些目录。在这些目录里可以看到许多应用程序的符号链接, 这些符号链接都指向 busybox。

[1] BosyBox本身不带glibc/uclibc。用户须自行为系统配置这些库并安装在/lib目录下。没有库支持的基本文件系统只能运行静态链接的外部程序

[2] 如文件系统带有共享库, 也可以采用动态链接方式

4.3.3 配置文件系统

- 创建etc 目录, 在etc 下建立inittab、rc、motd三个文件。

  /etc/inittab

  ==========

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/sh
::once:/usr/sbin/telnetd -l /bin/login
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

  此文件由系统启动程序init 读取并解释执行。以# 开头的行是注释行。

  /etc/rc

  ==========

```
#!/bin/sh
hostname x210
mount -t proc proc /proc
/bin/cat /etc/motd
```

  此文件要求可执行属性, 用命令“chmod +x rc”修改其属性。rc**文件和其他脚本文件**(.sh)**第一行的**# **不是注释**。

  /etc/motd

  ==========

```
  Welcome to
  ==============================
          ARM-LINUX WORLD
  ==============================
  x210v3 @ S5PV210/Cortex-A8
  ported by FANG YUAN
```

  此文件内容随意, 由/etc/rc 调用打印在终端上。

  在etc 目录下再创建init.d 目录, 并将 /etc/rc 向 /etc/init.d/rcS 做符号链接。此文件为 inittab 指定的启动脚本：

```
  $ mkdir init.d
  $ cd init.d
  $ ln -s ../rc rcS
```

- 创建dev 目录, 并在该目录下建立必要的设备：

```
$ mknod console c 5 1
$ mknod null c 1 3
$ mknod zero c 1 5
```

- 建立proc 空目录, 供proc 文件系统使用。
- 建立lib 目录, 将交叉编译器链接库路径下的下面几个库复制到lib目录：
  ld-2.10.1.so, libc-2.10.1.so, libm-2.10.1.so 并做相应的符号链接：
  ln –s ld-2.10.1.so ld-linux.so.3
  ln –s libc-2.10.1.so libc.so.6
  ln –s libm-2.10.1.so libm.so.6

  如果busybox以静态链接方式编译, 没有这些库, 不影响系统正常启动, 但会影响其他动态链接的程序运行。

至此文件系统目录构造完毕。从根目录看下去, 应该至少有下面几个目录:
bin dev etc lib lost+found mnt proc sbin
它们是下面制作文件系统的基础。

4.3.4 制作ramdisk 文件映像

配置内核时, ramdisk 要求在“Block devices→”中选中“RAM block device support”, 并设置适当的ramdisk 大小。在“General setup” 设置分支里选中“Initial RAM filesystem and RAM disk (initramfs/initrd) support”。

为了生成并修改ramdisk, 需要在主机上创建一个空文件并将它格式化成ext2fs 文件系统映像。格式化后的文件就可以像普通文件系统一样在主机上进行挂载和卸载。挂载后可以进行正常的文件和目录操作, 卸载后, 如果原映像文件仍然存在, 则更新到卸载之前的操作内容。[3]

```
$ dd if=/dev/zero of=ramdisk_img bs=1k count=8192
$ mke2fs ramdisk_img
$ mount ramdisk_img
$ (复制文件系统目录和文件，及其他一些必要的设置)
$ umount /mnt/ramdisk
```

注意, 此时虽然ramdisk_img从形式上看和普通文件没什么区别, 但它却是一个完整独立的文件系统映像。逻辑上, 它和u盘、SD卡甚至硬盘是等同的。

内核支持压缩方式的ramdisk, 以节省FLASH 占用空间。通常用下面的方式压缩和解压(mount之前必须解压):

```
$ gzip ramdisk_img
$ gunzip ramdisk_img.gz
```

bootloader 通过bootargs 向内核传递信息，指示它挂载ramdisk 作为根文件系统。同时ramdisk 的映像文件也应装入内存的对应位置:

```
x210 # setenv ramdisk root=/dev/ram rw initrd=0x40000000,8M
x210 # setenv bootargs console=ttySAC2,115200 $ramdisk
x210 # tftp 0x40000000 ramdisk_img.gz
```

[3] 挂载ext2fs 文件系统需要root权限。实验室已在/etc/fstab 中设置指定文件和目录, 允许普通用户将文件ramdisk_img 挂载到/mnt/ramdisk

4.3.5 制作init_ramfs

也可以将之前制作的根文件系统做进内核映像中，使内核成为一个完整的独立系统。

首先，将根文件系统用cpio 打包并压缩:

```
$ find /mnt/ramdisk -print |cpio -H newc -o |gzip -9 > ramdisk.cpio.gz
```

这种做法不要求制作独立的文件系统。之所以这里使用“/mnt/ramdisk”，是因为之前恰好做过一个完整的根文件系统并挂载到这个目录下。

将生成的文件ramdisk.cpio.gz 复制到内核源码目录,并在内核配置选项中, “Initial RAM filesystem and RAM disk”下面的“Initramfs source file(s)” 写上这个文件名。[4]

重新编译内核，将该文件编入内核文件zImage，并复制到tftp目录下。

在目标板中加载该内核，启动。

4.4 实验内容

- 编译BusyBox, 以BusyBox 为基础，构建一个适合的文件系统;
- 制作ramdisk 文件系统映像, 用你的文件系统启动到正常工作状态;
- * 研究NFS 作为根文件系统的启动过程。

4.5 实验报告要求

- 讨论你的嵌入式系统所具备的功能；
- 比较romfs、ext2fs/ext3fs、JFFS2 等文件系统的优缺点；
- * 考虑制作一个 YAFFS2 文件系统作为系统的根文件系统。

[4]解压cpio: cpio --no-absolute-filenames -imdv < foo.cpio

第五章　图形用户接口

5.1　实验目的

- 了解嵌入式系统图形界面的基本编程方法
- 学习图形库的制作

5.2　原理概述

5.2.1　Frame Buffer

显示屏的整个显示区域，在系统内会有一段存储空间与之对应。通过改变该存储空间的内容达到改变显示信息的目的。该存储空间被称为 Frame Buffer， 或显存。显示屏上的每一点都与 Frame Buffer 里的某一位置对应。所以，解决显示屏的显示问题，首先需要解决的是 Frame Buffer 的大小以及屏上的每一像素与 Frame Buffer 的映射关系。

按照显示屏的性能或显示模式区分，显示屏可以以单色或彩色显示。单色用1 位来表示(单色并不等于黑与白两种颜色，而是说只能以两种颜色来表示。通常取允许范围内颜色对比度最大的两种颜色)。彩色有2、4、8、16、24、32等位色。这些色调代表整个屏幕所有像素的颜色取值范围。如：采用8 位色/像素的显示模式，显示屏上能够出现的颜色种类最多只能有2^8 种。究竟应该采取什么显示模式，首先必须根据显示屏的性能，然后再由显示的需要来决定。这些因素会影响 Frame Buffer 空间的大小,因为 Frame Buffer 空间的计算大小是以屏幕的大小和显示模式来决定的。另一个影响 Frame Buffer 空间大小的因素是显示屏的单/双屏幕模式。

单屏模式表示屏幕的显示范围是整个屏幕。这种显示模式只需一个 Frame Buffer 来存储整个屏幕的显示内容，并且只需一个通道来将 Frame Buffer 内容传输到显示屏上 (Frame Buffer 的内容可能需要被处理后再传输到显示屏)。双屏模式则将整个屏幕划分成两部分。它有别于将两个独立的显示屏组织成一个显示屏。单看其中一部分，它们的显示方式是与单屏方式一致的，并且两部分同时扫描，工作方式是独立的。这两部分都各自有 Frame Buffer， 且他们的地址无需连续(这里指的是下半部的 Frame Buffer 的首地址无需紧跟在上半部的地址末端)，并且同时具有独立的两个通道将 Frame Buffer 的数据传输到显示屏。

Frame Buffer 通常就是从内存空间分配所得，并且它是由连续的字节空间组成。由于屏幕的显示操作通常是从左到右逐点像素扫描、从上到下逐行扫描，直到扫描到右下角，然后再折返到左上角，而 Frame Buffer 里的数据则是按地址递增的顺序被提取，当 Frame Buffer 里的最后一个字节被提取后,再返回到 Frame Buffer 的首地址，所以屏幕同一行上相邻的两像素被映射到Frame Buffer 里是连续的，某一

行的最末像素与它下一行的首像素反映在 Frame Buffer 里也是连续的，并且屏幕上最左上角的像素对应 Frame Buffer 的第一单元空间，最右下角的像素对应 Frame Buffer 的最后一个单元空间。

5.2.2 Frame Buffer与色彩

计算机反映自然界的颜色是通过 RGB (Red– Green– Blue) 值来表示的。如果要在屏幕某一点显示某种颜色，则必须给出相应的 RGB 值。 Frame Buffer 为屏幕提供显示的内容，就必须能够从 Frame Buffer 里得到每一个像素的 RGB 值。像素的 RGB 值可以直接从 Frame Buffer 里得到，或是从是调色板间接得到(此时 Frame Buffer 存放的并不是RGB 值，而是调色板的索引值。通过索引值可以获得调色板的 RGB 值)。

Frame Buffer 是由所有像素的 RGB 值或 RGB 值的部分位(RGB 由红、绿、蓝各8 位组成，共24 位，称为真彩色。由于某些显示屏的数据线有限，只有16 条数据线或更少，这时只能取 R、 G、 B 部分位与数据线对应)所组成。例如，16 位/像素模式下， Frame Buffer 里的每个单元为16 位，每个单元代表一个像素的RGB 值(RGB565)，如下图。

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R	R	R	R	R	G	G	G	G	G	G	B	B	B	B	B

有了以上的分析，就可以用下面的计算公式

$$FrameBufferSize = \frac{Width \times Height \times Bit\_per\_Pixel}{8}$$

计算 Frame Buffer 的大小(以字节为单位)。

5.2.3 LCD控制器

在 Frame Buffer 与显示屏之间还需要一个中间件，该中间件负责从 Frame Buffer 里提取数据，进行处理，并传输到显示屏上。

处理器内部集成LCD 控制器, 将 Frame Buffer 里的数据传输到 LCDC 的内部, 然后经过处理，输出数据到 LCD 的输入引脚上。

本实验系统使用的是32位LCD，像素分辨率 8000×480。 相关驱动程序在$(KERNEL-PATH)/drivers/ video/ samsung/ s3cfb.c中。

5.2.4 Frame Buffer操作

Frame Buffer设备是/dev/fb (它通常是字符设备/dev/fb0 的符号链接，该设备主设备号是29，次设备号是0)。了解这个设备的参数可以通过 FBIOGET_FSCREENINFO、 FBIOGET_VSCREENINFO 命令，如：

```
......
struct fb_var_screeninfo vinfo; // #include <linux/fb.h>
......
fd = open ( "/dev/fb", O_RDWR );
......
ioctl( fd, FBIOGET_VSCREENINFO, &vinfo )
```

可以获得显示器色位、分辨率等信息 (vinfo.bits_per_pixel、 vinfo.xres、 vinfo.yres)。

获取 Frame Buffer 缓冲区首地址的系统调用是

```
unsigned char *fbp = 0;
......
fbp = (unsigned char *)mmap(0, screensize,\
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

screensize 是根据显示器信息计算出的缓冲区大小，通过 mmap() 函数获得的缓冲区首地址。从该首地址开始、以 screensize 为大小的范围即是显示缓冲区的内存映射地址。如果采用RGB–24位色，在坐标(x, y)处画一个红点，可以用下面的方法：

```
// draw_point(int x, int y)

    offset = (y * vinfo.xres + x) * vinfo.bits_per_pixel / 8;
    *(unsigned char *)(fbp + offset + 0) = 255;
    *(unsigned char *)(fbp + offset + 1) = 0;
    *(unsigned char *)(fbp + offset + 2) = 0;
    ......
```

如果是16位色(RGB565)，须根据格式要求将RGB压缩到16位，再填充对应字节：[1]

```
// draw_point(int x, int y)

    offset = (y * vinfo.xres + x) * vinfo.bits_per_pixel / 8;
    color = (Red << 11) | ((Green << 5) & 0x07E0) | (Blue & 0x1F);
    *(unsigned char *)(fbp + offset + 0) = color & 0xFF;
    *(unsigned char *)(fbp + offset + 1) = (color >> 8) & 0xFF;
    ......
```

将显示缓冲区清零， memset(fbp, 0, screensize)， 即可以实现清屏(黑色)操作。

使用完毕应通过 munmap() 释放显示缓冲区。

5.3 实验内容

5.3.1 实现基本画图功能

在 Frame Buffer 基础上编写画点、画线的API函数，供应用程序调用，实现任意曲线的画图功能。

[1] 不同位端(endian)的处理器，移位及高低字节顺序有所不同。

5.3.2 合理的软件结构

将调用设备驱动的基本 API函数独立地构成一个函数库，为用户程序屏蔽底层硬件信息，直接提供一些简单的画图调用。函数库可以是独立编译后的“.o”文件或由归档管理器 ar 生成的库文件,或是将“.o”文件链接而成的共享库“.so”。(例如，静态库文件名为 libXXX.a，或共享库文件名 libXXX.so, 编译链接时可用“–l XXX” 选项)

5.4 实验报告要求

- 总结 Frame Buffer 的操作方法；
- 探讨软件结构的层次关系；
- 思考：如果一帧显示数据的计算量很大，连续图像的刷新、显示将消耗比较多的时间，此时如何较好地实现连续画面的动态显示？

第六章　触摸屏移植

6.1　实验目的

- 了解嵌入式系统中触摸屏的原理;
- 学习开源软件的移植方法。

6.2　Linux系统的触摸屏支持

触摸屏是目前最简单、方便、自然的一种人机交互方式，在嵌入式系统中得到了普遍的应用。触摸屏库除了用于支持图形接口环境以外，它本身也可以作为触摸屏应用软件编程的学习范例。

6.2.1　内核配置

Linux操作系统内核支持多种触摸屏设备。在Linux内核源码配置界面中，找到并选中正确的驱动，将其编入内核。(Device Drivers → Input device support → Touchscreens → x210 touchscreen driver)

内核中，触摸屏可以是独立驱动，也可以加入Event interface。后者通过/dev/input/eventX 设备存取输入设备的事件。建议在内核配置中也选中Event interface。

6.2.2　触摸屏库tslib

下载触摸屏库tslib-1.0.tar.bz2，并将其解压到工作目录。

进入解压目录，依次执行下面的操作:

```
$ export CC=arm-none-linux-gnueabi-gcc
$ ./autogen.sh
$ ./configure --host=arm-linux --prefix=/path/to/install
$ make install
```

以上过程注意事项:

- 必须事先设置好环境变量PATH，加入交叉编译器路径，否则在“make”中交叉编译命令不能正确执行;
- --prefix 选项用于“make install”的安装目录，请使用一个拥有写权限的目录，编译完成后会将结果集中存放于此。如果不指定安装目录，默认的安装目录一般是/usr/local，普通用户没有写权限，此时不宜用“make install”命令，可以仅用“make”命令，结果分散在src/.libs(库)和plugins/.libs(插件)中。

- 编译过程中可能会有错误提示“undefined reference to ‘rpl_malloc’ ”, 可在configure 之前设置环境变量“export ac_cv_func_malloc_0_nonnull =yes”。

正确编译后，会在安装目录下新生成etc、bin、lib、include 四个子目录。etc里的ts.conf 是库的配置文件，bin下面的可执行程序包括触摸屏校准和测试工具，lib里是触摸屏的动态链接库和模块插件，include 下面的tslib.h 可用于基于触摸屏库的应用程序二次开发。

6.2.3 触摸屏库的安装和测试

将前面产生的文件按目录对应关系分别复制到目标系统的相应目录中，编辑ts.conf 文件，去掉“# module_raw input”前面的注释。按下面的方式设置环境变量:

- export TSLIB_TSDEVICE=/dev/input/event2

  触摸屏设备文件或Event interface设备文件。eventX的主设备号是13，次设备号从64开始，可通过/proc/bus/input/devices 文件获知触摸屏的次设备号。

- export TSLIB_CONFFILE=/etc/ts.conf

  触摸屏库的配置文件。一般需要保留这几个模块:

  - variance, 用于过滤AD转换器的随机噪声。它假定触点不可能移动太快，其阈值(距离的平方)由参数delta 指定。
  - dejitter, 去除抖动，保持触点坐标平滑变化。
  - linear, 线性坐标变换。

- export TSLIB_PLUGINDIR=/lib/ts

  插件模块文件(.so)

- export TSLIB_CONSOLEDEVICE=none

  终端设备，缺省的是/dev/tty，此处不需要。

- export TSLIB_FBDEVICE=/dev/fb0

  帧缓冲设备文件

- export TSLIB_CALIBFILE=/etc/pointercal

  校准文件。早期触摸屏由于工艺原因，每台机器的坐标读取数值差异较大，使用前必须通过校准工具将触摸屏和液晶屏坐标进行校准，产生一个校准文件。

以上准备工作就绪后，尝试执行/bin/ts_test。

6.3 实验内容

完成触摸屏移植。

分析ts_test.c，利用触摸屏库编写一个能进行触摸屏操作的应用程序，功能自定。

第七章　Qt/Embedded移植

7.1　实验目的

- 了解嵌入式GUI—Qt/E 软件开发平台的构架；
- 学习Qt/E 移植的基本步骤与方法。

7.2　Qt/E 介绍

Qt/Embedded 是跨平台的c++ 图形用户界面(GUI)工具包，它是著名的Qt 开发商 TrollTech 发布的面向嵌入式系统的Qt版本。Qt 是目前 KDE 等项目使用的 GUI 支持库，许多基于 Qt 的 X-Window 程序可以非常方便地移植到嵌入式 Qt/Embedded 版本上。自从 Qt/Embedded 发布以来，有许多嵌入式 Linux 开发商利用 Qt/Embedded 进行了嵌入式 GUI 的应用开发。

Qt/Embedded 注重于能给用户提供精美的图形界面所需的所有元素，而且其开发过程是基于面向对象的编程思想，并且 Qt/Embedded 支持真正的组件编程。

TrollTech 公司所发布的面向嵌入式系统的 QT/E 版本提供源代码。用户必须针对自己的嵌入式硬件平台进行裁剪、编译和移植。尽管 Qt/Embedded 可以裁剪到630K，但它对硬件平台具有较高的要求。目前 Qt/Embedded 库主要针对手持式信息终端。

本实验主要完成 Qt/Embedded 在嵌入式实验平台上的移植。

7.2.1　Qt/E软件包结构

Qt/E 系统源码一般包括以下几个软件包：

- 触摸屏支持库tslib.tar.bz2；
- Makefile 生成工具tmake-1.11.tar.gz，它主要由一些脚本程序组成；
- 开发平台编译环境库及工具程序qt-x11-2.3.2.tar.gz；
- 目标平台Qt/Embedded 核心库。用户可到 TrollTech 主页上[1]下载Qt/Embedded的某个版本的源代码。本实验提供qt-embedded-2.3.7.tar.gz；
- Qt桌面环境qtopia-free-1.7.0.tar.gz。

[1] ftp://ftp.trolltech.com/qt/source

7.3 Qt/E 编译

7.3.1 设置环境

为了以下编译过程顺利进行，先将上面的压缩文件全部解压。假设各自被解压到下面的目录(一般Qt/E 的移植过程也是按这个顺序进行操作的)：

1. ～/work/tslib
2. ～/work/tmake-1.11
3. ～/work/qt-2.3.2
4. ～/work/qt-embedded-2.3.7
5. ～/work/qtopia-1.7.0

在编译Qt/Embedded时，用户在PC机上应对编译时所需的环境变量进行设置，这些设置的主要参数包括：

- QTDIR — Qt解压后的所在的目录
- LD_LIBRARY_PATH — Qt共享库存放的目录
- QPEDIR — qtopia解压后的所在的目录
- TMAKEPATH — tmake编译工具的路径
- TMAKEDIR－tmake编译工具的目录
- PATH — 交叉编译工具arm-linux-gcc的路径

根据上面的解压目录，环境变量应作如下设置：

```
$ export QTDIR=~/work/qt-embedded-2.3.7
$ export QPEDIR=~/work/qtopia-1.7.0
$ export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
$ export TMAKEDIR=~/work/tmake-1.11
$ export TMAKEPATH=~/work/tmake-1.11/lib/qws/linux-arm-g++
$ export PATH=~/work/tmake-1.11/bin:/usr/local/arm-linux/bin:$PATH
```

7.3.2 编译过程

编译触摸屏库

Qt/Embedded 支持鼠标和键盘的操作，但现在许多嵌入式系统都使用触摸屏作为输入设备，所以用户必须将触摸屏的相关操作编译成共享库或静态库。

触摸屏库的编译过程可参考第六章内容。

将编译完成后的库复制到qt-embedded-2.3.7目录：

```
$ cp -a src/.libs/* ../qt-embedded-2.3.7/lib
$ cp -a plugins/.libs/*.so ../qt-embedded-2.3.7/lib
```

编译qt-x11工具

在～/work/qt-2.3.2 下执行

```
$ export QTDIR=~/work/qt-2.3.2
$ export QTEDIR=~/work/qt-embedded-2.3.7
$ export QPEDIR=~/work/qtopia-free-1.7.0
$ export PATH=$QTDIR/bin:$PATH
$ export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
$ ./configure -no-opengl -no-xft
$ make
$ make -C tools/qvfb
$ mv tools/qvfb/qvfb bin
$ cp bin/uic $QTEDIR/bin
```

生成的uic 和moc 作为Qt 应用程序的转换工具，qvfb 是Qt 开发平台的仿真工具。

编译Qt/Embedded

先将补丁文件里的文件替换掉源码包里的对应文件(主要是针对目标平台触摸屏代码和编译器的设置)，然后在～/work/qt-embedded-2.3.7目录下执行

```
$ export QTDIR=~/work/qt-embedded-2.3.7
$ cp ~/work/qtopia-free-1.7.0/src/qt/qconfig-qpe.h src/tools
$ ./configure -xplatform linux-arm-g++ -qconfig qpe -depths 16 -no-qvfb
$ make sub-src
```

这一步完成后，生成目标平台的Qt 核心库libqte.so* 等等。

编译Qtopia

在～/work/qtopia-free-1.7.0/src 下面执行

```
$ ./configure -platform linux-arm-g++
$ make
```

编译完成后会产生apps、bin、doc、etc、help、include、plugins等目录及目录下的文件。至此，编译过程基本结束。

7.3.3 Qt/Embedded的安装

准备将待安装的文件放在一个独立的目录下。新建一个目录～/work/qpe，将 qtopia-free-1.7.0/src 下面的 apps、 bin、 etc、 plugins、 i18n、 lib、 pics 这些目录连同下面的子目录和文件复制到该目录下，同时将 qt-embedded-2.3.7/lib 下面的库连同字体目录也复制到 qpe/lib 下(注意保持原来的目录结构)。由于字体文件比较大，可适当删除一些不常用的字体库，保留 *.qpf 文件和 fontdir 文件。另外，还要将触摸屏配置文件～/work/tslib/etc/ts.conf 复制到 etc 目录。

将整个qpe 目录复制到目标系统文件系统的 /usr 目录下，再为 qpe 建立一个启动脚本 (/usr /bin /qpe.sh)：

```
$ export QTDIR=/usr/qpe
$ export QPEDIR=/usr/qpe
$ export LANG=zh_CN
$ export LD_LIBRARY_PATH=/usr/qpe/lib:$LD_LIBRARY_PATH
$ export QT_TSLIBDIR=/usr/qpe/lib
$ export TSLIB_CONFFILE=/usr/qpe/etc/ts.conf
$ export TSLIB_PLUGINDIR=/usr/qpe/lib
$ export QWS_MOUSE_PROTO=TPanel:/dev/touchscreen/ucb1x00
$ export KDEDIR=/usr/qpe
$ /usr/bin/ts_calibrate
$ /usr/qpe/bin/qpe &> /dev/null
```

将Qt/E 系统与 BusyBox 结合，按第 四 章的方法重新制作文件系统映像; 根据需要, 修改启动脚本inittab(或rc)的启动执行步骤。

7.4 实验要求

完成一个Qt/Embedded 系统的编译和安装。

****

[附] 编译过程中的一些错误及修正

由于编译器版本及源代码规范性等方面的原因，对源码软件编译过程中经常会碰到一些编译错误。对这些编译错误，最好能根据编译器给出的错误提示，找到出错的地方，有针对性地加以修正。下面是在编译Qt/E 过程中可能出现的一些错误及解决办法：

1. `qt-2.3.2`: include/qvaluestack.h:57: 错误.....
   修改include/qvaluestack.h 第57 行，将
   remove(this->fromLast()); 改为
   this->remove(this->fromLast());
2. `qt-embedded-2.3.7`: include/qwindowsystem_qws.h:229: error: 'QWSInputMethod' has not been declared
   在include/qwindowsystem_qws.h 里加上类声明：
   class QWSInputMethod;
   class QWSGestureMethod;
3. `qt-embedded-2.3.7`: *** [allmoc.o] 错误1
   向前追溯出错位置，在include/qsortedlist.h 中，将第51行改为：
   ~QSortedList() { this->clear(); }
   同样性质的"错误"还有很多，取决于编译器版本。这里不再一一列举。
4. `qtopia-free-1.7.0`: Makefile:10: *** 遗漏分隔符。停止。
   修改src/3rdparty/libraries/libavcodec/Makefile，删除 10、 14、 18 行的–e

5. `qtopia-free-1.7.0`: libraries/qtopia/backend/event.cpp:404: error: ISO C++ says that these are ambiguous,.....
   c++对操作符"<="理解有歧义。可将局部变量i 声明为int。
6. `qtopia-free-1.7.0`: libraries/qtopia/qdawg.cpp:243: error: extra qualification 'QDawgPrivate::.....
   去掉类定义中的本类声明。
7. `qtopia-free-1.7.0`: libavformat/img.c:723: error: static declaration of 'pgm_iformat' follows non-static declaration
   函数或变量属性声明冲突。
8. `qtopia-free-1.7.0`: 对'__cxa_guard_release'未定义的引用
   出现在链接阶段。到编译器路径下找到该函数或变量所属的库(libstdc++.so)，在编译qt-embedded-2.3.7 时加上库的链接。

第八章　音频接口程序设计

8.1　实验目的

- 了解音频编、解码的作用和工作原理；
- 学习Linux 系统的音频接口编程方法。

8.2　接口介绍

S5PV210 片内集成了音频子系统，支持多种音频编解码，包含IIS总线接口。IIS总线通过串行方式传输数字音频、调制/解调, 实现音频输入/输出、寄存器控制和状态信息获取等, 可与多种音频接口芯片连接。

本实验系统使用的CODEC芯片为WM8976。它采用 Δ–Σ 转换工作原理进行模拟/数字之间的转换。该芯片驱动程序在

$(KERNELPATH)/drivers/sound/soc/codecs/wm8976.c

8.3　原理概述

内核编译时需要有声音支持。

8.3.1　OSS

Linux 系统的音频驱动主要有两类: OSS(Open Sound System)和ALSA(Advanced Linux Sound Architecture)。其中OSS 主要出现在早期的Linux版本中。与OSS 相关的设备节点主要有两个：

- /dev/dsp(主设备号14，次设备号3)，负责音频数据的输入 (A/D 转换)和输出 (D/A 转换)、工作方式设置(采样/输出频率、通道数、数据格式等等)；
- /dev/mixer(主设备号14，次设备号0)，混音及音量设置、高低音等等。

请参考$(KERNEL_PATH)/include/linux/soundcard.h 中的说明完成音频接口设置。(如，设置双声道，channels=2; ioctl(fd, SNDCTL_DSP_CHANNELS, &channels);) 编写应用程序，实现简单的录音和放音。记录模拟/数字音频转换结果。

通过/dev/dsp(或/dev/audio，主设备号13，次设备号4)设置给定的采样/输出模式。常用的设置命令有：

- SNDCTL_DSP_RESET
- SNDCTL_DSP_SPEED，采样/输出率，如8000Hz、44100Hz、48000Hz等。$\Delta-\Sigma$转换器通过晶振的有限个分频得到采样率/输出率，因此不能直接实现任意频率的采样/输出。
- SNDCTL_DSP_SAMPLESIZE，采样值的数据位大小。
- SNDCTL_DSP_CHANNELS，通道数，mono(1)或stereo(2)。
- SOUND_PCM_WRITE_CHANNELS，输出通道数，通常和输入通道数一致。
- SNDCTL_DSP_SETFRAGMENT，缓冲数据块大小。
- SNDCTL_DSP_SETFMT、SNDCT_DSP_GETFMTS，设置/获取数据格式。这些格式包括 μ–律 (AFMT_MU_LAW)、 A–律 (AFMT_A_LAW)、 无符号8位 (AFMT_U8)、 带符号8位(AFMT_S8)、16位大端或小端模式 (AFMT_S16_LE、 AFMT_U16_BE)等等。

对混音器(/dev/mixer)的常用命令有：

- SOUND_MIXER_NRDEVICES，获取设备数量。
- SOUND_MIXER_VOLUME，总音量设置。音量取值范围是0~100。
- SOUND_MIXER_BASS、SOUND_MIXER_TREBLE，低音、高音设置。
- SOUND_MIXER_PCM、 SOUND_MIXER_LINE、 SOUND_MIXER_MIC 等，对各音源音量的独立设置。
- SOUND_MIXER_IGAIN，输入增益。
- SOUND_MIXER_OGAIN，输出增益。

8.3.2 ALSA

现在的Linux发行版更多的采用ALSA音频驱动。与OSS不同的是, ALSA应用程序通过ALSA API完成对设备的操作，不再使用open,close,ioctl,read,write 等低级系统调用。因此ALSA 应用程序中看不到设备文件，有的只是对ALSA 函数的调用。编译ALSA程序需要链接asound 库。

下面是一些API的例子：

```
/* Allocate the snd_pcm_hw_params_t structure on the stack. */
snd_pcm_hw_params_t *hwparams;
snd_pcm_hw_params_alloca(&hwparams);

/* 打开 PCM 设备 */
char *pcm_name = "plughw:0,0";
snd_pcm_open(&pcm_handle, pcm_name, SND_PCM_STREAM_PLAYBACK, 0);

/* 初始化 PCM 参数*/
snd_pcm_hw_params_any(pcm_handle, hwparams);

/* PCM 命令集的形式是
```

```
            snd_pcm_hw_params_can_<capability>
            snd_pcm_hw_params_is_<property>
            snd_pcm_hw_params_get_<parameter>
        – 一些重要的参数，包括缓冲区大小、通道数、采样格式、速率等，可以通过
            snd_pcm_hw_params_set_<parameter>调用实现

        */

        /* 设置数据格式：16bit-signed-little-endian */
        snd_pcm_hw_params_set_format(pcm_handle, hwparams, SND_PCM_FORMAT_S16_LE);

        /* stereo */
        snd_pcm_hw_params_set_channels(pcm_handle, hwparams, 2);

        /* 设置采样率 44.1kHz */
        snd_pcm_hw_params_set_rate(pcm_handle, hwparams, 44100, 0);

        /* 将数据写入设备 (Digital to Analogue)，函数返回实际写入的帧数 */
        snd_pcm_write(pcm_handle, buffer, num_of_frames);


        /* 对混音器操作，通过 snd_mixer_<parameter/property> 一组指令完成 */
```

8.4 实验内容

根据内核配置，选择适当的音频驱动，实现数字音频的采集与回放。

8.5 实验报告要求

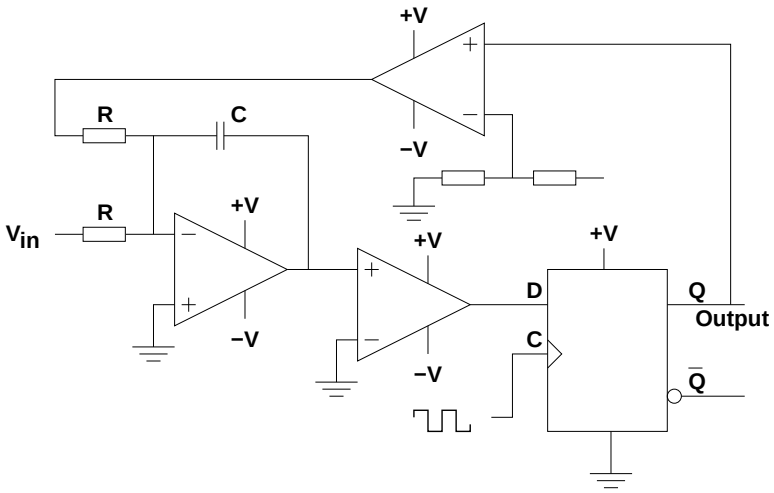
- 将实验采集的数据与信号源产生的实际信号对比，将输出的预期信号与示波器测量到的信号对比，分析产生差异的原因；
- 思考：如果在信号采集过程中还包含了数据处理工作，如何保证信号的连续性?

[附] $\Delta - \Sigma$ AD转换器原理

Delta-Sigma ($\Delta - \Sigma$) ADC

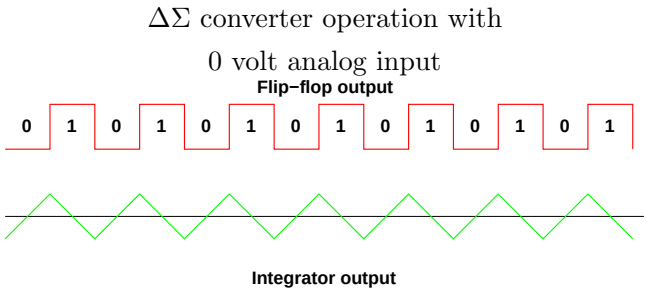
One of the more advanced ADC technologies is the so-called delta-sigma, or $\Delta - \Sigma$ (using the proper Greek letter notation). In mathematics and physics, the capital Greek letter delta (Δ) represents difference or change, while the capital letter sigma (Σ) represents summation: the adding of multiple terms together. Sometimes this converter is referred to by the same Greek letters in reverse order: sigma-delta, or $\Sigma - \Delta$.

In a $\Sigma - \Delta$ converter, the analog input voltage signal is connected to the input of an integrator, producing a voltage rate-of-change, or slope, at the output corresponding to input magnitude. This ramping voltage is then compared against ground potential (0 volts) by a comparator. The comparator acts as a sort of 1-bit ADC, producing 1 bit of output ("high" or "low") depending on whether the integrator output is positive or negative. The comparator's output is then latched through a D-type flip-flop clocked at a high frequency, and fed back to another input channel on the integrator, to drive the integrator in the direction of a 0 volt output. The basic circuit looks like this:

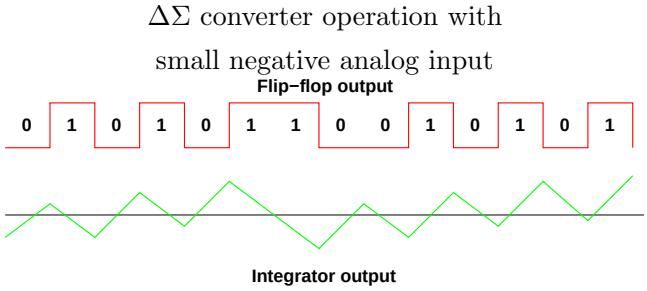


The leftmost op-amp is the (summing) integrator. The next op-amp the integrator feeds into is the comparator, or 1-bit ADC. Next comes the D-type flip-flop, which latches the comparator's output at every clock pulse, sending either a "high" or "low" signal to the next comparator at the top of the circuit. This final comparator is necessary to convert the single-polarity 0V/5V logic level output voltage of the flip-flop into a +V/-V voltage signal to be fed back to the integrator.If the integrator output is positive, the first comparator will output a "high" signal to the D input of the flip-flop. At the next clock pulse, this "high" signal will be output from the Q line into the noninverting input of the last comparator. This last comparator, seeing an input voltage greater than the threshold voltage of 1/2 +V, saturates in a positive direction, sending a full +V signal to the other input of the integrator. This +V feedback signal tends to drive the integrator output in a negative direction. If that output voltage ever becomes negative, the feedback loop will send a corrective signal (-V) back around to the top input of the integrator to drive it in a positive direction. This is the delta-sigma concept in action: the first comparator senses a difference (Δ) between the integrator output and zero volts. The integrator sums (Σ) the comparator's

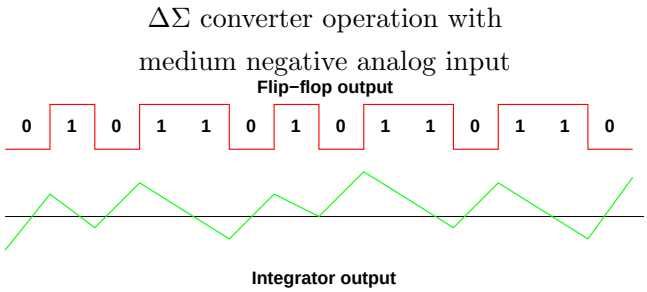
output with the analog input signal. Functionally, this results in a serial stream of bits output by the flip-flop. If the analog input is zero volts, the integrator will have no tendency to ramp either positive or negative, except in response to the feedback voltage. In this scenario, the flip-flop output will continually oscillate between "high" and "low," as the feedback system "hunts" back and forth, trying to maintain the integrator output at zero volts:



If, however, we apply a negative analog input voltage, the integrator will have a tendency to ramp its output in a positive direction. Feedback can only add to the integrator's ramping by a fixed voltage over a fixed time, and so the bit stream output by the flip-flop will not be quite the same:

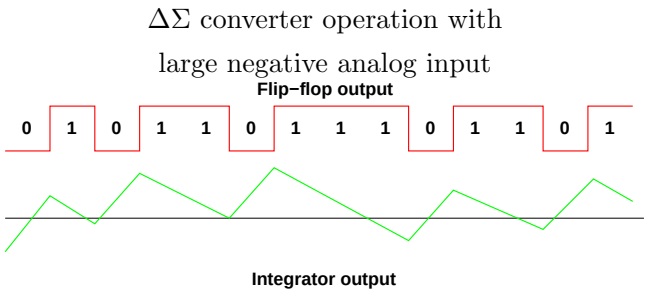


By applying a larger (negative) analog input signal to the integrator, we force its output to ramp more steeply in the positive direction. Thus, the feedback system has to output more 1's than before to bring the integrator output back to zero volts:



As the analog input signal increases in magnitude, so does the occurrence of 1's in the digital output of the flip-flop:

A parallel binary number output is obtained from this circuit by averaging the serial stream of bits together. For example, a counter circuit could be designed to collect the total number of 1's output by



the flip-flop in a given number of clock pulses. This count would then be indicative of the analog input voltage. Variations on this theme exist, employing multiple integrator stages and/or comparator circuits outputting more than 1 bit, but one concept common to all $\Delta - \Sigma$ converters is that of oversampling. Oversampling is when multiple samples of an analog signal are taken by an ADC (in this case, a 1-bit ADC), and those digitized samples are averaged. The end result is an effective increase in the number of bits resolved from the signal. In other words, an oversampled 1-bit ADC can do the same job as an 8-bit ADC with one-time sampling, albeit at a slower rate.

第九章　内核和设备驱动编程

9.1　实验目的

- 学习Linux操作系统下内核程序的编写和应用；
- 学习可编程接口芯片的编程控制方法。

9.2　内核程序结构

9.2.1　内核模块

一个内核模块至少包括两个函数:

- 由module_init(function) 声明，缺省函数名为int init_module()，在装载时被调用
- 由module_exit(function) 声明，缺省函数名为void cleanup_module()，卸载模块时被调用

2.6内核版本编译内核模块通常使用下面的Makefile：

```
# Makefile (!not makefile)
ifneq ($(KERNELRELEASE),)
obj-m       := mod_name.o
else
KDIR        := /lib/modules/`uname -r`/build
PWD         := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

执行命令“make”后生成以“.ko”为后缀的文件名。

2.4版本下，编译一个内核模块可以用如下命令

```
$ gcc -Wall -DMODULE -D__KERNEL__ -DLINUX -c -O2 mod_name.c\
   -I /usr/src/linux-2.4.26/include
```

生成后缀名“.o”的文件，它的作用和用法跟2.6内核模块的“.ko”完全一样。

注意，内核模块的头文件路径通常不是应用程序的头文件路径 /usr/include，因此 gcc 编译命令不能默认地找到它们，需要用 “-I”选项为其指定路径。PC机上，这个路径通常是 /usr/src/linux-x.xx/include，

/lib/modules/$(VERSION)/build/include也给出了它的链接。系统未安装内核源码头文件将无法正常编译通过。

内核模块不是一个可独立运行的程序，它要依赖操作系统或应用程序对其调用才能够实现它的功能。内核的加载可以通过系统命令insmod 完成:

```
$ insmod mod_name.ko 参数...
```

成功加载后，可以用lsmod 看到该内核模块。卸载内核模块用系统命令 rmmod：

```
$ rmmod mod_name
```

由于内核模块运行于系统最高特权级，因此可以做你想做的任何事情。

9.2.2 设备文件及设备驱动

设备文件允许进程同内核中的设备驱动通信，并且通过它们和物理设备通信。 Linux 系统中有两类设备文件，一类是字符设备，一类是块设备。设备文件由命令 “mknod”创建并被赋予一个主设备号和一个次设备号。

```
$ mknod device_name c MAJOR MINOR -m 666
```
[1]

为实现对设备的操作，内核模块需要调用函数 register _chrdev 在注册表中对设备进行注册。注册函数传递一个 file_operations 结构，包含了对设备文件的打开、释放、读、写及I/O控制等各种操作的函数指针。这项工作通常在模块加载时完成。模块卸载时还需要调用 unregister_chrdev ，以释放系统资源。

9.3 可编程定时器/计数器的应用

9.3.1 可编程定时器/计数器

早期的个人计算机中，有一片可编程的定时器/计数器 8253，作为系统的硬件时钟设备。 8253在系统中占用 40H~43H端口。三个定时器/计数器的时钟输入均为 1.19MHz，各自承担以下功能:

- T/C0，系统的日时钟，初始化为工作方式三，计数初值0。输出接往可编程中断控制器 8259A 的 IR0，作为系统的计时中断信号。
- T/C1，动态存储器刷新时钟，初始化为工作方式二，计数初值为12H。
- T/C2，控制系统的扬声器，产生声音信号。它的控制端 GATE2和扬声器前均接有控制信号，这些控制信号来自可编程 I/O接口芯片 8255的 PB0和 PB1。 8255初始化已将B口设为方式0输出。

主中断控制器 8259A 占用 20H 和 21H 端口，其IR2供级联次中断控制器输入。如果存在，次中断控制器端口为 0A0H 和 0A1H。

个人计算机为可编程并行接口 8255A 分配端口 60H~63H，初始化后A口作为键盘输入端口，B口用于一些控制信号输出。

目前的计算机中，尽管以上各个独立的功能芯片都已经不存在，但系统集成化后的功能依旧保留，结构原理及编程控制方法也几乎完全一样。

T/C0和T/C1在系统中的作用比较重要，轻易不要改动它们的设置。

[1]参数’c’表示字符设备;’-m 666’设定读写允许。由于是以root 权限建立的文件，因此普通用户无法事后修改读写属性。如果有错，只能删除重建。

9.3.2 编写8253的内核模块和应用程序

在 8253提供的六种工作方式中，只有方式二和方式三是不需要硬件触发能产生连续波形的方式，其中方式三的输出近似方波。我们可以利用T/C2对扬声器的控制功能。如果T/C2产生适当频率的方波，并能将相关的开关打开，使其作用在扬声器上，便可以听到该频率的声音。(思考：为什么不用方式二？)

编写内核模块，直接操作扬声器相关端口，或字符设备驱动程序，供应用程序调用，用扬声器演奏一段乐曲。

由于内核模块运行于0级特权上，因此可以直接使用汇编指令对I/O端口操作。(或使用内核提供的c语言的低级函数inb_p()、outb_p()等等)

下表是可供参考的音符与频率对应关系(单位:Hz):

音符	1	2	3	4	5	6	7
低音	131	147	165	175	196	220	247
中音	262	294	330	349	392	440	494
高音	523	587	659	698	784	880	987

9.4 通用异步串行接口

9.4.1 UART在个人计算机中的发展

IBM为其第一款个人计算机(PC)配备了 8250 的串行卡。8250 没有发送/接收缓冲器，因此其速度相对较慢。在推出 IBM-PC/AT 时选用了更高性能的芯片 16450 。它带有一个字节的发送/接收缓冲器，并修正了旧芯片的缺陷。随着 National Semiconductor 公司推出的引脚兼容的 UART 芯片 16550A 和 16650, 使得个人计算机配置的串行通信性能不断提高。目前的个人计算机系统不再使用单独的 UART 芯片，同其它可编程器件一样，已被更高性能的芯片集成。从软件的角度来看，操作系统和应用程序的操作好像单独的 UART 仍然装在串行适配卡上一样。

在标准配置中，个人计算机串行端口的基地址为 3F8H、 2F8H、 3E8。 对串口的编程兼容 16550 UART 规范。一般的初始化顺序是：

1. 将DLAB置1，写除数寄存器设置波特率。个人计算机提供给 UART 的时钟为 1.8432MHz ，波特率为1843200/(16×除数值)；
2. 置DLAB为0，设置其工作方式(数据位、校验位等等)。

下面就可以对数据端口进行输入/输出操作了。根据工作方式的要求，还需要对数据状态进行查询、错误监测以及调制解调器的控制等等。

Linux 操作系统对串口的操作通过设备文件 /dev/ttySx 完成。

有条件的情况下，用示波器观察异步串行数据的数据帧结构。

9.5 实验内容

- 完成一个内核模块的编写，实现内核模块的正确加载和卸载；

- 建立一个虚拟的字符设备驱动程序，至少要包含读、写功能，为用户程序提供内核空间与用户空间的数据交换，方案及实现过程自定；
- 在上面的基础上完成8253 或UART 的驱动，与自编的应用程序结合，实现特定的功能(如， 8253 的音乐播放， UART 的双机通信)。

9.6 实验报告要求

- 思考：设备文件的文件名、设备名和设备号各起什么作用？
- 查阅参考资料，画出个人计算机系统中与定时器/计数器有关的电路连接，或9针/25针串行端口连接器各引脚的信号；
- 考虑采用8253进行硬件定时的方案(暂不用中断实现)；
- 思考：如果用计算机的键盘模拟琴键，实现类似电子琴的功能，还需要解决哪些问题？
- 哪些因素影响串行数据传输的可靠性？

第十章　嵌入式系统中的I/O接口驱动

10.1　实验目的

- 学习嵌入式Linux操作系统设备驱动的方法；

10.2　接口电路介绍

Linux以模块的形式加载设备类型, 通常一个模块对应一个设备驱动, 因此是可以分类的。将模块分成不同的类型并不是一成不变的, 开发人员可以根据实际工作需要在一个模块中实现不同的驱动程序。一般情况, 一个设备驱动对应一类设备的模块方式, 这样便于多个设备的协调工作, 也利于应用程序的开发和扩展。

设备驱动程序负责将应用程序如读、写等操作正确无误的传递给相关的硬件, 并使硬件能够做出正确反应。因此在编写设备驱动程序时, 必须要了解相应的硬件设备的寄存器、IO口及内存的配置参数。

设备驱动在准备好以后可以编译到内核中, 在系统启动时和内核一起启动, 这种方法在嵌入式 Linux 系统中经常被采用。在开发阶段, 设备驱动的动态加载更为普遍。开发人员不必在调试过程中频繁启动机器就能完成设备驱动的调试工作。

嵌入式处理器片内集成了大量的可编程设备接口, 为构成处理器系统带来了极大的便利。S5PV210 有237 只多功能引脚, 大多数是功能复用的, 可以通过初始化编程将它们设置为GPIO(General Purpose I/O, 通用IO)或者某个具体功能。本章实验通过学习GPIO 对一些设备的控制, 掌握Linux设备驱动的基本方法。

10.2.1　LED

本系统有5 个LED, 其中一个是电源指示, 三个通过GPJ0 的三只引脚控制, 一个通过PWMTOUT1控制。结构图如右。

通过GPIO控制LED, 还需要了解系统的硬件资源分配。表10.1 列出了GPJ0的相关寄存器, 寄存器的功能含义在表10.2中。

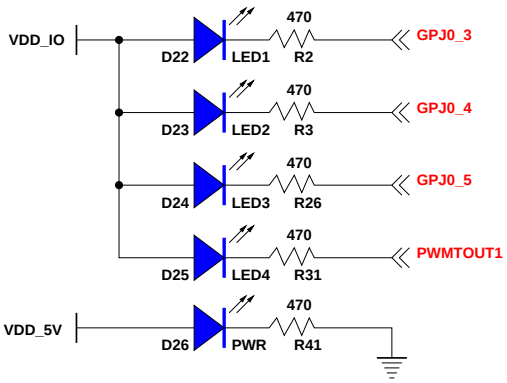


图 10.1： LED控制信号

表 10.1: GPJ0 寄存器地址

寄存器名	地址	描述
GPJ0CON	0xE020_0240	GPJ0 配置寄存器
GPJ0DAT	0xE020_0244	GPJ0 数据寄存器
GPJ0PUD	0xE020_0248	GPJ0 上拉/下拉寄存器
GPJ0DRV	0xE020_024C	GPJ0 驱动强度控制寄存器
GPJ0CONPDN	0xE020_0250	GPJ0 掉电模式配置寄存器
GPJ0PUDPDN	0xE020_0254	GPJ0 掉电模式上拉/下拉寄存器

表 10.2: 寄存器描述

寄存器	位域	描述
GPJ0CON[7]	[31:28]	0000 = Input 0001 = Output 0010 = MSM_ADDR[7] 0011 = CAM_B_DATA[7] 0100 = CF_DMACKNs 0101 = MHL_D0 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[7]
GPJ0CON[6]	[27:24]	0000 = Input 0001 = Output 0010 = MSM_ADDR[6] 0011 = CAM_B_DATA[6] 0100 = CF_DRESETN 0101 = TS_ERROR 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[6]
GPJ0CON[5]	[23:20]	0000 = Input 0001 = Output 0010 = MSM_ADDR[5] 0011 = CAM_B_DATA[5] 0100 = CF_DMARQ 0101 = TS_DATA 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[5]
GPJ0CON[4]	[19:16]	0000 = Input

表 10.2: 寄存器描述(续)

寄存器	位域	描述
		0001 = Output 0010 = MSM_ADDR[4] 0011 = CAM_B_DATA[4] 0100 = CF_INTRQ 0101 = TS_VAL 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[4]
GPJ0CON[3]	[15:12]	0000 = Input 0001 = Output 0010 = MSM_ADDR[3] 0011 = CAM_B_DATA[3] 0100 = CF_IORDY 0101 = TS_SYNC 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[3]
GPJ0CON[2]	[11:8]	0000 = Input 0001 = Output 0010 = MSM_ADDR[2] 0011 = CAM_B_DATA[2] 0100 = CF_ADDR[2] 0101 = TS_CLK 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[2]
GPJ0CON[1]	[7:4]	0000 = Input 0001 = Output 0010 = MSM_ADDR[1] 0011 = CAM_B_DATA[1] 0100 = CF_ADDR[1] 0101 = MIPI_ESC_CLK 0110 ∼ 1110 = Reserved 1111 = GPJ0_INT[1]
GPJ0CON[0]	[3:0]	0000 = Input 0001 = Output 0010 = MSM_ADDR[0] 0011 = CAM_B_DATA[0] 0100 = CF_ADDR[0]

表 10.2: 寄存器描述(续)

寄存器	位域	描述
		0101 = MIPI_BYTE_CLK 0110 ~ 1110 = Reserved 1111 = GPJ0_INT[0
GPJ0DAT	[7:0]	端口配置为输入时, 从寄存器读入值对应位反映引脚电平状态; 端口配置为输出时, 写出位产生对应引脚电平; 端口配置为功能引脚时, 寄存器值不确定
GPJ0DRV[n]	[2n+1:2n] n=0~7	00 = 1× 01 = 2× 10 = 3× 11 = 4×
GPJ0CONPDN[n]	[2n+1:2n] n=0~7	00 = 输出0 01 = 输出1 10 = 输入 11 = [保留]
GPJ0PUDPDN[n]	[2n+1:2n] n=0~7	00 = 禁止上拉/下拉 01 = 下拉 10 = 上拉 11 = [保留]

10.2.2 I/O端口地址映射

RISC处理器(如ARM、PowerPC等)通常只实现一个物理地址空间, 外设I/O端口成为内存的一部分。此时, CPU可以像访问一个内存单元那样访问外设I/O端口, 而不需要设立专门的外设I/O指令。这两者在硬件实现上的差异对于软件来说是完全透明的, 驱动程序开发人员可以将存储器映射方式的I/O端口和外设内存统一看作是“I/O内存” 资源。

I/O设备的物理地址是已知的, 由硬件设计决定。但是 CPU 通常并没有为这些已知的外设I/O内存资源的物理地址预定义虚拟地址范围, 驱动程序不能直接通过物理地址访问I/O设备, 而必须通过页表将它们映射到内核虚地址空间, 然后才能根据映射所得到的内核虚地址范围, 通过访内指令访问这些I/O设备。Linux 的内核函数 ioremap() 用来将I/O设备的物理地址映射到内核虚地址空间。其原型如下：

void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags)

端口释放时, 应通过函数iounmap()取消ioremap()所做的映射：

void iounmap(void * addr)

当I/O设备的物理地址被映射到内核虚拟地址后, 就可以像读写RAM 那样直接读写I/O设备资源了。例如可以通过下面的方式点亮最左边的一个LED：

```
#define GPJ0CON          0xE0200240
#define GPJ0DAT          0xE0200244
......

  volatile int *pConfReg = ioremap(GPJ0CON, 4);/* 映射控制寄存器地址*/
  volatile int *pDataReg = ioremap(GPJ0DAT, 4);/* 映射数据寄存器地址*/

  *pConfReg &= 0xFFFF0FFF;                    /* 只改变GPJ0_3设置*/
  *pConfReg |= 0x00001000;                    /* GPJ0_3 设为输出*/

  *pDataReg &= ~(1 << 3);                     /* GPJ0_3 输出低电平*/
......
```

为了保证驱动程序的跨平台的可移植性, 建议使用 Linux 中特定的函数来访问 I/O 内存资源, 如readb()、 readw()、 writeb()、 writew() 等。在RISC处理器里, 它们实际上就是对存储器读写的重定义：

```
#define readb(addr)  (*(volatile unsigned char *)__io_virt(addr))
#define readw(addr) (*(volatile unsigned short *)__io_virt(addr))
#define readl(addr) (*(volatile unsigned int *)__io_virt(addr))

#define writeb(b,addr) (*(volatile unsigned char *)__io_virt(addr) = (b))
#define writew(b,addr) (*(volatile unsigned short *)__io_virt(addr) = (b))
#define writel(b,addr) (*(volatile unsigned int *)__io_virt(addr) = (b))

......
```

10.2.3 Key pad

S5PV210 内部设计了矩阵键盘接口, 但在我们的实验系统中, 只引出了少数的按键开关, 这些按键都通过GPIO按位进行简单操作。在嵌入式系统中, 复杂的人机交互可以通过更先进的设备(例如触摸屏等)完成。按键开关逻辑结构如图10.2。其中KP_COL0在GPIO的GPJ1 组, KP_COL1、KP_COL2、KP_COL3在GPJ2 组, EINT2 和EINT3 在GPH0 组。EINT2和EINT3可以实现外部中断。

请根据数据手册, 设计实现按键的设备驱动。

10.2.4 PWM 定时器

S5PV210内部有五个32位PWM定时器, 其中Timer0、1、2、3 四个定时器可以输出驱动PWM引脚。定时器时钟源来自APB-CLOCK, Timer0、1 共用一个8 位预定标器, Timer2、3、4 共用另一个8 位定标器, 作为第一级分频；每个定时器还有各自的除法器(分频数1、2 、4 、8 、16)作为第二级分频。

每个定时器有一个32位的减法计数器, 由定时器计数寄存器TCNTBn载入初值。该值减到零时向CPU产生中断, 并自动从TCNTBn重载, 进入下一轮循环, 除非定时器控制寄存器TCONn 中的定时器

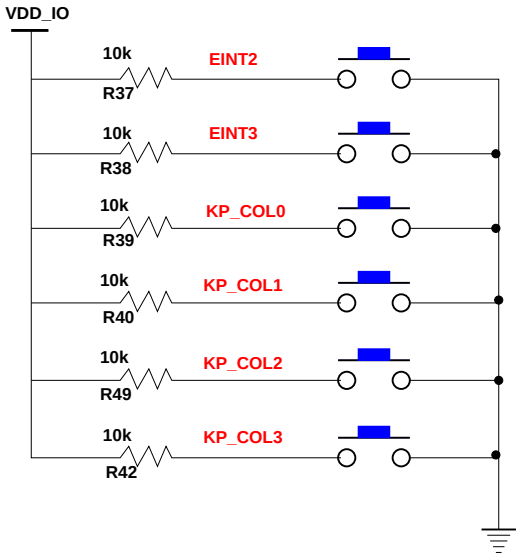


图 10.2: 按键控制信号

允许位被清除。清除TCONn 的允许位导致定时器停止工作, 计数器不再重载。

PWM方式使用定时器比较寄存器TCMPBn。当递减计数器的值与TCMPBn 相匹配时, 定时器输出电平发生改变, 因此, 比较寄存器决定了占空比。

TCNTBn 和TCMPBn 是双缓冲的, 允许在计数过程中更新, 更新值在下一个计数周期生效。

表10.3列出了与定时器相关的控制寄存器地址。定时器输入时钟

$$CLOCK_{PWM} = \frac{PCLK}{(prescaler + 1) * divider}$$

第一级分频系数prescaler 来自TCFG0:

TCFG0[15:8] = prescaler1, 用于Timer2、3、4;

TCFG0[7:0] = prescaler0, 用于Timer0、1;

第二级分频数“divider value”由TCFG1 中的MUXn 提供:

MUX	分频/开关
0000	1/1
0001	1/2
0010	1/4
0011	1/8
0100	1/16
0101	SCLK_PWM

[31:24]	[23:20]	[19:16]	[15:12]	[11:8]	[7:4]	[3:0]
Reserved	N/A	MUX4	MUX3	MUX2	MUX1	MUX0

表 10.3: 定时器相关寄存器地址

寄存器名	地址	描述
TCFG0	0xE250_0000	定时器配置寄存器1(2个8-bit 预定标器)

表 10.3: 定时器相关寄存器地址(续)

寄存器名	地址	描述
TCFG1	0xE250_0004	定时器配置寄存器2(5个二级分频选择开关)
TCON	0xE250_0008	定时器控制寄存器
TCNTB0	0xE250_000C	Timer 0 计数缓冲寄存器
TCMPB0	0xE250_0010	Timer 0 比较缓冲寄存器
TCNTO0	0xE250_0014	Timer 0 计数器观察寄存器
TCNTB1	0xE250_0018	Timer 1 计数缓冲寄存器
TCMPB1	0xE250_001C	Timer 1 比较缓冲寄存器
TCNTO1	0xE250_0020	Timer 1 计数器观察寄存器
TCNTB2	0xE250_0024	Timer 2 计数缓冲寄存器
TCMPB2	0xE250_0028	Timer 2 比较缓冲寄存器
TCNTO2	0xE250_002C	Timer 2 计数器观察寄存器
TCNTB3	0xE250_0030	Timer 3 计数缓冲寄存器
TCMPB3	0xE250_0034	Timer 3 比较缓冲寄存器
TCNTO3	0xE250_0038	Timer 3 计数器观察寄存器
TCNTB4	0xE250_003C	Timer 4 计数缓冲寄存器
TCNTO4	0xE250_0040	Timer 4 计数器观察寄存器

控制寄存器TCON 用于5个定时器的工作方式设置。

表 10.4: 定时器控制寄存器

TCON	位域	描述
Reserved	[31:23]	—
Timer 4 自动重载	[22]	0 = 单次 1 = 自动重载
Timer 4 手动更新	[21]	0 = 无操作 1 = 更新TCNTB4
Timer 4 启动/停止	[20]	0 = 停止 1 = 启动Timer4
Timer 3 自动重载	[19]	0 = 单次 1 = 自动重载
Timer 3 输出反转开关	[18]	0 = 不反转 1 = TOUT_3 反转
Timer 3 手动更新	[17]	0 = 无动作 1 = 更新TCNTB3

表 10.4: 定时器控制寄存器(续)

TCON	位域	描述
Timer 3 启动/停止	[16]	0 = 停止 1 = 启动Timer 3
Timer 2 自动重载	[15]	0 = 单次 1 = 自动重载
Timer 2 输出反转开关	[14]	0 = 不反转 1 = TOUT_2 反转
Timer 2 手动更新	[13]	0 = 无动作 1 = 更新TCNTB2,TCMPB2
Timer 2 启动/停止	[12]	0 = 停止 1 = 启动Timer 2
Timer 1 自动重载	[11]	0 = 单次 1 = 自动重载
Timer 1 输出反转开关	[10]	0 = 不反转 1 = TOUT_1 反转
Timer 1 手动更新	[9]	0 = 无动作 1 = 更新TCNTB1,TCMPB1
Timer 1 启动/停止	[8]	0 = 停止 1 = 启动Timer 1
Reserved	[7:5]	
死区允许/禁止	[4]	
Timer 0 自动重载	[3]	0 = 单次 1 = 自动重载
Timer 0 输出反转开关	[2]	0 = 不反转 1 = TOUT_0 反转
Timer 0 手动更新	[1]	0 = 无动作 1 = 更新TCNTB0,TCMPB0
Timer 0 启动/停止	[0]	0 = 停止 1 = 启动Timer 0

使用定时器输出功能时, 还需要正确设置相应的GPIO引脚。

系统底板上用PWM 控制的设备有两个：蜂鸣器和LED4。请根据数据手册完成对这两个设备的驱动。

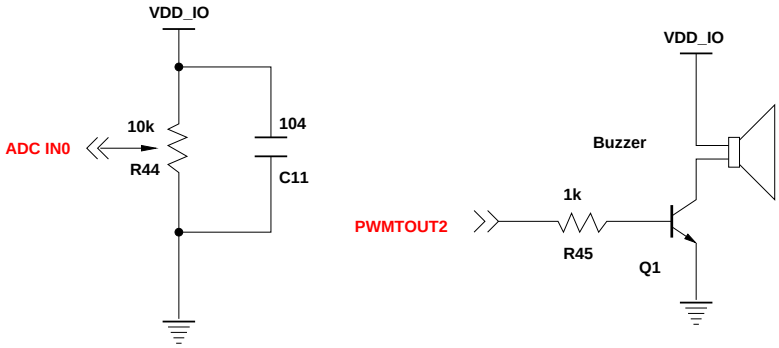


图 10.3: ADC和蜂鸣器

10.3 实验内容

根据硬件接口资料, 实现任意一个设备的基本控制功能, 包括驱动程序和用户程序。

第十一章　mplayer移植

11.1　实验目的

- 掌握Linux 系统中应用软件移植的过程和方法
- 理解软件层次依赖关系

11.2　软件介绍

MPlayer是一款开源的多媒体播放器，支持几乎所有的音频和视频播放, 以GNU 通用公共许可证发布, 可在各主流操作系统使用, 是Linux 系统中最重要的播放器之一。MPlayer 中还包含音视频编码工具mencoder。MPlayer 本身是基于命令行界面的程序, 不同操作系统、不同发行版可以为它配置不同的图形界面, 使其外观多姿多彩。MPlayer 本身也可以编译成GUI 方式。

MPlayer 除了可以播放一般的磁盘媒体文件外, 还支持CD、VCD、DVD等多种物理介质和多种网络媒体(rtp://、rtsp://、http://、mms:// 等)。视频播放时, 它还支持多种不同格式的外挂字幕。大部分音视频格式都能通过ffmpeg 项目(另一个开源项目, 提供音视频编解码库支持) 的libavcodec 函数库原生支持。对于那些没有开源解码器的格式，MPlayer使用二进制的函数库。它能直接调用Windows的DLL。

11.3　编译准备

下载MPlayer 源代码MPlayer-1.0rc2.tar.bz2, libmad-0.15.1b.tar.gz和zlib-1.2.3.tar.bz2, 分别将其解压。libmad 是高品质全定点算法的MPEG 音频解码库。

准备一个有操作权限的工作目录, 例如~/workspace, 作为下面编译结果的暂存目录。以后的编译过程中，将编译选项--prefix 设置为该目录。所有编译完成后再将其内容移至开发板适当位置。缺省的安装路径是/usr/local, 该路径其一需要root 权限，其二,它是主机系统的一个重要目录, 如果被目标机架构arm的代码覆盖, 会影响主机的正常工作。

将交叉编译器路径添加到环境变量“PATH” 中。

11.4　编译

1. 进入libmad-0.15.1b 目录,配置编译环境:

   ./configure --enable-fpm=arm --host=arm-none-linux-gnueabi --disable-debugging --prefix=/home/student/workspace

选项“--host”是编译器前缀。

2. 编译及安装: make install

   在编译时会提示错误: cc1: error: unrecognized command line option “-fforce-mem”。这是因为gcc3.4或更高版本已经将fforce-mem 选项去除了。只需要在Makefile 中找到该字符串,将其删除即可。

   编译完成后会将静态库libmad.a 和动态库madlib.so 安装到/home/student/workspace/lib 目录下。如果是动态链接，需要将动态链接库复制到目标系统的/lib 目录下。如果不想用动态链接，可以在上面的编译选项中添加一条“--disable-shared”。此原则同样适用于下面的zlib 库。

3. 进入zlib-1.2.3 解压目录, 按下面的步骤编译安装:

   export CC=arm-none-linux-gnueabi-gcc

   ./configure --prefix=/home/student/workspace

   make install

4. 进入MPlayer 解压目录, 进行如下配置:

   ./configure --cc=arm-none-linux-gnueabi-gcc --target=arm-linux --enable-static --prefix=/home/student/workspace --disable-mp3lib --disable-dvdread --disable-mencoder --disable-live --enable-mad --disable-armv5te --disable-armv6 --enable-libavcodec_a --enable-ossaudio --extra-cflags='-I /home/student/workspace/include' --extra-ldflags='-L /home/student/workspace/lib'

   上面最后两个选项用到了之前准备的libmad 和libz 的头文件及生成的库文件路径。配置正确后，可以用make 命令编译。如果一切正常，便可在当前目录下生成可执行文件mplayer。注意最后链接时用到的库。如果是动态链接，这些库需要复制到目标系统的/lib 目录。

最后，尝试在目标机上播放一些音视频文件。

11.5 扩展功能

1. 尝试编译具有图形用户界面的MPlayer 播放器
2. 用--enable-mencoder 选项编译mencoder，并利用它进行音视频编码、转码。