嵌入式系统

实 验 指 导 书

(基于 BeagleBone Black)

南京大学 电子科学与工程学院

2019 年 10 月

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。

目录

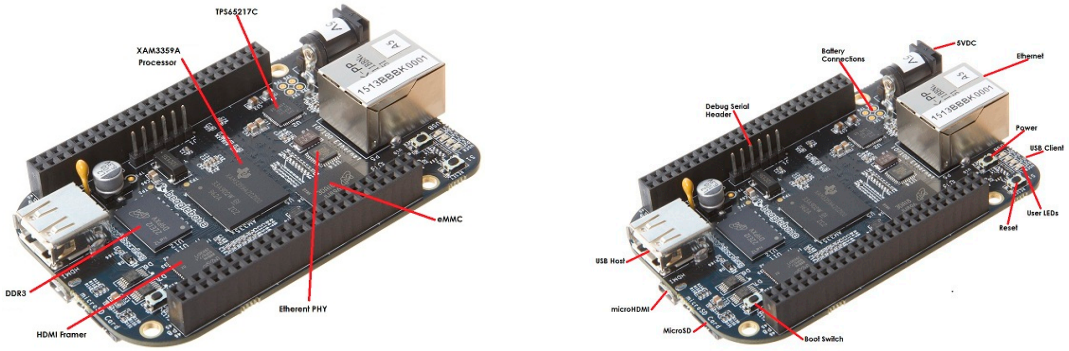
实验 1

实验系统介绍

1.1 性能概括

BeagleBone Black 基于 TI 的嵌入式处理器 Sitara AM3358(ARM Cortex–A8 架构), 主频 1GHz, 3D 图形引擎 SGX530。板载 512M DDR3L 内存和 4GiB eMMC 闪存。采用 6 层板工艺设计。主要性能和接口列表如下:

- 内核:AM3358(ARM Cortex–A8), 主频 1GHz
- 内存:512MB DDR3L
- Flash:4GB eMMC
- 电源管理: TPS65217PMIC
- USB Client(USB0, mini–USB), USB HOST(USB1)
- UART0 (3.3V TTL)
- microSD 卡接口 (3.3V)
- HDMI 高清视频接口 (Max. 1280x1024)
- 音频输出:HDMI



(a) 主要部件　　(b) 接口

图 1.1: BeagleBone Black 外观

- 有线以太网:10/100M,RJ45, LAN8720
- 3D 高性能图形加速
- 扩展接口:McASP, SPI, I2C, LCD, MMC,GPIO, ADC 等

1.2 软件

在 Beagle Bone Black 平台上, 除了可以进行 Linux 系统的移植和开发以外, 还可以移植 RTEMS、VxWorks 等多种实时操作系统以及进行无操作系统的裸机实验。本课程主要围绕 Linux 操作系统进行开发。

1.2.1 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, 再通过交叉编译工具链生成最终目标文件, 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 binutils, 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。

1.2.2 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 /opt/armhf-linux-2018.08, 可执行程序在 /opt/armhf-linux-2018.08/bin 目录下. 实验中请将该目录添加到环境变量 “PATH” 中, 或在编译时给出完整的路径和编译程序名 /opt/armhf-linux-2018.08/bin/arm-none-linux-gnueabi-gcc. 可以通过下面的方法检查编译器是否正确安装:

首先编写一个 C 语言原文件 (设文件名是 hello.c), 在该目录下执行

```
$ arm-none-linux-gnueabi-gcc -o hello hello.c
```

看看是否生成了名为 hello 的可执行程序. 或者使用下面的 Makefile

```
CC      = /opt/armhf-linux-2018.08/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
```

1.2.3 嵌入式系统软件

嵌入式操作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(git://git.denx.de/u--boot.git。针对本系统，可以直接使用 am33xx_evm 的缺省配置进行编译。

在 https://github.com/beagleboard/linux 可以下载针对 BeagleBone-Black 移植的 Linux 内核。用户可根据自己的需求在此基础上进一步裁剪和优化。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。

实验 2

嵌入式系统开发环境

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。

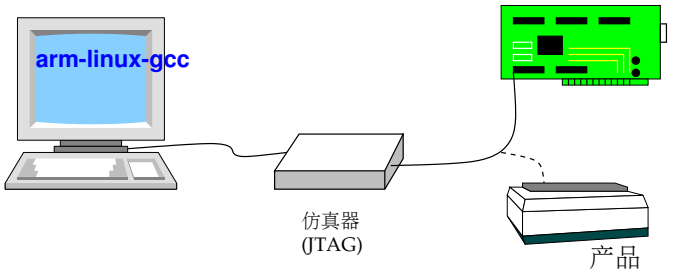


图 2.1: 宿主机/目标机的开发模式

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 bootloader 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 bootloader 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。

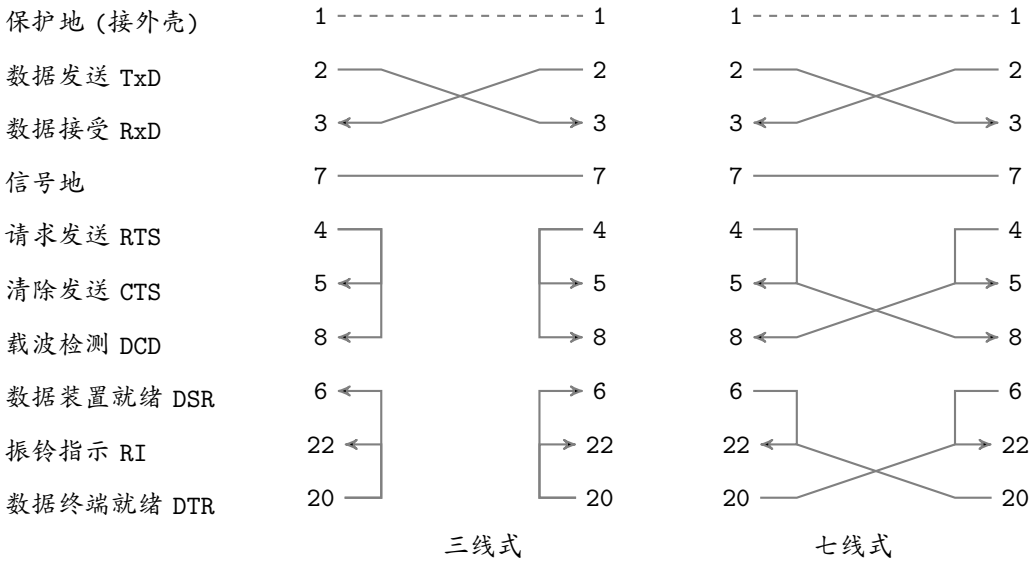


图 2.2: PC 机串口引脚信号

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

- 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。本实验通过 USB 转 RS232 芯片连接串口, 设备是 /dev/ttyUSB0。
- 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 "U–Boot #"。"`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:

[1] ^A 是 minicom 的控制键, 可以激活其他功能设置, 例如 ^A–z 调出全部功能菜单, ^A–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

```
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

```
setenv bootargs console=ttyO0,115200 root=/dev/mtdblock2 rw
        rootfstype=yaffs2 init=/linuxrc
```

表示用串口设备 ttyO0 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 "fatload mmc 0 0x82000000 zImage;bootz 0x82000000"
```

表示将 eMMC 卡的第一分区 (FAT 文件系统) 内核映像文件 zImage 读到内存 0x82000000 起始的地址中, 然后从 0x82000000 处开始运行。

- `tftp`: tftp 命令, 如

```
tftp 0x82000000 zImage
```

将 tftp 服务器目录中的文件 zImage 通过 tftp 协议读入内存 0x82000000 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
    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, 在主机启用 NFS 会打开这两个服务:

[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.

```
# service nfs-kernel-server start
```

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 192.168.1.101
```

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 地址为 192.168.1.100):

```
# mount 192.168.1.100:/opt /mnt -o noloc,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      = /opt/armhf-linux-2018.08/bin/arm-linux-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

实验 3

BootLoader

3.1 实验目的

- 了解操作系统的启动过程
- 学习制作可引导的存储卡

3.2 实验步骤

3.2.1 配置 bootloader 选项

在下载的 u–boot 目录中, 执行 `make am335x_evm_defconfig`, 即完成针对 BB–Black 的缺省 bootloader 配置。也可以用 `make menuconfig` 进入菜单界面, 对 bootloader 的单个选项进行合理的取舍。

完成配置后, 用 “make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-” 进行编译。“arm-none-linux-gnueabi-” 是 ARM 编译器的前缀。编译完成后, 生成 MLO 和 u–boot.img。

3.2.2 制作 TF 卡

将前面生成的 MLO 和 u–boot.img 用下面的命令写入 TF 卡, BeagleBone Black 上电时按住 S2, 系统首先选择 MMC0(microSD 卡插槽) 启动。

```
# dd if=MLO of=/dev/mmcblk0 count=1 seek=1 bs=128k
# dd if=u-boot.img of=/dev/mmcblk0 count=2 seek=1 bs=384k
```

再在该卡上创建两个文件系统, 一个用于存放启动内核相关的固件及内核映像, 用 VFAT 格式化; 一个用于准备作为根文件系统, 用 inode 型文件系统格式化 (Ext4FS、BtrFS、ReiserFS 等等均可, 要求内核支持)。前者不宜过大, 够用即可。

```
# mkfs.vfat /dev/mmcblk0p1
# mkfs.ext4 /dev/mmcblk0p2
```

设备名 /dev/mmcblkX 因时而异。如果通过 USB 转接, 设备名也可能是 /dev/sdbX。请认清操作对象, 错误的操作可能会破坏重要数据。此外注意在分区时不要覆盖之前写入 TF 卡的裸数据所在扇区。

如果要求系统能自动引导操作系统, 还需要在引导分区上建立 u–boot 的脚本文件。u–boot 默认的脚本文件名是 uEnv.txt。它至少应包含下面的内容:

- 将指定的内核映像文件加载到内存的指定位置
- 设定向内核传递的启动参数, 特别重要的是根文件系统的类型和位置
- 跳转到内核解压的内存地址开始运行, 向内核交权

一个典型的 uEnv.txt 内容如下:

```
loadkernel=fatload mmc 0 0x82000000 zImage
loadfdt=fatload mmc 0 0x88000000 /dtbs/am335x-boneblack.dtb

rootfs=root=/dev/mmcblk0p2 rw rootfstype=ext4
loadfiles=run loadkernel; run loadfdt
mmcargs=setenv bootargs console=ttyO0,115200n8 ${rootfs}

uenvcmd=run loadfiles; run mmcargs; bootz 0x82000000 - 0x88000000
```

实验 4

Linux 内核配置和编译

4.1 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

4.2 相关知识

4.2.1 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, 包括 Samsung 的 S3C 系列 mach–s3cXXXX、TI 的 davinci 系列 mach–davinci 以及 OMAP 系列 mach–omapX 等等。
- `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), 其中所包含的 `main.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。
- `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`：包含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。内核源码每个目录下一般都有一个 Kconfig 文件和一个 Makefile 文件, 前者用于生成配置界面, 后者用于构造依赖关系, 他们他们通过脚本文件发生作用。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。

4.2.2 配置内核的基本结构

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` 定义了 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 `Kconfig`：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 GTK+ 库的图形界面 `make xconfig` 和基于 Qt 库的图形界面 `make gconfig`);
4. `scripts` 目录下的脚本文件。

4.2.3 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, 每个配置变量具有一下三种不同的取值：

- 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, 配置文件中该变量被注释掉。

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 Kconfig 文件中用配置语言来编写配置脚本, 顶层Makefile 调用 scripts/config, 按照 arch/arm/Kconfig 来进行配置 (对于 ARM 架构来说)。命令执行完后生成保存有配置信息的配置文件 .config。

4.3 编译内核

4.3.1 Makefile 的选项参数

编译 Linux 内核常用的 make 命令参数包括 config、clean、mrproper、zImage、bzImage、modules、modules_install 等等。

1. config、menuconfig、xconfig 等: 内核配置。按不同的界面调用./scripts/config 进行配置。命令执行后产生文件 .config, 其中保存着配置信息。下次配置后将产生新的 .config 文件, 原 .config 更名为 .config.old, 供 make oldconfig 使用。
2. 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。
3. mrproper: 删除以前在构核过程产生的所有文件, 即除了做 clean 外, 还要删除 .config 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。
4. 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。如果核心代码没有错误, 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 的基础上产生的压缩核心映像文件。正常编译完成后, 生成的 ARM 内核映像文件在目录 arch/arm/boot 中, 需将其移至 tftp 服务器目录供下载。
5. 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。make modules_install 将这些文件复制到内核模块文件目录。在 PC 机上通常是 /lib/modules/$VERSION/.....。对于嵌入式开发, 应通过变量 INSTALL_MOD_PATH 指定复制的目标目录, 移植时将他们复制到目标系统的/lib/modules 目录。

4.3.2 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 系统中可以通过内核模块完成这些操作。

4.4 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:

```
u-boot # tftp 0x82000000 zImage
u-boot # bootz 0x82000000
```

4.5 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?

实验 5

嵌入式文件系统的构建

5.1 实验目的

- 了解嵌入式操作系统中文件系统的类型和作用;
- 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。

5.2 Linux 文件系统的类型

5.2.1 Ext 文件系统

Ext2FS 是 Linux 2.4 版本的标准文件系统, 取代了早期的扩展文件系统 (ExtFS)。ExtFS 支持的文件最大为 2GB, 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。

为增强文件系统的健壮性, 又开发了日志型文件系统 Ext3FS 和 Ext4FS, 在基于 ROM 的存储设备上有效地提高了文件系统的可靠性。目前主流 Linux 发行版普遍都使用 Ext4FS 文件系统, 其中包括服务器和工作站, 甚至一些嵌入式设备。

5.2.2 NFS 文件系统

NFS 是一个 RPC (远程系统调用) 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 2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:

```
u-boot # setenv rootfs root=/dev/nfs rw nfsroot=<server_ip>:<Root_Dir>
u-boot # setenv nfsaddrs nfsaddrs=<ip>:<server_ip>:<gateway>:<mask>
u-boot # setenv bootargs console=ttyS0,115200 $rootfs $nfsaddrs
```

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

5.2.3 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 David 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”。

5.2.4 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。

5.2.5 RAM Disk

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 RAM Disk。将 RAM Disk 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。通常在制作 RAM Disk 时还会对文件系统映像进行压缩, 以节省存储空间。但它也有缺点: 由于将内存的一部分用作 RAM Disk, 这部分内存不能再作其他用途; 此外系统运行时更新的内容无法保存, 系统关机后内容将丢失。

5.3 文件系统的制作

5.3.1 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 `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]

5.3.2 BusyBox 的编译

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

[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.

- 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适当的取舍, 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`。

5.3.3 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 BeagleBone
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
          BB -- BLACK
  ==============================
```

[2]如文件系统带有共享库, 也可以采用动态链接方式

此文件内容不影响系统正常工作, 由 /etc/rc 调用打印在终端上。文件名来自 Message Of ToDay 的缩写。

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 [3]:

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

- 建立 proc 和sys 空目录, 供 PROC 和 SYSFS 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-2.23.so, libc-2.23.so, libm-2.23.so

  并做相应的符号链接:

```
$ ln -s ld-2.23.so ld-linux-armhf.so.3
$ ln -s libc-2.23.so libc.so.6
$ ln -s libm-2.23.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    sys
```

它们是下面制作文件系统的基础。其中lost+found 是在格式化 Ext2FS 文件系统时生成的。

5.3.4 制作 RAM Disk 文件映像

配置内核时, RAM Disk 要求在 “Block devices→” 中选中 “RAM block device support”, 并设置适当的 RAM Disk 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 RAM Disk, 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

[3]如果内核支持了 devtmpfs 自动挂载功能, 创建设备文件的工作会由系统自动完成。此外, 手工创建设备文件的工作 mknod 应在目标文件系统中进行, 因为设备文件不能用 cp 命令复制。

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。[4]

```
$ 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 RAM Disk, 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 RAM Disk 作为根文件系统。同时 RAM Disk 的映像文件也应装入内存的对应位置:

```
u-boot # setenv ramdisk root=/dev/ram rw initrd=0x88080000,8M
u-boot # setenv bootargs console=ttyO0,115200 $ramdisk
u-boot # tftp 0x88080000 ramdisk_img.gz
```

5.3.5 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, 进入根文件系统结构所在目录/mnt/ramdisk, 在这里创建 init。可以直接使用 BusyBox 编译出来的 init:

```
$ ln -s bin/busybox init
```

如果 BusyBox 不选择编译 init, 也可以创建一个以 init 命名的脚本, 然后用cpio 命令将整个目录结构打包并压缩:

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

注意将生成的文件 initrd.cpio.gz 放到另外的目录 (这里放到用户主目录下), 以免被递归。

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 initrd.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。

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。

如需在原有基础上修改, 同样也需要用 cpio 解包:

[4]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。

```
$ gunzip -cd ~/initrd.cpio.gz | cpio -i
```

5.4 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 RAM Disk 文件系统映像像, 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。

5.5 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/Ext4FS、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。

实验 6

图形用户接口

6.1 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

6.2 原理概述

6.2.1 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, 也称显存。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。

6.2.2 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 [1]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

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

计算出以字节为单位的 Frame Buffer 的大小。

6.2.3 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。

6.2.4 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:

[1]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

```
#include <linux/fb.h>
......
struct fb_var_screeninfo vinfo;
......
    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 a red point at (x, y) in RGB888

    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: [2]

```
// draw a red point at (x, y) of color (Red/Green/Blue)

    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。

[2]不同位端 (endian) 的处理器, 移位及高低字节顺序有所不同。

6.3 实验内容

6.3.1 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。

6.3.2 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`"。链接时, 可使用`-l foo` 选项链接静态库文件 libfoo.a 或共享库文件。

6.4 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, 静态库和共享库在软件结构中的地位和特点;
- 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?

实验 7

音频接口程序设计

7.1 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。

7.2 原理概述

内核编译时需要有声音支持。

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。

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`ioctl()` 设置命令有:

- `SNDCTL_DSP_RESET`: 声音设备复位。
- `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, 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 (8 位/16 位)。
- `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`、`SNDCTL_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) 操作的常用命令有:

- 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。

请参考内核 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。

7.2.1 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 */
```

7.3 实验内容

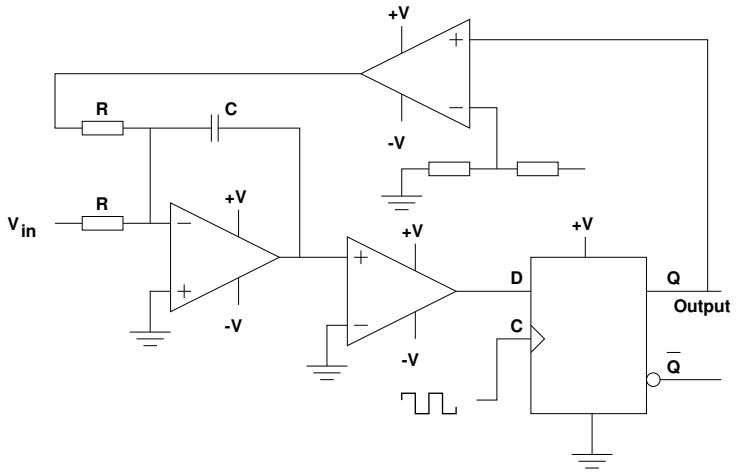
根据内核配置, 选择适当的音频驱动, 实现数字音频的采集与回放。

7.4 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

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:

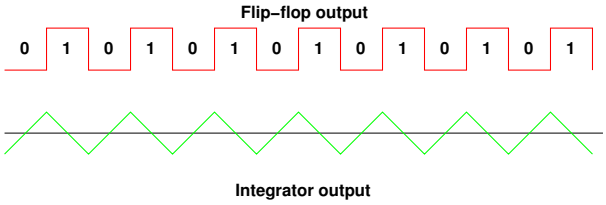


The leftmost op-amp is the (summing) integrator. The next op-amp the integrator feeds into is the comparator, or 1-bit ADC. Next comes the D-type flip-flop, which latches the comparator's output at every clock pulse, sending either a "high" or "low" signal to the next comparator at the top of the circuit. 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:

$\Delta\Sigma$ converter operation with

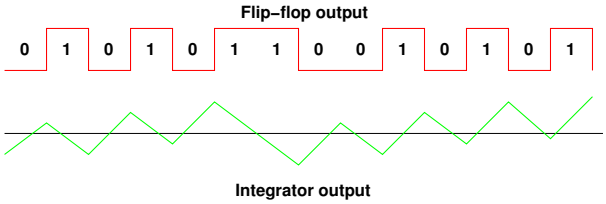
0 volt analog input



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:

$\Delta\Sigma$ converter operation with

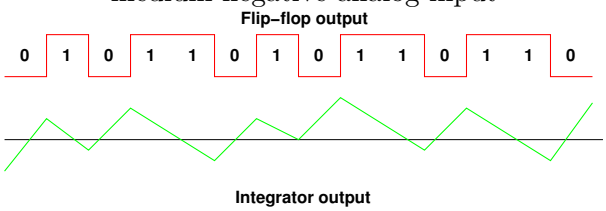
small negative analog input



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:

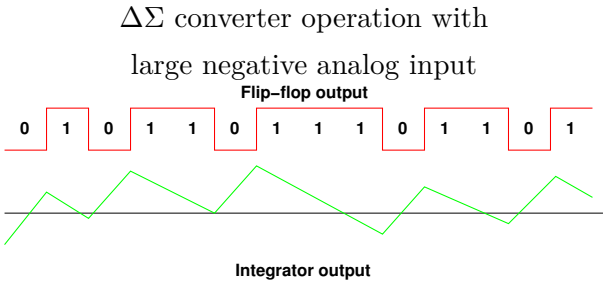
$\Delta\Sigma$ converter operation with

medium negative analog input



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.

实验 8

触摸屏移植

8.1 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。

8.2 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.

8.2.1 触摸屏的基本原理

触摸屏是由触摸板和显示屏两部分有机结合在一起构成的设备. 根据不同的感应方式, 触摸板又有电阻式、电容式、声表面波式等不同构成. 早期电阻式触摸板由两层透明的金属氧化物导电层构成. 当触摸屏被按压时, 平常相互绝缘的两层导电层就在触摸点位置形成接触. 由于当触摸板在 X 和 Y 方向分布了均匀电场, 按压点相当于在 X 和 Y 方向形成了电阻分压. A/D 转换器对该电压采样便可得到按压点的位置坐标.

电容触摸屏利用人体电流感应进行工作. 当被触碰时, 人体电容和触摸板形成耦合, 触摸屏四个角上的电感应设备可以检测到电流的变化. 控制器通过对这四个电流比例的计算得到触摸点的位置.

以上提到的几种触摸板, 无论采用何种传感原理, 最终都要通过 A/D 转换器变成数字量进行分析计算. Linux 系统内核中完成 A/D 转换部分, 而触摸屏库则给应用程序提供方便的接口.

8.2.2 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。

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。

8.2.3 触摸屏库 tslib

下载触摸屏库 tslib-1.0.0.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:

```
$ ./autogen.sh
$ ./configure --host=arm-linux --prefix=/path/to/install
$ make
$ 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。`make` 命令正确完成后, 结果分散在 `src/.libs` (库) 和 `plugins/.libs` (插件) 中, 可通过手工复制到目标系统。
- 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。

8.2.4 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 `/usr` 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=/usr/lib/ts`

  插件模块文件 (.so) 所在目录。

- `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 `/usr/bin/ts_test`。

8.3 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。

实验 9

嵌入式系统中的 I/O 接口驱动

9.1 实验目的

学习嵌入式 Linux 操作系统设备驱动的方法。

9.2 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。am335x 实现 4 组 GPIO 模块、每组 32 只引脚的输入/输出控制功能, 它们可用于信号的输入/输出、键盘控制以及其他信号捕获中断功能。有些 GPIO 的引脚可能与其他功能复用。

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。

9.3 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。端口释放时, 应通过函数 `iounmap()` 取消 `ioremap()` 所做的映射。两个内核函数的原型如下:

```
void * ioremap(unsigned long phys_addr,
               unsigned long size,
               unsigned long flags);
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。

9.4 LED 控制

在 BB–Black mini–USB 接口附近, 有四个可供用户控制的 LED, 他们来自 GPIO1 模块的 21∼24 引脚。我们可以通过下面的方式控制 usr0 LED(GPIO1_21):

```
#define GPIO1_BASE     0x4804C000
#define GPIO_OE        (GPIO1_BASE+0x134)
#define GPIO_IN        (GPIO1_BASE+0x138)
#define GPIO_OUT       (GPIO1_BASE+0x13C)
......

volatile int *pConf    = ioremap(GPIO_OE, 4); /* 映射方向寄存器 */
volatile int *pDataIn  = ioremap(GPIO_IN, 4); /* 映射输入寄存器 */
volatile int *pDataOut = ioremap(GPIO_OUT, 4);/* 映射输出寄存器 */

    *pConf    &= ~(1<<21);                /* 将pin21设为输出      */
    *pDataOut |=  (1<<21);                /* pin21 置高电平, 灯灭*/

    *pDataOut &= ~(1<<21);                /* pin21 置低电平, 灯亮*/

    ......
```

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))

......
```

9.5 实验内容

BB–black 的扩展连接器 P8、P9 引出了大量的 GPIO 以及其他可编程功能模块。根据硬件接口资料, 完成任意一个设备的基本控制功能 (包括驱动程序和用户程序), 实现人–机交互以及相关模块的扩展功能。

实验 10

Qt/Embedded 移植

10.1 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。

10.2 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 的图形界面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 630KB, 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。

10.2.1 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 ftp://ftp.trolltech.com/qt/source 或者 https://www.qt.io 下载 Qt/Embedded 的某个版本的源代码。本实验推荐版本是 qt-embedded-2.3.7.tar.gz;

- Qt 桌面环境 qtopia-free-1.7.0.tar.gz。

10.3 Qt/E 编译

10.3.1 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` — 包含交叉编译工具 `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
```


10.3.2 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8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。

10.3.3 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 5 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, 加入上面的脚本命令。

10.3.4 Qt-4.8 版本编译

目前的 Qt 最新版本是 5.9(2017 年 7 月发布)。新版本支持更多的特性、更好的人机交互体验, 例如支持触摸屏的滑动和多点触控等等。但编译相当耗时。

Qt 以商业版权和 LGPL 版权两种版权协议发布。这里以 Qt-4.8.4 的 opensource 版本为例介绍 Qt 的编译移植过程。

与低版本编译顺序类似, 应先编译触摸屏库, 将其安装在指定目录, 并在 Qt 解压目录下面的 mkspecs/qws/linux-arm-g++/qmake.conf 文件中添加如下几行:

```
QMAKE_INCDIR    = ~/work/build/include
QMAKE_LIBDIR    = ~/work/build/lib
QMAKE_LIBS      = -lpng -lz -lts
```

~/work/build 目录为 libpng, libz 和 libts 的交叉编译安装目录。配置 Qt 编译环境如下命令:

```
$ ./configure \
    -prefix ~/work/build \
    -opensource \
    -confirm-license \
    -release -shared \
```

```
-embedded arm \
-xplatform qws/linux-arm-g++ \
-depths 16,24,32 \
-fast \
-optimized-qmake \
-no-pch \
-no-largefile \
-qt-sql-sqlite \
-system-zlib -system-libtiff -system-libpng -system-libjpeg \
-qt-freetype \
-no-qt3support \
-no-mmx -no-sse -no-sse2 \
-no-3dnow \
-no-openssl \
-no-opengl \
-webkit \
-no-phonon \
-no-nis \
-no-cups \
-no-glib \
-no-xcursor -no-xfixes -no-xrandr -no-xrender \
-no-separate-debug-info \
-make libs -make examples -make tools -make docs \
-qt-mouse-tslib -qt-mouse-pc -qt-mouse-linuxtp
```

编译完成后, 将 ~/work/build 目录平移到目标文件系统, 仿照低版本的按照方式设置环境变量。

10.4 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。

实验 11

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 (动态链接库)。

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-1.2.8.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`, 该路径一方面需要 root 权限, 另一方面, 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, 可能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-linux \
        --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 `/usr/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-1.2.8 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 --host=arm-linux
$ make && make install
```

4. 进入 MPlayer 解压目录, 进行如下配置:

```
$ ./configure \
    --cc=arm-linux-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 目录里。

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。

实验 12

实时操作系统 RTEMS

12.1 实验目的

了解实时操作系统的特性

12.2 实时操作系统 RTEMS 简介

许多嵌入式应用对任务的响应时间和处理时间都有非常严格的要求。实时操作系统 (Real-Time Operating Systems) 就是基于这样的背景发展起来的, 它属于嵌入式操作系统的一类。其核心特征是实时性, 而实时性的本质是任务处理所化费时间的可预测性, 即任务需要在规定的时限内完成。

遵循 GNU 公共版权协议的开源操作系统 RTEMS(Real-Time Executive for Multiprocessor Systems) 属于硬实时嵌入式操作系统。该项目最初起于 1980 年代, 用于军事目的。其中的字母 “M” 从导弹 (Missile) 到军事 (Military) 演化到现在的 “多处理器” 的概念。它支持包括 POSIX 在内的多种开放 API 标准, 移植了包括 NFS 和 FATFS 的多种文件系统和 FreeBSD TCP/IP 协议栈。最近的版本 4.10.2 [1] 支持包括 X86、ARM、MIPS、SPARC, POWERPC, TI-C3X 等在内的数十种处理器架构。

12.3 编译 RTEMS

1. 下载 RTEMS 源码 (https://www.rtems.org)。
2. 编译工具链
3. 编译内核
   (a) 进入 rtems 主目录, 执行./bootstrap
   (b) 进入 c/src/lib/libbsp/arm/beagle, 修改与 BB–Black 相关的代码
   (c) 在 rtems 主目录下建立 build 目录, 在 build 目录执行编译配置

[1] 2015 年 7 月

```
$ ../cofigure --target=arm-rtems4.10 \
    --enable-posix
    --enable-itron \
    --enable-networking \
    --enable-cxx \
    --enable-maintainer-mode \
    --enable-rtemsbsp="beagleboneblack" \
    --enable-tests=samples \
    --prefix=build_install
```

正确完成以上编译后, 会在 `build_install` 目录下生成 RTEMS 内核的二进制文件和一些静态库, 在 `build` 目录下生成可独立运行的二进制文件如 `hello.exe`、`ticker.exe` 等等。他们来自 `testsuites/samples` 目录下的一些小程序源码, 缺省的 `configure` 配置下被编译。

仿照 Linux 内核映像的生成过程, 制作可执行程序的映像文件。

```
$ arm-rtems4.10-objcopy -O binary --strip-all hello.exe hello.bin
$ gzip -9 hello.bin
$ mkimage -A arm -O rtems -T standalone -a 0x80000000 -e 0x80000000 \
          -n RTEMS -d hello.bin.gz hello.img
```

将 `hello.img` 复制到 TF 卡的 boot 分区, 并在 u–boot 的配置文件中指定加载的文件。

12.4 编译内核映像 RKI(rtems kernel image)

上面编译出的 `.exe` 文件是孤立的任务, 如无人–机接口, 不能直观地看到运行结果。RKI 可以在 RTEMS 操作系统上创建一个 shell, 从而实现交互任务。

1. 下载 rki (https://github.com/alanc98/rki)
2. 编译

   ```
   $ make ARCH=arm-rtems4.10 BSP=beagleboneblack \
       RTEMS_BSP_BASE=rtems/build_install
   ```

   其中 `rtems/build_install` 是 RTEMS 内核编译后的安装目录 (`--prefix` 指向的目录)。

正确完成编译后会生成 `rki.bin`。使用 `mkimage` 工具制作内核映像 `kernel.img`, 将其复制到 TF 卡 boot 分区并用 u–boot 加载。上电、启动, 用串口终端进入 shell。

12.5 实验报告要求

- 总结 RTEMS 移植方法
- 尝试在 RTEMS 系统中编写一个多任务程序

```
reading kernel.img
367540 bytes read in 37 ms (9.5 MiB/s)
## Booting kernel from Legacy Image at 80800000 ...
   Image Name:   RTEMS
   Image Type:   ARM RTEMS Kernel Image (gzip compressed)
   Data Size:    367476 Bytes = 358.9 KiB
   Load Address: 80000000
   Entry Point:  80000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
## Transferring control to RTEMS (at address 80000000) ...

RTEMS Beagleboard: am335x-based

RTEMS Kernel Image Booting

*** RTEMS Info ***
COPYRIGHT (c) 1989-2008.
On-Line Applications Research Corporation (OAR).
rtems-4.10.99.0(ARM/ARMv4/beagleboneblack)

BSP Ticks Per Second = 100
*** End RTEMS info ***

Populating Root file system from TAR file.
Setting up filesystems.
Initializing Local Commands.
Running /shell-init.
1: mkdir ram
2: mkrfs /dev/ramdisk
3: mount -t rfs /dev/ramdisk /ram
mounted /dev/ramdisk -> /ram
4: hello
Hello RTEMS!
Create your own command here!
Starting shell....

RTEMS Shell on /dev/console. Use 'help' to list commands.
[/] #
```

图 12.1: RTEMS 终端