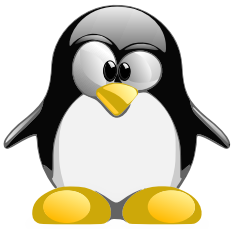


嵌入式 Linux 操作系统实验

南京大学　电子科学与工程学院

撰稿人：方元

2021 年 8 月

目录

插图

程序清单

1

引导加载

嵌入式系统是我们的研究对象。本系列实验中, 我们以 BeagleBone Black 和树莓派3B 作为典型案例分析。在宿主机—目标机开发模式里, 它们是目标机, 或叫做 “目标系统”, 在实验环境下，又习惯地把它们叫做 “开发板”。

1.1 引导加载的作用

PC 机中的引导加载 (Bootloader) 程序由 BIOS和位于硬盘的主引导记录 MBR (**M**aster **B**oot **R**ecorder) 中的 OS Bootloader 共同组成。BIOS 在完成硬件检测和资源分配后, 将硬盘 MBR 中的 Bootloader 读到系统的 RAM 中, 然后将控制权交给 OS Bootloader。Bootloader 的主要运行任务就是将内核映像文件从硬盘上读到 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。

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, 以便完成一些高级语言不能完成的工作。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. 设置堆栈;
4. 跳转到阶段二的 C 程序入口点。

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. 将内核映像和根文件系统映像读到 RAM 空间中;
4. 为内核设置启动参数;
5. 跳转到内核入口点, 内核启动。

Bootloader 也是嵌入式 Linux 系统开发过程中差异最大的地方。以下分别介绍 BeagleBone Black 和树莓派的引导启动过程。

1.2 BeagleBone Black

1.2.1 BeagleBoard 简介

BeagleBone Black 是基于 TI 的嵌入式处理器 Sitara AM335X 设计的单板计算机, Cortex-A8 架构, 主频 1GHz, 板载 512M DDR3L 内存和 2GiB/4GiB eMMC FLASH 存储器, 3D 图形引擎 SGX530, 采用 6 层板工艺设计。元器件布局见图1.1, 板载接口及功能列表如下:

- 有线以太网: 10/100M,RJ45, LAN8720
- microSD 卡接口 (3.3V)
- 电源管理: TPS65217PMIC
- USB Client(USB0, mini-USB),
- USB HOST(USB1)
- HDMI 高清视频输出接口 (Max. 1920x1080@24Hz)
- 音频输出: 通过 HDMI
- 3D 高性能图形加速 SGX530
- UART0 调试接口
- 两组 2×23 扩展接口: McASP, SPI, I2C, LCD, MMC, GPIO, ADC 等

BeagleBoard 出厂时已安装了 Bootloader (U-Boot) 和 Linux 操作系统。如果需要从外接 microSD 卡上启动系统, 或者重新安装 Bootloader, 就需要编译 U-Boot 了。

1.2.2 编译 U-Boot

很多嵌入式 Linux 使用 U-Boot 引导。U-Boot 源码在 https://gitlab.denx.de/u-boot, 使用下面的命令克隆项目:

```
$ git clone https://gitlab.denx.de/u-boot/u-boot.git
```

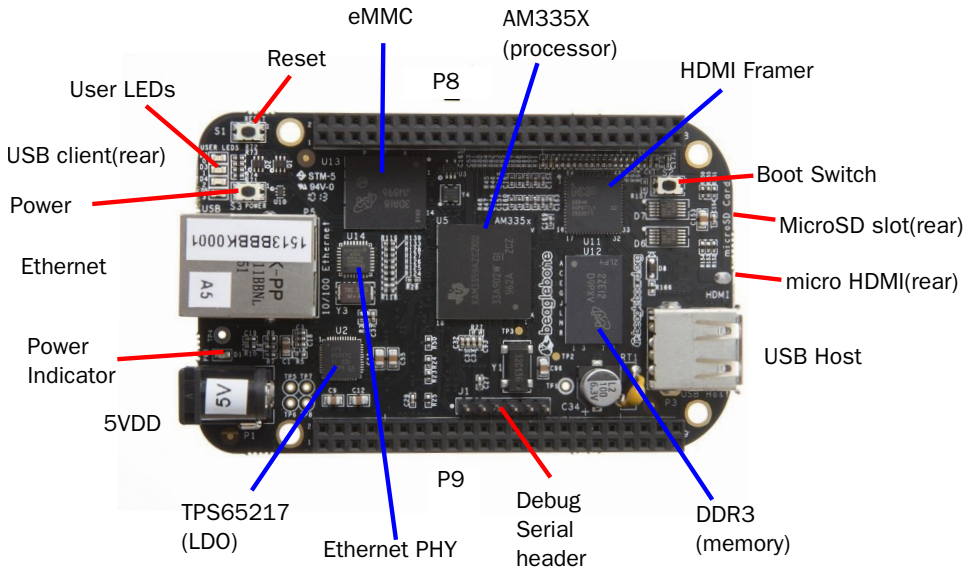


图 1.1: BeagleBone Black 布局

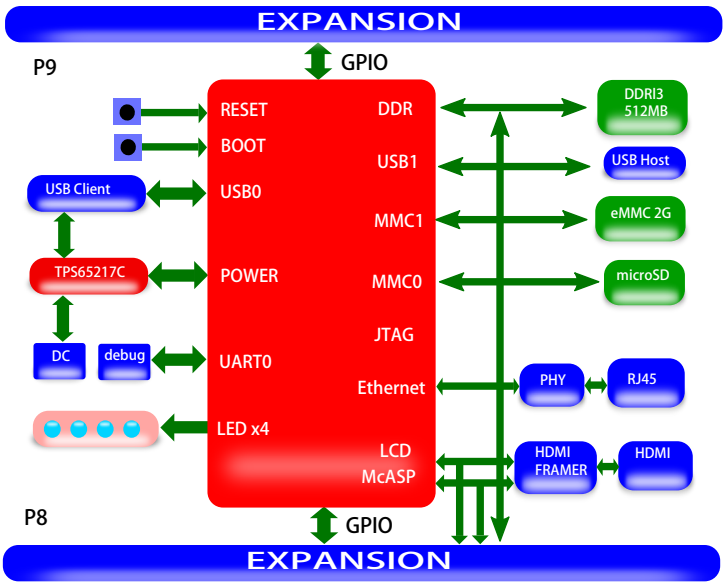
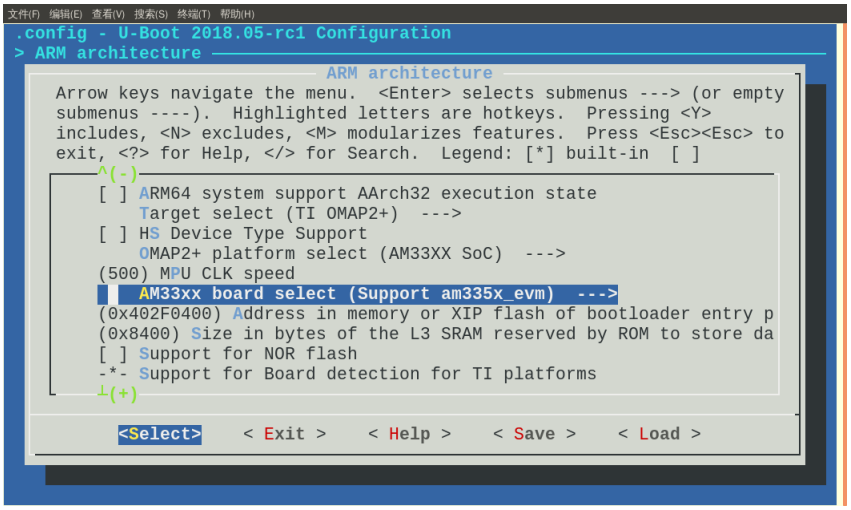


图 1.2: BeagleBone Black 框图

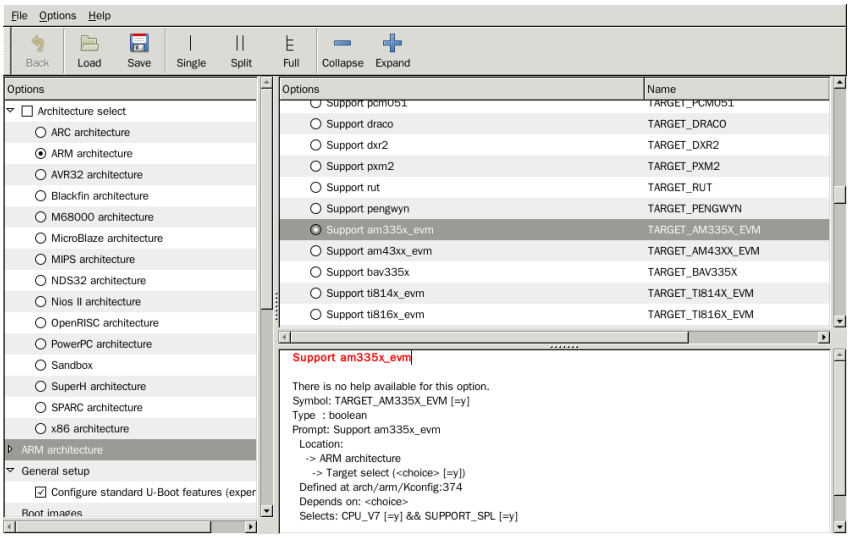
BeagleBone Black 使用 am335x_evm_defconfig 的缺省配置文件。在下载的 U-boot 目录中, 执行 `make am335x_evm_defconfig`, 即完成针对 BeagleBone Black 的缺省 Bootloader 配置。在此基础上, 还可以用 `make menuconfig` 或 `make gconfig` 进入配置界面, 对 Bootloader 的单个选项进行合理的取舍。前者为字符菜单界面 (图1.3a), 后者是图形配置界面 (图1.3b)。完成配置后, 用下面的命令编译:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-
```

编译完成后, 生成 MLO 和 u-boot.img。



(a) make menuconfig



(b) make gconfig

图 1.3: U-Boot 配置界面

1.2.3 制作 microSD 卡

将前面生成的 MLO 和 u-boot.img 用下面的命令写入 microSD 卡:

```
# 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
```

设备名 /dev/mmcblkX 因时而异。如果通过 USB 转接器转接, 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。BeagleBone Black 上电时按住 S2 (图1.1 Boot Switch), 系统首先选择 MMC0 (microSD 卡插槽) 启动。

再在该卡上创建两个文件系统分区。一个引导分区, 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 Linux 分区, 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)。内核映像文件一般不超过 10MB, RAM

Disk 映像文件、设备树文件等等也在 10MB 以内, 可以以此估算 VFAT 分区的大小。剩余空间作为根文件系统, 按 Ext4 分区格式化。图1.4是在一个 2GB microSD 卡上规划的分区方案。

```
命令(输入 m 获取帮助)： p
Disk /dev/sdb：1.9 GiB，2032664576 字节，3970048 个扇区
单元：扇区 / 1 * 512 = 512 字节
扇区大小(逻辑/物理)：512 字节 / 512 字节
I/O 大小(最小/最佳)：512 字节 / 512 字节
磁盘标签类型：dos
磁盘标识符：0xc66ac9af

设备            启动    起点     末尾     扇区   大小 Id 类型
/dev/mmcblk0p1        2048   133119   131072   64M  c W95 FAT32 (LBA)
/dev/mmcblk0p2      133120 3970047 3836928  1.8G 83 Linux
```

图 1.4: 2GB microSD 卡的分区方案 (使用 fdisk /dev/sdb 命令)

注意在分区时不要覆盖之前写入 microSD 卡的裸数据所在扇区 (即第一个分区的起点不要太低)。保存分区表、退出, 用下面的命令格式化:

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

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。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
```

以上表示把 eMMC 设备 0 分区的 zImage 文件读入内存 0x82000000、dtbs/am335x-boneblack.dtb 读入 0x88000000, 指定根文件系统/dev/mmcblk0p2, 设置启动参数 `bootargs`, 最后通过 `bootz` 命令启动内核。`bootz` 可带三个参数, 依次是内核地址、根文件系统地址、设备

树地址。当某项参数缺失时, 用符号 “-” 填充。此处 `bootz` 的第二个参数由 `bootargs` 的 `root` 参数指定。

如果需要把 Bootloader 刷到板载 eMMC FLASH, 可在 Beagleboard Linux 系统启动后 (外扩 MMC0 启动或板载 eMMC 启动), 使用 dd 命令把编译 U-Boot 生成的文件 MLO 和 u-boot.img 写入 eMMC 设备。

1.2.4 U-Boot 用法

U-Boot 的主要任务是引导操作系统。但在开发阶段, 连接串口监视、系统上电后的短时间内, 在 minicom 窗口通过人工干预, 也可以进入 U-Boot 的交互环境, 出现 “`U-Boot#`” 提示符, 在这个提示符下通过人工操作尝试系统启动。当确定能够完成正确的启动后, 再把这些操作写入 uEnv.txt 文件, 让 U-Boot 上电后自动执行。

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: 本机 IP 地址, serverip: TFTP 服务器 IP 地址, gatewayip: 网关。如果本机和服务器在同一个子网, 可以不设 gatewayip。

    ```
    U-Boot# set ipaddr 192.168.2.123
    U-Boot# set serverip 192.168.2.23
    U-Boot# set gatewayip 192.168.2.1
    ```

    U-Boot 的命令具有自动补全功能, 即如果开头的几个字母能指向一个唯一存在的命令, 敲入制表符键 TAB, 这条命令的后面几个字母就自动填补上了; 也可以不用制表符键补全, 直接用开头几个字母作为命令的替代 (但需要确定其唯一性)。这里的 “set” 即表示 “setenv”。
  - 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:

    ```
    U-Boot# set bootargs console=ttyO0,115200n8 root=/dev/mmcblk0p2 rw
        ↪ rootfstype=ext4 init=/linuxrc
    ```

    参数格式 “arg=val”, 参数项之间用空格分隔。上面表示用串口设备 ttyO0 作为终端, 波特率 115200bps, 无校验位, 8 位数据位, 根文件系统在板载 eMMC FLASH 的第二分区, 以读写允许方式挂载, ext4 文件系统, 内核启动后执行根目录下的 linuxrc 命令。
  - bootcmd, 启动命令, 上电或者执行 boot 命令时调用。如

    ```
    U-Boot# set 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。
  - bootdelay: 启动延迟秒数, 在上电后的这段时间里可以通过键盘干预, 打断正常引导过程。
- tftpboot: tftp 文件传输, 如:

```
U-Boot# 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: 将设置的参数保存到 Bootloader 的数据区。未经保存的参数, 重启后将恢复原状。
- printenv: 打印当前已设置的参数。
- help: 帮助命令, 可以打印 U-Boot 支持的所有命令清单以及每个命令的简短说明。

由于 U-Boot 是可配置的, 一些不常用的命令可以在编译阶段排除, 以起到精简的作用。

1.3 树莓派

1.3.1 简介

树莓派是英国一个非盈利机构树莓派基金会开发的一种卡片式计算机, 最初的目的是用于对少年儿童进行计算机普及教育。由于其开放的设计、模块化结构及良好的性价比, 它被广泛应用于多种嵌入式领域, 广受电子爱好者的欢迎。

树莓派采用博通公司的处理器芯片 BCM283X 作为核心处理器, 2 代之前处理器架构是 Arm 32 位的 Arm1176jzf/Cortex-A7, 3 代之后采用 64 位四核 Cortex-A53/Cortex-A72, 最高主频达到 1.5GHz, 还带有 VideoCore-IV 图像处理单元 GPU (**G**raphical **P**rocessing **U**nit)。树莓派 3 代之后开始支持板上无线网络和蓝牙, 并可以通过 HDMI (**H**igh **D**efinition **M**ultimedia **I**nterface) 接口输出高清视频。部分片内设备通过一组 2×20 引出。

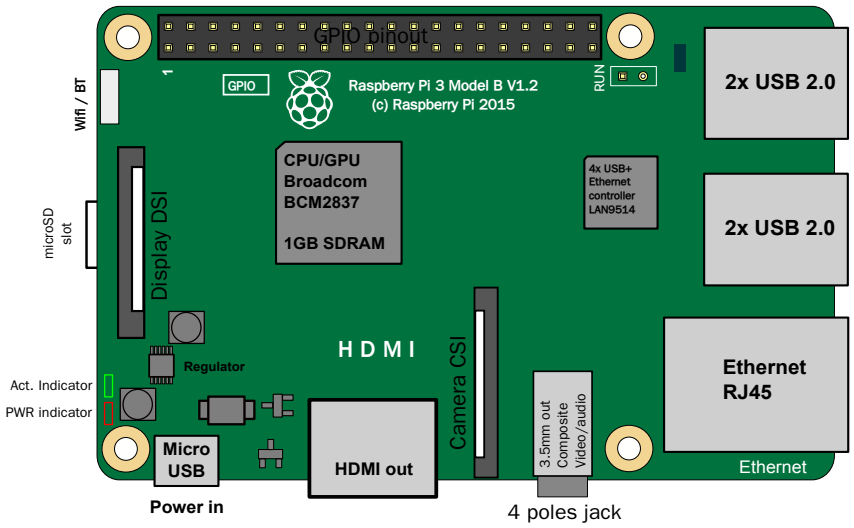


图 1.5: 树莓派 3B

1.3.2 安装 Bootloader

树莓派没有板载 FLASH, 全部软件都要安装在 microSD 卡上。同 BeagleBone Black 一样, 将 microSD 卡划为两个分区, 第一个分区作为引导分区, 格式化成 VFATFS, 第二个分区格式化

成 Ext4FS, 作为 Linux 系统的根分区。由于 Arm 64 位 Linux 内核暂不支持压缩格式, 引导分区需要大一些。

树莓派的引导程序是 Berryboot, 固件由博通公司开发, 不属于开源软件。但二进制代码可以通过 https://sourceforge.net/projects/berryboot/免费获得。只需要将下载的二进制代码压缩包解压到 microSD 卡的引导分区即可。Berryboot 的脚本文件是 config.txt, 其中有系统时钟设置、GPU 内存分配、显示器设置、内核引导方式等等。与内核启动相关的设置如下:

```
initramfs initramfs.gz followkernel
kernel kernel8.img
cmdline cmdline.txt
device_tree=bcm2710-rpi-3-b.dtb
```

其中 kernel8.img 是内核映像文件, initramfs.gz 是压缩的初始化 RAM Disk 映像。Bootloader 会自动识别设备型号, 并根据型号加载设备树文件, 因此这里的设备树文件也可以不写。传递给内核的启动参数由文件 cmdline.txt 给出。向内核传递启动参数与在 U-Boot 中设置 bootargs 性质相同, 只是手段的差别。一个典型 cmdline.txt 文件内容如下:

```
dwc_otg.lpm_enable=0 console=tty1 console=ttyAMA0,115200 root=/dev/mmcblk0p2
    ↪ kgdboc=serial0,115200 rootfstype=ext4 elevator=deadline fsck.repair=yes
    ↪ rootwait
```

这些参数相当于内核启动的命令行, 必须写在一行。传递给内核的参数, 有些是内核可识别的, 直接在内核启动过程中处理; 有些在操作系统的初始化过程 (init 进程) 处理。这些参数可以从 PROCFS 文件系统 /proc/cmdline 文件中读到。

树莓派 3 型需要在 config.txt 中用下面两行指定作为调试的串口位置:

```
enable_uart=1
dtoverlay=pi3-miniuart-bt
```

完整的 Berryboot 包含一个精简版 Linux 系统, 它提供一个图形界面, 帮助用户通过互联网下载安装不同的操作系统。

1.4 实验内容

制作一个可启动的 microSD 卡, 暂时先借用一个可用的内核映像文件。通过串口监控到内核的启动过程, 这个实验的目的就达到了。在后面的实验中, 逐渐用自己编译的内核和文件系统映像代替。

2

嵌入式 Linux 开发环境

Linux 诞生于 1991 年, 是遵循开源版权协议的操作系统。该系统是由芬兰学生 Linus Torvalds 出于学习兴趣而设计的。最初的系统只支持 X86。由于它的开放性, 迅速得到了广大计算机爱好者的响应, 其影响力迅速扩大。至 2017 年, 全球超级计算机 500 强都已采用 Linux 操作系统。Linux 内核已被移植到数十种处理器架构和几百个硬件平台。大到超级计算机、小到可穿戴设备, 都有 Linux 操作系统的身影。它已成为嵌入式领域中最重要的操作系统之一。

2.1 Linux 操作系统的特点

多数场合中, 我们提到的 “操作系统”, 是指面向消费者的、可用的、一个完整的系统, 它包括以操作系统内核为中心的系统软件和大量的应用软件。但对于嵌入式系统则不同。嵌入式系统是嵌入在产品中的计算机, 消费者面对的是产品, 而不是计算机。由于产品在体积、形状、成本、能耗等方面的要求, 其作为控制的计算机系统必须经过专门的设计, 与通用计算机的软件有很大不同。极端情况下, 嵌入式操作系统可能就是指运行在这个系统上的唯一程序。

而当谈及 Linux 操作系统时, 也可能包含两个意思: 一是指以 Linux 内核为中心、由大量的 GNU 软件 (也可以包括其他遵循开源版权协议的软件) 构成的一个完整的桌面系统。其实这种情况, 用发行版名称称呼这个系统更为确切, 如 Debian、Raspberry Pi OS 等等; 而另一种情况则更偏重于其内核, 如 Android 操作系统, 它也是以 Linux 为内核, 但不属于 Linux 操作系统。嵌入式 Linux 通常属于后一种情况, 它以 Linux 为内核, 但通常没有公认的发行版名称。由于产品的多样性, 嵌入式 Linux 操作系统尚未形成一个统一的标准, 它们之间的最大共性就是拥有一个相同名称的内核。

开发嵌入式系统的软件, Linux 有得天独厚的优势:

1. Linux 内核采用 GPL(GNU **G**eneral **P**ublic **L**isence) 开源版权协议, 是自由软件, 不存在商业软件的诸多限制。只要遵守版权协议, 就可以移植、研究、以及再发布。Linux 世界有丰富的开源软件支持, 可以满足各种应用场合的需要。
2. Linux 向来以稳定性和可靠性著称。Linux 内核空间和用户空间的分离、进程地址空间的分离、用户之间的权限设计, 多种安全技术措施共同保障系统安全。由于软件是开源的, 不

存在暗箱, 恶意代码很难有容身之地; 世界各地的软件工程师和 Linux 用户都在热心地为开源社区提出建议, 一旦出现 bug, 可以很快得到修正。

3. Linux 具有良好的可移植特性。Linux 内核支持包括 X86、Arm、PowerPC 等几十种主流处理器架构, 大量的软件遵循 POSIX及 ISO 相关设计标准, 在不同的平台之间不经修改或只需要少量的修改就可以直接使用, 为嵌入式软件开发带来了很大方便。
4. Linux 系统的设备独立性简化了设备的使用, 提高了外设资源的利用率。Linux 系统将所有资源均视为文件 (即所谓的 “一切皆是文件”), 所有设备也被当作文件统一处理。应用程序像使用文件一样, 操纵、使用这些设备, 而不必了解设备的具体细节。Linux 内核支持目前所有已知的磁盘分区格式 (多文件系统支持), 此外还支持 FLASH 文件系统和内存文件系统, 扩展了嵌入式应用的存储能力。
5. Linux 系统具有完善的网络功能。Linux 内核实现了完整的 TCP/IP 网络协议, 具有强大的网络通信能力。在万物互联的时代, Linux 的网络功能为系统通信提供了丰富的支持。
6. Linux 可以使用多种开发手段, 这得益于 Linux 世界的开放性。Linux 提供了多种编程语言开发工具, 此外还大量地使用了脚本语言, 除了方便编程以外, 也大大方便了系统的维护。脚本语言不需要专门的开发工具, 程序员或者服务器维护人员只需要用自己熟悉的文本编辑工具就可以完成自己的工作。

由于 Linux 操作系统的源代码是完全公开的, 它不仅可以作为目标系统的软件, 应用于嵌入式产品, 其本身也是学习操作系统、计算机软件和硬件的良好素材。

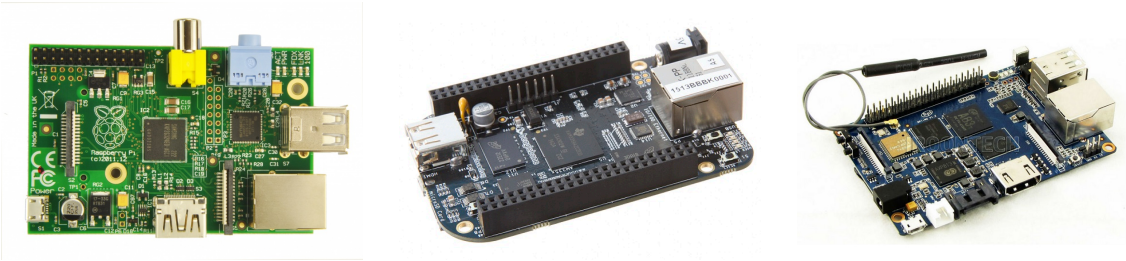
2.2 嵌入式硬件平台

虽然 Linux 操作系统在嵌入式系统应用中有很多优势, 但它对硬件平台有一定的要求。首先它是基于 32 位处理器设计的, 后来发展到支持 64 位系统, 8 位或 16 位的单片机系统无法支持; Linux 最初的设计目标是桌面计算机, 目前也要求处理器有足够的速度、充足的存储空间。虽然也有不少开发人员将 Linux 内核向低端移植, 但高端处理器仍然是它的主要目标, Arm 处理器的 Arm9 以上以及 Cortex-A 系列才能较好地满足 Linux 的运行环境。

另一方面, 原生的 Linux 不是实时操作系统, 而是分时操作系统。分时操作系统的主要设计目标是充分发挥系统资源的整体效率, 而不是将目标聚焦在完成某个任务的时间效率上, 在有实时应用的需求时必须通过特别的措施加以保证。

一些开发人员在公版 Linux 基础上进行了实时性移植。其中比较著名的有 Linux 基金会维护的 Real-Time Linux 和新墨西哥数据挖掘与技术学院的 RTLinux。RT 为 Real Time 的缩写, 二者目标相同, 名称相似, 因此常常被混淆。但它们是两个独立移植的系统, 实现实时性的方法也不尽相同。

2012 年 2 月, 英国的一个非盈利机构树莓派公司推出了一款信用卡大小的单板计算机, 核心处理器是博通公司基于 Arm11 设计的 SoC, 可以运行 Linux 操作系统, 其设计目标是面向中小学及发展中国家的计算机基础教育。这款名为 “树莓派” 的单板计算机迅速受到广大计算机爱好者的关注, 其他一些科技公司也先后开发了类似低成本、高性能的单板计算机。



(a) RPi-B(2012.2) (b) BB-Black(2013.4) (c) BananaPi-M3(2015.12)

图 2.1: 各种派

这些单板计算机的共同特点是, 一、处理器性能强大, 主频工作于几百兆甚至上千兆赫兹, 有存储器管理能力, 并且已经开始向 64 位发展; 二、内存容量大, 板载内存通常在几百兆字节到几个吉字节, 具备支持高级操作系统的能力; 三、具有网络甚至无线网络的通信能力; 四、扩展能力强, 包括标准的 USB 设备支持、高性能显示输出以及大量的片内设备的引出。尤其是扩展能力, 给了爱好者发挥的巨大空间, 也是它们能在嵌入式应用中体现其优势的地方。

这些板卡原则上可以运行多种操作系统, 例如树莓派的官方网站就提供过 Android、Windows 10 IoT Core, RISC OS 等不同操作系统。但大量的应用仍然是基于 Linux 操作系统开发的。因此本书也是围绕 Linux 操作系统的开发而展开。

2.3 嵌入式系统的软件构成

由于嵌入式系统设计的千差万别, 在不同平台上的开发会有一定的差异性, 但仍然是有迹可循的。掌握这些规律, 有助于我们灵活应对不同的平台。

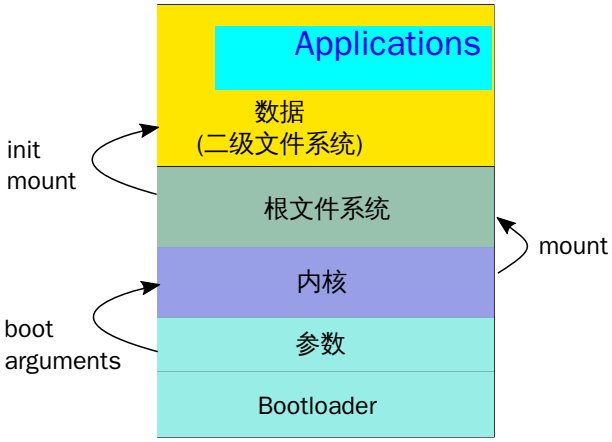


图 2.2: 嵌入式 Linux 系统的软件层次

从开发的角度看, 嵌入式 Linux 系统上开发应用需要按下面的步骤逐步建立完成:

1. 引导加载器(Bootloader)。这部分不属于操作系统, 但对于操作系统的启动是非常重要的。它负责操作系统的启动方式, 向 Linux 内核传递启动参数, 有时还负责操作系统核心的升级。

2. 移植 Linux 内核, 并通过 Bootloader引导。内核可以作为一个映像文件存在于某个磁盘分区上或者网络的某个位置, 也可以是存放于只读存储器某个区域内的数据。Bootloader 将文件或数据读入嵌入式系统的内存, 并将程序运行指针转到内核的启动位置, 从而引导内核的启动。
3. 制作根文件系统。根文件系统在 Linux 内核启动的后期自动挂载。这里所谓的 “自动” 不是指必然挂载成功, 需要根据 Bootloader 传递的参数找到正确的根文件系统的位置。

   根文件系统需要具备 Linux 系统工作的基本环境, 它包括 Linux 系统环境的初始化、建立用户环境、允许用户登录并使用一些基本的操作命令。如果根文件系统是 RAM Disk, 系统启动后还应该考虑挂载只读存储设备 (如 FLASH、硬盘等) 的二级文件系统, 并通过 `switch_root` 将根文件系统切换到只读存储设备上。

   当根文件系统上移植了 Linux 的基本命令后, 这些工作可以通过一些脚本程序完成。
4. 开发应用软件。依照嵌入式系统的不同应用场景, 开发或移植相关的软件。其中可能还会涉及一些库的移植、设备驱动程序的开发。

以上各层软件既相互依赖, 也相对独立: Linux 内核正常启动后, Bootloader 的使命即告完结, 其设置的参数 (如网卡 IP 地址) 不会传递给 Linux, 甚至网卡能否工作都取决于 Linux 内核的工作情况, 与 Bootloader 无关; 根文件系统如何制作, 与内核版本无关, 只取决于内核的支持方式; 大多数应用软件也与内核版本无关, 少数情况有可能不被内核支持, 例如陈旧的内核缺少新软件所需要的系统功能调用或设备驱动。

2.4 建立开发环境

嵌入式系统与通用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 (**B**asic **I**nput/**O**utput **S**ystem), 因此基本输入输出系统需要开发人员自己设计;
- 嵌入式系统缺乏友好的人机界面, 为软件开发和调试带来困难;
- 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。

2.4.1 软件环境

开发嵌入式 Linux 最重要的软件是交叉编译器。

Linux 操作系统内核是 C 语言写的, 大量的系统软件和应用软件也是 C 语言写的, C 语言理所当然地成了 Linux 系统开发的最重要的工具。在 Linux 世界, GCC 是 C 语言编译工具的不二之选, 并且 GCC 也不仅仅是 C 语言的编译器, 它还可以支持 C++、Fortran、Java 等其他一些编程语言。

开发环境是个人计算机, 实现的应用产品目标是在嵌入式系统上, 我们将这种开发模式称为宿主机—目标机模型。宿主机即通用计算机, 目标机就是我们的开发对象。图2.3是开发嵌入式系统的一种典型连接方式。

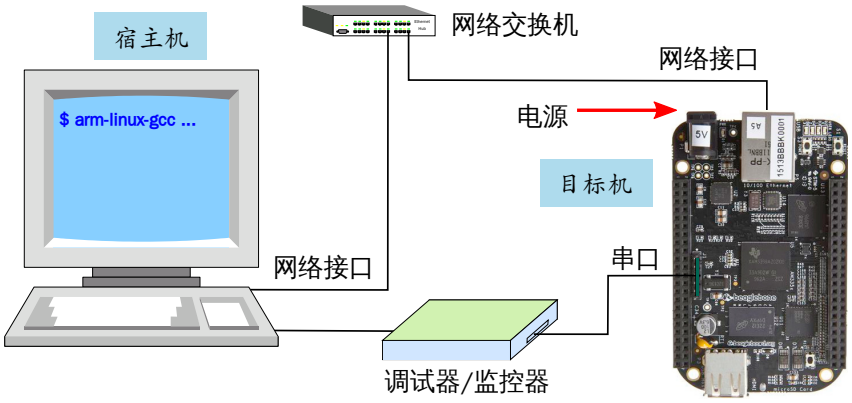


图 2.3: 嵌入式系统开发: 主机与目标系统的连接

在嵌入式系统上, 处理器指令集与个人计算机使用的 X86 指令集不兼容。因此, 简单地使用宿主机上的编译器编译出来的二进制代码是不能在目标平台上运行的, 这里需要用到一种特殊的编译方式—交叉编译。交叉编译工具链是这样一组工具: 它运行在 A 平台 (这里的 A 通常是个人计算机, X86 架构), 它将源程序编译成在 B 平台运行的二进制程序 C (这里的 B 指的是目标平台, 它可能是 Arm 架构、PowerPC架构, 等等), 二进制程序 C 不能在 A 平台运行。大多数情况下, 我们不会在 B 平台使用编译工具, 因为嵌入式系统无论在性能还是可支配资源配置都远不及个人计算机, 况且初期的目标系统根本就没有软件环境。

交叉编译工具链应包含下面几个部分:

- C 语言标准库、C++ 库和 GCC 库;
- 二进制处理工具, 包括汇编器、链接器、代码转换与分析工具等等;
- 编译器集合, 可以支持 C、C++、Objective-C、Java、FORTRAN、GO 等多种编程语言, 但大多数情况下只用到 C 和 C++ 的编译功能;
- C/C++ 的头文件, 以及来自内核但面向用户空间支持的头文件;
- 调试器 GDB (GNU Debugger)。

交叉编译器可以从软件库安装, 也可以自己移植。建议直接安装发行版维护的交叉编译器, 省时省力。在 Ubuntu/Debian 系列的发行版上, 安装 Arm 32 位交叉编译器的命令是:

```
# apt install g++-arm-linux-gnueabihf
```

安装 Arm 64 位交叉编译器:

```
# apt install g++-aarch64-linux-gnu
```

依照依赖关系, 安装包管理器会安装所有 C 和 C++ 的基础包, 包括支持指令集架构版本的库和二进制工具。Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A15 的指令集是 32 位 Armv7-A,

Cortex-A53、Cortex-A72 基于 64 位指令集 Armv8-A。Arm 的 64 位处理器兼容 32 位指令集, 也就是说, 如果不介意 64 位 Arm 处理器跑 32 位系统的话, 都可以按 32 位指令集编译。

在 Linux 各发行版维护的软件仓库中, 32 位编译器前缀是 `arm-linux-gnueabihf-`, 64 位编译器前缀是 `aarch64-linux-gnu-`。为简单起见, 建议将这两组命令各做一组短前缀的软链接`arm-linux-`和 `aarch64-linux-`, 以简化键盘输入工作。类似下面的操作:

```
# ln -s arm-linux-gnueabihf-gcc arm-linux-gcc
# ln -s arm-linux-gnueabihf-g++ arm-linux-g++
...
```

向系统添加软件需要超级用户权限。这里使用 "`#`" 表示超级用户执行的命令。一些系统通过 `sudo` 获取超级用户权限, 也有人通过 `su` 命令切换到超级用户、甚至直接以超级用户身份登录。下文仅通过提示符 "`#`" 和 "`$`" 区别超级用户和普通用户的操作, 而不再讨论超级用户权限的获取方式。此外建议尽量限制超级用户的作用范围, 特别是在开发目标系统软件、不涉及主机设置的工作时, 原则上不应以超级用户权限操作。

由发行版维护的软件, 安装后, 其可执行程序通常会放在环境变量 `PATH` 所列的某个目录中, 因此可以无障碍地运行。如果是自己移植的编译器, 安装的路径可能不在环境变量 `PATH` 所列中 (例如安装到 /opt/armhf-linux-2020 目录), 这就需要把编译器可执行程序安装的路径添加到 `PATH` 中。可以在每次打开终端的时候执行:

```
$ export PATH=/opt/armhf-linux-2020/bin:$PATH
```

一劳永逸的方法是把这一行命令写入个人用户的启动初始化脚本文件.profile 里, 这个文件在个人用户的主目录下, 是一个隐藏文件。

作为开发的个人计算机, 建议使用 Linux 操作系统, 因为针对 Linux 目标系统, Linux 操作系统的开发工具比较丰富, 也比较容易配置, 并且目标系统的软件很多都可以在主机上进行验证。在其他操作系统上安装 Linux 虚拟机也是一种选择, 但需要做好 "虚拟机"—"主机"—"目标机" 的网络连接的设置工作。

由于 Linux 系统的发行版众多, 上层应用软件并不在 POSIX 标准的规范之内, 一些功能在不同发行版中的实现方法不尽相同, 最典型的就是软件包管理软件。本书所涉及的操作是在 Ubuntu 20.04 下实现的。使用其他发行版, 应重点考虑功能, 不必拘泥于命令本身。

2.4.2 硬件环境

串口通信

多数嵌入式系统都通过异步串行接口 UART (**U**niversal **A**synchronous **R**eceiver-**T**ransmitter) 进行初级监控。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。

当 Bootloader 支持串口通信协议时, 宿主机便可以通过串口设备管理和监控目标设备的运行情况。现在的通用计算机系统已基本淘汰了 RS-232C 标准的串行接口, 我们可以使用 USB-

UART转接器实现这项功能。图2.4 是 USB-UART 转接器, BeagleBone Black、树莓派的调试接口位置见图 2.5 和 2.6。使用时, 将转接器的 TxD (发送端) 接目标系统的 RxD (接收端), 转接器的 RxD 接目标板的 TxD, GND 互相连通。目标系统使用自己的供电系统, 转接器的电源不接。其他类似的单板计算机请参照相关说明找到它们的调试接口。

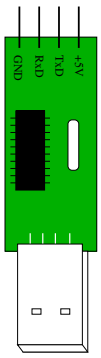


图 2.4: USB-UART 转接器

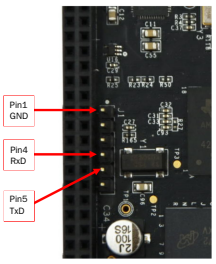


图 2.5: beagleboard 调试接口

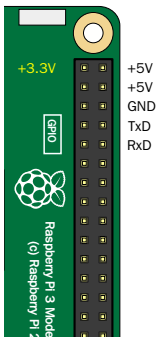


图 2.6: 树莓派调试接口

PC 上插入 USB-UART 转接器, 正确驱动后, 会生成相应的设备文件。使用 `dmesg` 查看设备是否被识别 (不同型号的转接器, 显示内容会有差异):

```
$ dmesg | tail
...
usb 1-3: new full-speed USB device number 18 using xhci_hcd
usb 1-3: New USB device found, idVendor=067b, idProduct=2303
usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-3: Product: USB-Serial Controller
usb 1-3: Manufacturer: Prolific Technology Inc.
pl2303 1-3:1.0: pl2303 converter detected
usb 1-3: pl2303 converter now attached to ttyUSB0
```

再看看是否生成了设备文件 ttyUSBx:

```
$ ls -l /dev/ttyUSB?
crw-rw---- 1 root dialout 188, 0 1月  19 09:12 /dev/ttyUSB0
```

由于该设备属于 dialout 组, 建议将用户添加到这个组, 以免普通用户在操作这个设备时使用超级用户权限。PC 端的 Linux 可以用 `minicom` 或 `screen` 终端命令使用串口设备。推荐使用 `minicom`。

首次使用 `minicom` 时, 用带有 `-s` 的选项启动, 进入设置页面。使用过程中如果要调整工作方式, 随时可以通过组合键 “Ctrl+A o” 调出设置页面 (图2.7)。

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。通过 USB 转 UART 芯片连接串口, 设备是/dev/ttyUSBx;

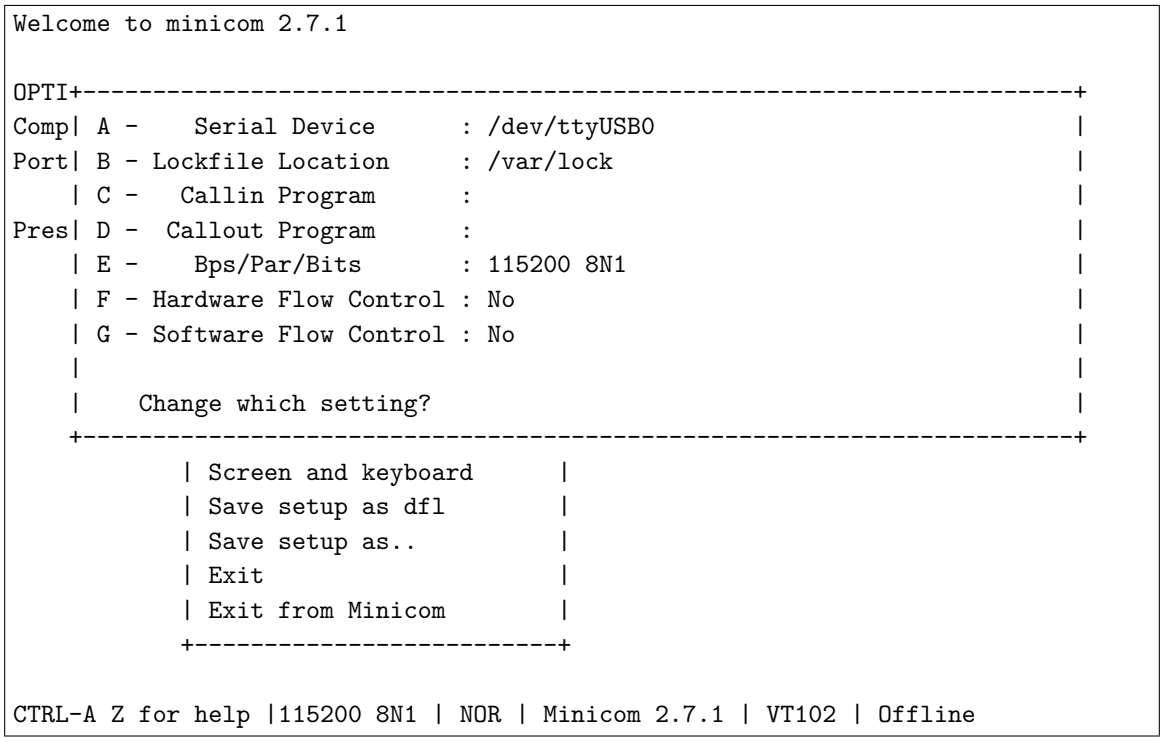


图 2.7: minicom 设置界面

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。这组参数按目标系统的要求设置。树莓派、Beagleboard 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)。多数嵌入式平台要求设置为 “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。目标板上电后, 如果 Bootloader 正常工作, 便可以在 minicom 主界面上看到系统的启动信息。

退出 minicom 的组合键是 “Ctrl+A x”; 如果一行文字过长, 超出终端宽度的信息会被截断, 此时可通过 “Ctrl+A w” (**w**rap/un**w**rap) 在折行/截断之间切换。完整的功能可使用 “Ctrl+A z” 调看 (注意图 2.7底行的提示信息)。

文件传输

串口设备传输能力比较低。以上面设置的串口波特率参数 115200bps 为例, 它只大致相当于每秒传输十几千字节, 如果传输上兆字节的文件会比较耗时。大批量数据传输还需要借助网络功能来实现, 尤其是在 Linux 系统正常工作后, 网络更是有着串口无法替代的功能。

TFTP (**T**rival **F**ile **T**ransfer **P**rotocol) 是一个基于 UDP (**U**ser **D**atagram **P**rotocol) 协议的简单文件传输协议。一些 Bootloader 默认支持这种传输协议 (如 U-Boot)。这种工作模式, 以

主机作为 TFTP 服务器, 目标板的 U-Boot 作为客户端。一些 Linux 的发行版通过超级服务器 xinetd 管理 TFTP。安装 tftp-server 后, /etc/xinetd.d/tftp 中大致是如下内容:

```
service tftp
{
    socket_type = dgram
    protocol  = udp
    wait = yes
    user = root
    server = /usr/sbin/in.tftpd
    server_args = -c -s /var/lib/tftpboot
    disable = yes
    per_source = 11
    cps = 100 2
}
```

将其中的 “disable” 的选项改为 “no”, “server_args” 设置服务器主目录 (此处设置为/var/lib/tftpboot/), 再用如下命令重启 tftp 服务:

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

U-Boot 通过 TFTP 传输文件, 需要完成下面的设置:

服务器端 PC 作为 TFTP的服务器, 将待传输的文件存放在服务器指定的目录 /var/lib/tftpboot/, 同时要注意文件的访问权限, 应开放给所有用户可读;

客户端 U-Boot 设置客户端的 IP 地址及服务器的 IP 地址, 这两个 IP 地址最好设置在同一个子网内。如果跨网段, 还需要设置正确的路由。

以上设置无误后, 可以尝试用 U-Boot 的 TFTP 下载命令传输文件。

若此时已有可用的内核和根文件系统映像文件, 可仿照第 38 页的操作命令, 将内核映像、根文件系统映像和设备树文件下载到 BeagleBone Black 并启动 Linux 系统。

建议由教师事先准备好可用的内核, 供学生熟悉开发过程。

网络文件共享

网络文件系统 NFS (**N**etwork **F**ile **S**ystem) 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, 它可以让微机系统通过网络将远端的 NFS 服务器共享出来的目录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 服务器的共享目录和权限在脚本文件 /etc/exports 中设定。下面是这个文件的样例:

```
# /etc/exports: the access control list for filesystems which
```

```
                may be exported to NFS clients.  See exports(5).
/srv/nfs4    192.168.2.*(rw,sync,no_subtree_check,no_root_squash)
```

它描述了服务器目录、服务的 IP 地址范围以及访问权限。在 PC 上启动 (start) 或重启 (restart)NFS 服务可以用下面的命令:

```
# service nfs-kernel-server [re]start
```

当服务脚本修改后, 使用下面命令使修改生效:

```
# exportfs -a
```

NFS 主要目的是为目标板的 Linux 系统提供存储服务。目标板上的 Linux 启动后, 可以使用类似下面的命令将服务器 (PC 端) 共享的 NFS 目录挂载到本机 (目标板) 的 /mnt 目录 (假设服务器的 IP 地址是 192.168.2.12):

```
# mount 192.168.2.12:/srv/nfs4 /mnt -o nolock,proto=tcp
```

这里使用了来自 BusyBox 的 `mount`命令, 它默认支持 NFS 的 UDP 协议版本, 通过选项强制它使用 TCP (**T**ransfer **C**ontrol **P**rotocol) 版本的 NFS 服务。PC 桌面系统安装的 `mount` 命令不需要这个选项。

以上操作正确完成, 需要: 一、嵌入式系统的 Linux 内核支持 NFS 客户端; 二、嵌入式系统的 IP 地址设置正确, 能够正常访问服务器。这样, 在目标板上访问 /mnt/ 目录时, 实际上访问的是服务器的 /srv/nfs4/ 目录的内容。NFS 服务大大扩展了嵌入式目标系统的存储空间。

在实验室环境下, 宿主机和目标机应通过交换机进行网络连接, 不建议将个人电脑与目标板用网线直连。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。

2.5 实验内容

2.5.1 熟悉交叉编译器

编写一个简单的 C 语言程序 prog.c (功能不限), 分别用主机编译器和交叉编译器将其编译成可执行程序:

```
$ gcc -o prog prog.c
$ arm-linux-gcc -o prog_arm prog.c
```

使用主机的命令 file 看看这两个文件有什么不一样:

```
$ file prog
...
$ file prog_arm
...
```

分别在主机执行这两个程序, 看看会有什么结果。如果有错误, 记录错误现象, 并分析其原因。

2.5.2 学习使用串口工具

通过 USB-UART 转接器连接目标板, 在 PC 端用 `minicom` 打开串口, 设置正确的参数; 给目标板上电, 观察 minicom 窗口的信息。

2.5.3 了解网络环境

Linux 操作系统使用 `ifconfig` 命令查看和配置网络设备。普通用户使用不带参数的 `ifconfig` 命令, 可以查到本机的网络设备名和 IP 地址, 超级用户可以用它改变网络设置。图2.8是一种实验室网络布局, 子网掩码是 255.255.255.0, 网段 192.168.2.0/24。嵌入式实验系统建议按固定地址分配 IP。动态分配的 IP 地址, 实验者可能会找不到自己控制的机器。通过无线路由器接入的笔记本电脑, 连接嵌入式系统时, 需要设置正确的路由和网络地址转换。

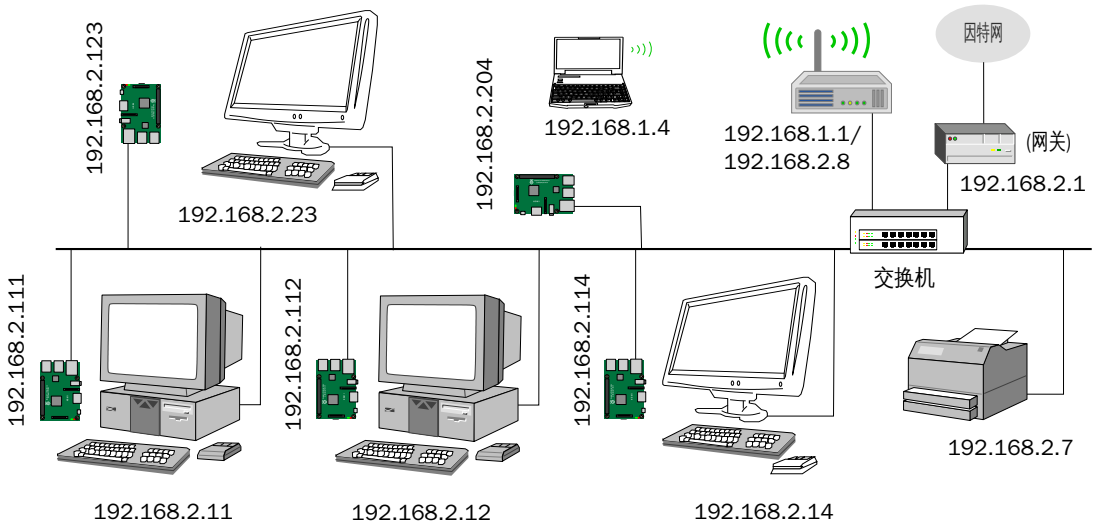


图 2.8: 实验室网络环境布局

如果嵌入式目标板已正常启动了 Linux, 请在目标系统上设置 IP 地址:

```
# ifconfig eth0 192.168.2.123
```

这里假设地址 192.168.2.123 可用且不与其他机器发生冲突。IP 地址分配方法请咨询网络管理员。

检验是否设置正确的方法是, 两边用 `ping` 命令, 各自以对方的 IP 地址为命令参数, 观察其打印输出。如果连通, 会持续打印 ttl (**t**ime **t**o **l**ive) 和响应时间, 如果不通, 则打印 “Destination Host Unreachable”:

```
$ ping 192.168.2.123
PING 192.168.2.123 (192.168.2.123) 56(84) bytes of data.
From 192.168.2.23 icmp_seq=1 Destination Host Unreachable
From 192.168.2.23 icmp_seq=2 Destination Host Unreachable
...
64 bytes from 192.168.2.123: icmp_seq=1 ttl=64 time=1.23 ms
```

```
64 bytes from 192.168.2.123: icmp_seq=2 ttl=64 time=0.741 ms
64 bytes from 192.168.2.123: icmp_seq=3 ttl=64 time=0.725 ms
64 bytes from 192.168.2.123: icmp_seq=4 ttl=64 time=0.742 ms
64 bytes from 192.168.2.123: icmp_seq=5 ttl=64 time=0.714 ms
```

两边连通的情况下, 将上面编译的两个程序 prog 和 prog_arm 复制到 PC 的 NFS 服务目录, 目标板上用 `mount` 命令挂载 PC 的网络文件服务。在目标系统上运行这两个程序, 看看结果有什么不同。如果此时目标系统尚没有正常启动 Linux, 这部分内容推后进行。

请详细描述你的网络实验环境, 标出主机和目标板的网络地址, 解释每条连线在实验中的作用。

3

Linux 内核移植

3.1 内核源码结构

3.1.1 获得内核源码

Linux 通用内核源代码的维护网站是 https://www.kernel.org/pub/linux。此外, 各个嵌入式平台都会有自己的分支。建议在开发时从平台维护的分支上下载移植, 因为平台维护的分支代码更新更快, 也更有针对性。

Beagle Board 的内核维护地址是 https://github.com/beagleboard/linux, 下面的命令将克隆这个项目:

```
$ git clone https://github.com/beagleboard/linux
```

克隆项目的好处是可以在不同版本之间随意切换, 而且当上游代码有更新时, 本地只要 `git pull` 就可以将源码与上游同步, pull 的下载量很小; 缺点是由于代码仓库记录了全部的开发过程, 首次克隆整个仓库的代码量太大。如果只需要在某个特定的版本上编译, 可以利用 github 的网页下载功能, 仅下载某一个版本的压缩包, 然后在本地解压。

树莓派的内核源码在 https://github.com/raspberrypi/linux, 下载方法相同。

3.1.2 内核源码组织方式

内核是一个完全独立的、完整的项目, 它不依赖 Glibc 库。内核源码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: 该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 目录下就是支持 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。其中的子目录 configs 包含了各种平台的缺省配置文件。
- include: 该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, 其中的 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 的系统功能调用函数。
- 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. pc: PCI 卡驱动程序代码, 包含 PCI 子系统映射和初始化代码等;
  4. scsi: 包含所有的 SCSI (**S**mall **C**omputer **S**ystem **I**nterface) 代码以及 Linux 所支持的所有的 SCSI 设备驱动程序代码;
  5. net: 网络设备驱动程序;
  6. sound: 声卡设备驱动程序;
  7. video: 视频显示设备驱动;
  8. ...

- lib: 放置内核的库代码;
- net: 包含内核中与网络相关的代码;
- ipc: 包含内核进程间通信 (**I**nter**P**rocess **C**ommunication) 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、Ext4FS 等。
- scripts: 该目录包含用于配置内核的脚本文件和程序。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赖关系很有帮助。

3.1.3 配置内核的基本结构

Linux 内核的配置系统由下面几个部分组成:

1. Makefile: 分布在 Linux 内核源码中各个子目录中的 Makefile 定义了 Linux 内核的编译规则, 它们决定了文件的依赖关系, 以及与内核的结合方式, 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, 形成菜单界面及其层次关系;
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 (基于逐项字符文本 `make config`, 基于 ncurses 库的字符菜单界面 `make menuconfig`, 基于 GTK+ 库的图形界面 `make gconfig` 和基于 Qt 库的图形界面 `make xconfig` 等等);
4. scripts 目录下的脚本文件负责组织编译工作。

3.1.4 编译规则 Makefile

利用 `make menuconfig` (或 `make config`、`make gconfig`...) 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。由于内核只负责一次备份, 开发人员应自行负责备份正确的配置文件, 便于在需要的时候迅速恢复。

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, 每个配置变量具有三种不同的取值, 在 menuconfig 界面中分别通过按键 “y”、“m”、“n” 设置, 或使用空格键在三种取值间切换:

- 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, 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, 并提供功能的说明, 让用户有机会选择新增功能项。Linux 的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。

3.2 编译内核

3.2.1 Makefile 的选项参数

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

1. `defconfig`: 复制缺省配置文件。对于 Arm 架构来说, 缺省配置文件在 arch/arm/configs 目录中, 它们都以 `_defconfig` 为后缀。Linux 内核可配置的功能有数千项, 即便仅仅是看一遍, 耗费的时间也相当可观。从缺省的配置开始进行修改, 可以大大节省配置的工作量。
2. `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。
3. `clean`: 清除以前构核所产生的所有的目标文件、模块文件以及一些临时文件等, 不产生任何新文件。在开发过程中尽量少用这个命令, 因为清除了中间生成的文件, 完整地编译一次内核还是比较耗时的。
4. `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, 把内核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。如果编译过程没有错误, 将产生文件 vmlinux, 这就是内核映像文件, 同时产生符号地址映像文件 System.map。zImage 和 bzImage 选项是在 make 的基础上产生的压缩内核映像文件。正常编译完成后, 生成的 Arm 内核映像文件在目录 arch/arm/boot 中, 在基于网络的嵌入式开发中需将其移至 TFTP 服务器目录供目标系统下载。
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.ko 模块文件。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/(版本号)/.....。对于嵌入式开发, 当然不能把这些文件复制到 PC 的系统目录, 况且权限也不允许。应通过变量 INSTALL_MOD_PATH 指定复制的目标目录, 移植时再将他们复制到目标系统的 /lib/modules 目录。

由于内核源码支持多种架构, 打开配置界面时, 主目录 Makefile 根据变量 ARCH 决定按哪个架构操作。缺省方式下, ARCH 这个环境变量是空的, Makefile 按主机的架构执行。嵌入式开发中, 要么通过export 设置环境变量:

```
$ export ARCH=arm
```

要么在调用 make 命令时设置这个变量:

```
$ ARCH=arm make menuconfig
```

ARCH 与内核 arch 目录下的子目录名对应。

还有一个方法是直接修改主目录 Makefile 文件中的这个参数, 不过不建议用这种方法。

图3.1a和3.1b分别是 BeagleBone Black 和树莓派配置界面的一页, 请注意最上面一行版本号前面的处理器指令集信息提示。

3.2.2 内核配置项介绍

首先复制缺省配置文件。Beagle Board 的缺省配置文件是bb.org_defconfig, 规范的操作是:

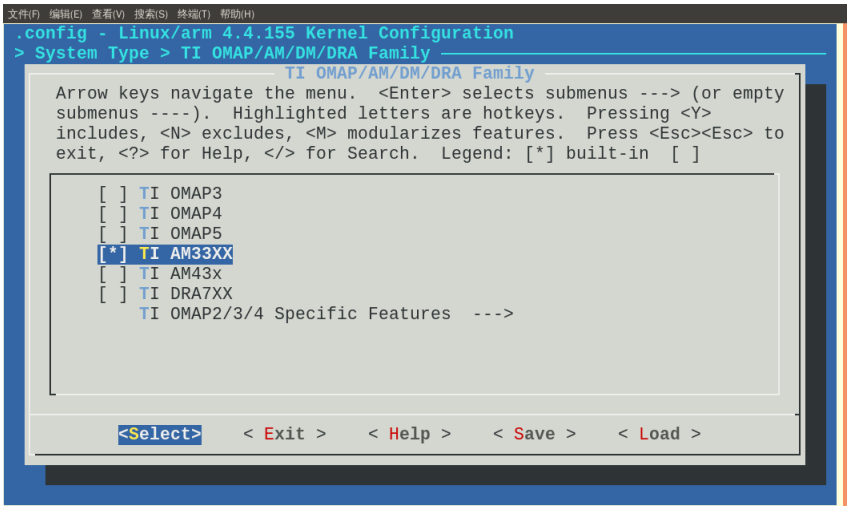
```
$ ARCH=arm make bb.org_defconfig
```

它与直接将这个缺省配置文件复制到内核源码主目录的.config 文件操作是等效的。

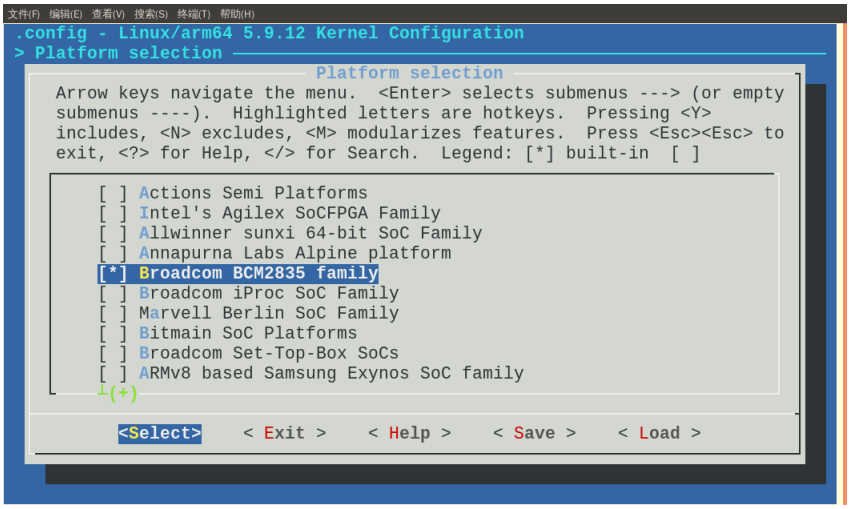
内核配置主菜单项里, 需要重点修改的有下面这些分支:

1. General setup, 内核配置选项和编译方式等等; 嵌入式开发中, 确定下面的选项:

```
General setup  --->
    [*] Initial RAM filesystem and RAM disk
       ↪ (initramfs/initrd)...
    ()    Initramfs source file(s)
    [*]   Support initial ramdisk/ramfs compressed using gzip
```



(a) BeagleBone Black



(b) 树莓派 (64 位指令集)

图 3.1: 配置内核的字符菜单界面

```
    ...
```

让内核支持初始化 RAM 文件系统及 RAM Disk, 并选择其下的一种压缩格式。压缩的目的是减小 RAM Disk 的磁盘占用空间, 因为在嵌入式应用中, RAM Disk 和内核映像一样都要保存在 FLASH 上; 在开发过程中, 如果通过 TFTP 传输, 较小的文件映像也有利于减少网络传输的时间。

如果以 RAM Disk 作为根文件系统, 还应做以下设置:

```
Devices Drivers  --->
    [*] Block devices --->
        <*>   RAM block device support
        (8192)  Default RAM disk size (kbytes)
```

该项应被编入内核, 而不是作为模块。因为模块只能存放在文件系统中, 而挂载文件系统之前不可能加载模块, 这就成了 “先有鸡还是先有蛋” 的问题。

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 或 Platform selection, 处理器架构及相关选项, 根据开发的对象选择。图3.1是两种平台的选择。

4. Networking options, 系统需要支持的网络协议。大多数 Linux 系统都需要网络支持, 即使不连入 InterNet 网。

5. Device Drivers, 设备驱动, 包括针对硬盘、CDROM 等以块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。和硬件设备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。Windows 操作系统中, 对不同文件系统的操作是通过应用层软件实现的, 但是在 Linux 系统中, 文件系统是通过内核模块支持的。开发者可以选择支持哪些文件系统, 哪些编译进内核, 哪些作为模块后期加载。

配置完成后, 用下面的命令编译内核、模块及安装模块:

```
$ ARCH=arm CROSS_COMPILE=arm-linux- make -j8
  ...
  OBJCOPY arch/arm/boot/Image
  Building modules, stage 2.
  Kernel: arch/arm/boot/Image is ready
  GZIP arch/arm/boot/compressed/piggy.gzip
  MODPOST 1011 modules
  AS arch/arm/boot/compressed/piggy.gzip.o
  LD arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
$ ARCH=arm CROSS_COMPILE=arm-linux- make modules
$ ARCH=arm CROSS_COMPILE=arm-linux- make modules_install INSTALL_MOD_PATH=/
    ↪ home/student/kmodules
```

这里通过变量 CROSS_COMPILE 指明编译器前缀。make 的选项 -j 表示并行操作的数量, 在多核处理器上可以大大减少编译时间。并行值可以通过命令nproc 查看。

编译内核的同时也会编译设备树文件。设备树文件用一种结构化的语言描述系统的硬件设备, 代替早期对每个具体设备的直接编程。设备树的引入, 简化了设备驱动的编写, 也使内核映像文件具有更好的通用性。同样的内核映像文件, 只要有针对性地替换不同的设备树文件, 就可以支持不同的子架构。

注意生成的内核映像文件的位置, 需要将内核和设备树文件复制到 TFTP 服务目录, 供 BeagleBone Black 下载。对于 BeagleBone Black, 设备树文件是 am335x-boneblack.dtb (在 arch/arm/boot/dts/目录); 对于不同型号的树莓派, 请查找对应的设备树文件并将它复制到引导分区。

启动树莓派内核的参数由 Berryboot 文件 config.txt 指导, 复制时需要将内核映像按 config.txt 中指定的名称命名。

3.3 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, 通过串口工具 `minicom` 监控在目标机中下载并运行。下载之前需要正确设置的网络地址, 启动之前需要设置好启动参数:

```
U-Boot# set ipaddr ...
U-Boot# set serverip ...
U-Boot# tftp 0x82000000 zImage
U-Boot# tftp 0x88000000 am335x-boneblack.dtb
U-Boot# set bootargs ...
U-Boot# bootz 0x82000000 - 0x88000000
```

由于系统尚无文件系统支持, 只能从串口终端看到内核启动过程中打印的一些信息, 无法进入交互状态。

4

根文件系统

文件系统是管理存储数据的软件, 它使用文件和目录这样的逻辑概念代替物理存储设备使用的数据块的概念。在存储设备上, 不同的数据组织方式就形成了不同的文件系统。Linux 内核启动过程中需要挂载一个根文件系统, 根文件系统中含有一些基本的初始化命令, 通过这些命令为操作系统建立起完整的工作环境。

4.1 Linux 文件系统的类型

多文件系统支持是 Linux 的一个显著特征。从文件系统的形态看, Linux 支持三类文件系统: 基于磁盘类[1] 的本地文件系统、基于网络协议的网络文件系统和基于内存的伪文件系统。本实验重点研究第一类。

4.1.1 Ext 系列文件系统

第一个专为 Linux 设计的文件系统 ExtFS (**Ext**ended **F**ile **S**ystem, 扩展文件系统) 出现在 0.96c 版的 Linux 内核。它支持的最大文件是 2GB, 最大文件名长度为 255 个字符。Ext2FS 是 Linux 2.4 版本的标准文件系统, 它支持的最大分区为 2~32TB, 最大文件 16GB~2TB (取决于块大小的设置)。为增强文件系统的健壮性, 开发人员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。Ext4FS 向后兼容 Ext3FS 和 Ext2FS。

除了 Linux 原生的文件系统以外, Linux 通过虚拟文件系统层, 广泛支持各种类型的文件系统, 包括 Windows 系统使用的 VFAT、NTFS, Apple 系统使用的 APFS、HFS+, 以及在嵌入式系统上基于 FLASH 的 YAFFS2、JFFS2 等等。

[1]不限于磁性存储介质, 也包括 FLASH、光盘等。因历史原因, 对这些存储设备的统称 “磁盘” 一词沿用至今。

4.1.2 JFFS2 文件系统

JFFS (**J**ournaling **F**lash **F**ile **S**ystem, 日志型闪存文件系统) 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。Red Hat 的 David Woodhouse 在 JFFS 的基础上进行了改进, 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。相对于 Ext 系列文件系统而言, JFFS2 在嵌入式设备中更受欢迎, 它有以下优点:

- 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。

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要过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”。

4.1.3 YAFFS2

YAFFS (**Y**et **A**nother **F**lash **F**ile **S**ystem) 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。

4.1.4 网络文件系统

网络文件系统 NFS 是一个 RPC (**R**emote **P**rocess **C**all, 远程系统调用) 服务。它由 SUN 公司开发, 于 1984 年推出。网络文件系统设计的初衷是为了在不同的系统之间实现文件共享, 所以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。网络文件系统服务器端的具体配置可参考2.4.2节相关内容。

4.2 文件系统的制作

4.2.1 BusyBox 介绍

BusyBox 最初是为 Debian 开发的 Linux 系统工具包, 首创者是 Bruce Perens, 后来又有许多开发者对 BusyBox 贡献力量, 其中包括 Linux 之父 Linus Torvalds。BusyBox 是一个高度可配置的软件包, 最终编译成一个叫做 `busybox` 的独立执行程序。它包含上百个小应用程序、mini-vi 编辑器、系统启动程序 init, 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, 工作相当繁琐, 而且一些嵌入式系统也难以承受其规模。BusyBox 全部编译后, 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, 以控制 BusyBox 的规模。

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。[2]

本实验基于 BusyBox 构建根文件系统。

4.2.2 BusyBox 的编译

```
BusyBox 主页: https://busybox.net/
BusyBox 源码: https://busybox.net/downloads/busybox-1.29.2.tar.bz2
```

将下载的 BusyBox 软件包解压缩。进入解压目录, 仿照内核配置编译过程, 执行 `make menuconfig` 或 `make gconfig`。由于它与处理器架构无关, 不需要设置 `ARCH`, 只需要在编译时指定针对目标平台的编译器。

配置界面的 “Settings” 菜单用于设置 BusyBox 的编译方式。大多数选项都可以接受默认设置, 以下是一些可能改变的设置:

- “Build static binary (no shared libs)”, 建议选择共享库编译。静态编译固然简单, 它省去了 BusyBox 对共享库的依赖。但目标系统开发其他软件时, 最终还是要安装共享库。否则会占用过多的存储资源;
- “Don’t use /usr”, 根据 Linux 操作系统的 FHS (**f**ilesystem **H**ierarchy **S**tandard, 文件层次结构标准) 标准[3], /usr/ 是 Linux 操作系统的二级目录结构, 一些非系统性命令和库以这个目录为起点。这不是强制选项。当选中这项时, BusyBox 不使用这个目录, 所有命令都在 /bin/ 和 /sbin/ 目录下。建议选择这项功能, BusyBox 仅使用 /bin/ 和 /sbin/ 目录, 将二级目录结构 /usr/ 留作将来的系统扩展 (说明: 这不是标准做法, 也不是最佳策略, 只是为了扩展方便。)。
- “Destination path for ’make install’ ”, 自定义安装路径。`make install` 会将文件复制到这个目录里。缺省的设置是./_install, 开发者可根据自己的偏好进行设置。

[2]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。

[3] FHS 是 Linux 基金会维护的一套文件层次结构标准, 它定义了 Linux 操作系统目录结构层次及内容。以根目录为一级目录, /usr/ 为二级目录, /usr/local/ 为三级目录。

开发者还应根据需要对系统命令进行适当的取舍, 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:

```
$ CROSS_COMPILE=arm-linux- make -j8
...
Trying libraries:  crypt m resolv
 Library crypt is not needed, excluding it
 Library m is needed, can't exclude it (yet)
 Library resolv is needed, can't exclude it (yet)
 Library m is needed, can't exclude it (yet)
 Library resolv is needed, can't exclude it (yet)
Final link with: m resolv
```

注意最后的链接提示。如果是动态链接, 它表示需要共享库 libm 和 libresolv。使用代码分析工具 readelf 可以查看具体链接的库文件名:

```
$ arm-linux-readelf -d busybox
Dynamic section at offset 0xf7f08 contains 26 entries:
  标记        类型                         名称/值
 0x00000001 (NEEDED)                     共享库：[libm.so.6]
 0x00000001 (NEEDED)                     共享库：[libresolv.so.2]
 0x00000001 (NEEDED)                     共享库：[libc.so.6]
 0x0000000c (INIT)                       0x14178
...
```

下面制作文件系统时也需要把这些库复制到目标系统。

用下面的命令安装:

```
$ CROSS_COMPILE=arm-linux- make install
```

安装后, 在安装目录下可以看到 bin/、sbin/ 子目录。如果配置时未选中 “Don’t use /usr”, 还应该有 usr/ 子目录。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 bin/busybox。

4.2.3 配置文件系统

仅有 BusyBox, 系统是不能进入正常工作状态的。Linux 内核启动时, 会自动挂载根文件系统, 并执行根文件系统中的 /sbin/init[4]。虽然编译 BusyBox 已生成 init, 但它需要一些配置文件指导如何初始化 Linux 工作环境。下面的工作以 BusyBox 的安装目录 _install 为起点。

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

[4] /sbin/init 不是内核唯一认可的第一号进程, 它还会查找另外几个程序并执行, 详情可查看内核源码 init/main.c, 此处略过。

清单 4.1: init 初始化脚本/etc/inittab

```
1  # /etc/inittab
2
3  # 初始化表项格式为   id:runlevels:action:process
4
5  # id: 进程的I/O设备, 对于 tty, 默认前缀"/dev/", 缺省为stdin/out
6  # runlevels: BusyBox 不使用这个字段
7  # action: 可选项为 sysinit, respawn, askfirst, wait, once,
8  #         restart, ctrlaltdel, and shutdown.
9  # process: 完整的命令行
10
11 ::sysinit:/etc/init.d/rcS
12 # 在终端启动一个 "askfirst" shell
13 ::askfirst:-/bin/sh
14
15 # 在/dev/ttyX 启动若干 "askfirst" shell
16 tty2::askfirst:-/bin/sh
17 tty3::askfirst:-/bin/sh
18
19 # 在指定 tty 设备启动 /sbin/getty
20 tty4::respawn:/sbin/getty 38400 tty4
21 tty5::respawn:/sbin/getty 38400 tty5
22
23 # 通过串口连接的终端
24 #::respawn:/sbin/getty 57600 ttyS2
25
26 # 启动 TELNET 服务
27 #::once:/sbin/telnetd -l /bin/login
28
29 ::ctrlaltdel:/sbin/reboot
30 ::shutdown:/bin/umount -a -r
```

/etc/inittab 文件是 init 的初始化脚本, 以 # 开头的行是注释行。inittab 每一行由以冒号分隔的四个字段组成, 它们依次是 ID、运行级别、动作、命令。BusyBox 的 init 不支持运行级别概念, 因此这个字段是空的。

即使没有这个文件, BusyBox 的 init 进程仍然可以工作, 它相当于有一个缺省的 inittab (清单4.2)。

清单 4.2: init 缺省脚本

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/sbin/swapoff -a
::shutdown:/bin/umount -a -r
::restart:/sbin/init
tty2::askfirst:/bin/sh
tty3::askfirst:/bin/sh
tty4::askfirst:/bin/sh
```

rc 文件 (名称来自 UNIX **r**un **c**ommands) 安排操作系统的启动过程:

```
#!/bin/sh
hostname BeagleBone
mount -t proc proc /proc
mount -t sysfs sysfs /sys
cat /etc/motd
```

此文件要求可执行属性, 用命令 "`chmod +x rc`" 修改其属性。注意, 脚本程序第一行的 "`#`" 不是简单的注释, 它与 "`!`" 连在一起, 具有特定的含义, 必须按语法要求书写。

按 Linux 操作系统的习惯, 在 etc 目录下创建 init.d 目录, 并将 /etc/rc 向 /etc/init.d/rcS 做符号链接。inittab 文件的 sysinit 项从它开始运行。

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

motd 文件, 文件名来自 **m**essage **o**f **t**o**d**ay 的缩写。当用户登录系统时, 此文件作为日常提示信息, 其内容不影响系统正常工作, 由 /etc/rc 调用打印在终端上 (rc 文件最后一行)。

```
  Welcome to
  ================================
          ARM-LINUX WORLD
          BeagleBone BLACK
  ================================
```

- 创建 proc 和 sys 空目录, 供 PROCFS 和 SYSFS 文件系统使用。当完成了目录挂载之后 (rc 文件 3、4 两行), 这两个目录自动进入工作状态。

  创建 dev 目录, 用于挂载 DEVTMPFS 文件系统。当内核选择:

```
Device Drivers --->
    Generic Driver Options --->
```

```
    [*] Maintain a devtmpfs filesystem to mount at /dev
    [*]   Automount devtmpfs at /dev, after the kernel mounted ...
```

系统挂载根文件系统后会自动挂载 DEVTMPFS 到这个目录, 并在这个目录下生成设备文件。非自动维护情况时, 手工创建的设备文件也要在这个目录下。

创建 mnt 目录, 用于挂载其他文件系统。

- 建立 lib 目录, 将交叉编译器链接库路径下的共享库 ld-2.25.so 和 libc-2.25.so 库复制到 lib 目录[5], 并做相应的符号链接:

```
$ ln -s ld-2.25.so ld-linux-armhf.so.3
$ ln -s libc-2.25.so libc.so.6
```

此外, 还要根据 BusyBox 的链接提示, 把 BusyBox 需要的共享库复制过来并做好相应的符号链接:

```
$ ln -s libm-2.25.so libm.so.6
$ ln -s libresolv-2.25.so libresolv.so.2
```

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。从安装目录./_install 看下去, 应该至少有下面几个目录:

```
bin    dev    etc    lib    mnt    proc    sbin    sys    [usr]
```

它们是下面制作文件系统的基础。

按以上构造, Linux 启动流程是:

1. 内核调 /sbin/init 程序, 作为 1 号进程; 此时, 多任务环境已创建;
   (a) /sbin/init 以 /etc/inittab 为脚本, 该文件调 /etc/rc.d/rcS;
   (b) 执行 /bin/sh 命令, 创建终端环境, 与用户交互;
   (c) CTRL+ALT+DEL 系统复位 (此功能在串口终端控制下无法实现);
   (d) 关机时卸载所有文件系统。
2. rcS 链接自 /etc/rc, 故实际上是运行 /etc/rc 程序;
3. /etc/rc 给主机命名、挂载伪文件系统/proc、/sys、打印 motd。

这种启动流程源自 System V UNIX。了解了这个过程, 我们可以根据应用系统的需要, 在 rc 文件中添加其他的初始化工作。

[5]实际版本号取决于交叉编译器使用的 Glibc 版本。

4.2.4 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, 系统掉电后数据将丢失。

RAM Disk 要求内核支持:

```
Devices Drivers --->
    [*] Block devices --->
        <*>   RAM block device support
        (8192)  Default RAM disk size (kbytes)
```

应保证制作的 RAM Disk 的大小不超过这里的限制。

内核启动的初始化 RAM Disk 支持:

```
General setup  --->
    [*] Initial RAM filesystem and RAM disk (initramfs/initrd)...
    ()    Initramfs source file(s)
    [*]   Support initial ramdisk/ramfs compressed using gzip
    [ ]   Support initial ramdisk/ramfs compressed using bzip2
    ...
```

选择一种压缩格式 (这里选择 gzip), RAM Disk 文件映像可以按此格式压缩。

由于下面制作的 RAM Disk 是 Ext2/Ext3/Ext4 文件系统, 内核还需要选中 Ext4FS 的支持, 这个选项在:

```
File systems
    <*> The Extended 4 (ext4) filesystem
```

为了生成并修改 RAM Disk, 需要在主机上创建一个空文件并按文件系统将它格式化。格式化后的文件就是文件系统映像。虽然从形式上看, 它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。它可以像普通文件系统一样在主机上用 mount/umount 进行挂载和卸载, 逻辑上, 它和 U 盘、SD 卡甚至硬盘是等同的。挂载后可以进行正常的文件和目录操作, 卸载后, 如果不是只读文件系统或只读挂载方式, 原映像文件的内容便得到更新。

手工挂载文件系统映像需要超级用户权限。为避免频繁越权，管理员可事先完成下面的工作：

1. 创建专门用于挂载 *RAM Disk* 的目录 */mnt/ramdisk/*
2. 在 */etc/fstab* 文件中规定挂载文件系统、目录和权限等参数，即在*/etc/fstab* 中添加如下一行：

```
ramdisk_img   /mnt/ramdisk  auto  noauto,users,loop   0  0
```

它允许普通用户将文件 *ramdisk_img* 挂载到 */mnt/ramdisk/*。

这种做法的局限性是，一、*RAM Disk* 必须以 *ramdisk_img* 命名；二、普通用户挂载和卸载 *RAM Disk* 只能在当前目录 (即该文件所在目标)。普通用户对文件的个性化命名只能在卸载之后。

RAM Disk 上使用 *mknod* 命令创建设备文件。在 *PC* 上，*mknod* 需要超级用户权限。建议管理员用下面的命令把 *mknod* 开放给普通用户：

```
# chmod u+x /bin/mknod
```

```
$ dd if=/dev/zero of=ramdisk_img bs=1K count=8K
$ mke2fs -E root_owner ramdisk_img
$ mount ramdisk_img
$ (将制作的 BusyBox 安装目录、库以及脚本文件复制到ramdisk)
$ cd /mnt/ramdisk
$ mkdir dev                  # 创建设备文件：
$ cd dev
$ mknod console c 5 1
$ mknod null c 1 3
$ mknod zero c 1 5
```

mke2fs 的-E 选项设置文件系统格式化的用户权限。缺少这个选项，文件系统挂载后，普通用户没有写权限。

这里制作的 RAM Disk 大小是 8MB。busybox 本身在 1MB 左右。如果在复制文件过程中发现装不下了，极有可能是交叉编译器的 Glibc 库太大。把 RAM Disk 做得大一些不是一个好的策略，因为在大多数应用中，RAM Disk 只是一个中间产物。较好的做法是对这些库做一个 strip 处理，去除二进制文件中的调试信息。下面是对比 libc 处理前后的结果：

```
$ ls -l libc-2.25.so
-rwxr-xr-x 1 student student 13947972 1月 22 12:41 libc-2.25.so
$ file libc-2.25.so
libc-2.25.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV),
   ↪ dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux
```

```
    ↪ 3.2.0, not stripped
$ arm-linux-strip libc-2.25.so
$ file libc-2.25.so
libc-2.25.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV),
    ↪ dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux
    ↪ 3.2.0, stripped
$ ls -l libc-2.25.so
-rwxr-xr-x 1 student student 1238704 1月 22 12:44 libc-2.25.so
```

目录结构详情如下:

```
/mnt/ramdisk/
    |-- bin/                    (内含 BusyBox 普通用户程序)
    |   |-- [ -> busybox
    |   |-- [[ -> busybox
    |   |-- busybox
    |   |-- cat -> busybox
    |   `-- ...
    |-- dev/
    |   |-- console             (设备文件)
    |   |-- null
    |   `-- zero
    |-- etc/
    |   |-- init.d/
    |   |   `-- rcS -> ../rc
    |   |-- inittab             (/sbin/init 脚本文件)
    |   |-- motd                (每日信息提示)
    |   `-- rc
    |-- lib/                    (共享库目录)
    |   |-- ld-2.25.so
    |   |-- ld-linux-armhf.so.3 -> ld-2.25.so
    |   |-- libc-2.25.so
    |   |-- libc.so.6 -> libc-2.25.so
    |   |-- libm-2.25.so
    |   |-- libm.so.6 -> libm-2.25.so
    |   |-- libresolv-2.25.so
    |   `-- libresolv.so.2 -> libresolv-2.25.so
    |-- lost+found              (Ext2/3/4 文件系统固有目录)
    |-- mnt/                    (空目录, 挂载文件系统用)
    |-- proc/                   (空目录, 用于挂载 PROCFS)
    |-- sbin/                   (内含 BusyBox 系统用户程序)
    |   |-- ifconfig -> ../bin/busybox
    |   |-- init -> ../bin/busybox  (Linux 系统启动进程)
    |   |-- mknod -> ../bin/busybox
    |   `-- ...
    |-- sys/                    (空目录, 用于挂载 SYSFS)
    `-- usr/                    (二级目录结构)
        |-- bin/
        |    `-- ...
        `-- sbin/
             `-- ...
```

回到 ramdisk_img 文件所在目录, 卸载 RAM Disk:

```
$ umount /mnt/ramdisk
```

内核支持压缩方式的 RAM Disk。如果 RAM Disk 存放在 microSD 卡上, 可将其压缩以节省 FLASH 占用空间。根据内核设置的 RAM Disk 压缩格式, 用下面的方式压缩:

```
$ gzip ramdisk_img
```

压缩后的文件名是在原文件后面自动加上一个文件后缀 ".gz"。

开发过程中, 常需要对文件系统反复挂载、修改, 挂载之前需要先对文件系统映像解压。解压方式是:

```
$ 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。连同编译内核时生成的 zImage 和设备树文件 am335x-boneblack.dtb 一同载入 BeagleBone 的内存, 使用 bootz 引导启动:

```
U-Boot# setenv ramdisk root=/dev/ram rw initrd=0x88080000,8M
U-Boot# setenv bootargs console=ttyS0,115200 $ramdisk
U-Boot# tftp 0x88080000 ramdisk_img.gz
U-Boot# tftp 0x82000000 zImage
U-Boot# tftp 0x88000000 am335x-boneblack.dtb
U-Boot# bootz 0x82000000 - 0x88000000
```

启动参数中, "/dev/ram" 并不是一个真正存在的设备文件, 内核仅根据 "root" 中包含的字符信息决定引导方式。

4.2.5 制作初始化 RAMFS

将根文件系统和内核合并在一个文件中, 做成一个独立的映像文件, 在嵌入式系统中也是一个比较常见的做法。

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, 通过脚本程序完成系统的初始化工作, 挂载文件系统并用 `switch_root` 命令切换根文件系统。在 PC 上, 这是比较通行的做法。

目录结构准备完毕, 用 `cpio` 命令将它打包并压缩:

```
$ 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。

initrd.cpio.gz 就是初始化 RAMFS。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, 修改内核配置选项:

```
General setup  --->
    [*] Initial RAM filesystem and RAM disk (initramfs/initrd)...
    (initrd.cpio.gz)  Initramfs source file(s)
    [*]   Support initial ramdisk/ramfs compressed using gzip
```

重新编译内核, 将该文件编入内核文件 zImage, 并复制到 TFTP 服务器目录下。

在目标板中加载该内核, 启动。这种形式的内核, 不需要另外设置根文件系统参数。

如果要修改 initrd.cpio.gz, 同样也需要用 cpio 解包:

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

4.2.6 网络文件系统

嵌入式 Linux 作为网络文件系统的客户端, 内核配置中应保证下面的选项:

```
File Systems  --->
    [*] Network File Systems  --->
    <*>   NFS client support
    <*>     NFS client support for NFS version 2
    <*>     NFS client support for NFS version 3
    [*]       NFS client support for the NFSv3 ACL protocol
       ↪ extension
    <*>     NFS client support for NFS version 4
    ...
    [*]   Root file system on NFS
```

如果需要内核启动时将网络文件系统作为根文件系统挂载, 还需要选中 “Root file system on NFS”, 而该选项依赖于网络协议支持:

```
[*] Networking support  --->
        Networking options  --->
            [*]  IP: kernel level autoconfiguration
            [*]    IP: DHCP support
            [ ]    IP: BOOTP support
            [*]    IP: RARP support
```

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
```

这里的 “/dev/nfs” 也不需要真正存在于文件系统中, 内核仅根据文字信息决定启动方式。系统启动后, NFS 服务器目录下的 “Root_Dir” 目录就成为嵌入式 Linux 系统的根目录 “/”, 该目录结构应与制作的 RAM Disk 相同。

4.3 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, 用你的文件系统启动到正常工作状态;
- 在 PC 上编写一个 C/C++ 程序, 功能自定。使用交叉编译器编译成可执行文件, 将它复制到网络文件系统的共享目录中, 尝试在目标系统上运行。

思考题: 本章介绍了三种根文件系统的构造方法。试对比它们的优劣。

5

建立使用环境

以 RAM Disk 作为根文件系统的 Linux 环境, 可用资源会受到很大限制。首先, RAM Disk 占用内存, 不可能做得很大; 其次, RAM Disk 毕竟是 RAM, 无法长久保留数据。此外, Linux 是一个多用户操作系统, 除非是特定应用系统, 否则, 还需要为它建立多用户环境。

5.1 文件系统扩展

作为根文件系统的 RAM Disk 通常只有几兆字节, 不足以支持后面的实验。
下面是利用网络文件系统扩展存储空间的方法。

1. 启动系统, 在终端配置网络, 挂载网络文件系统。如果 BusyBox 未使用 /usr/ 目录, 则需要先创建这个目录; 否则, 应先将 /usr/bin/ 和 /usr/sbin/ 目录下的程序复制到 /bin/ 和 /sbin/, 因为下面的操作将会占用 /usr/ 目录:

   ```
   # ifconfig eth0 192.168.2.123
   # mount 192.168.2.12:/srv/nfs4/extdir /usr -o nolock,proto=tcp
   ```

   此处假设嵌入式系统的 IP 地址是 192.168.2.123, PC 的 IP 地址是 192.168.2.12, PC 开放的 NFS 服务器目录是 /srv/nfs4, 其子目录 extdir/ 作为嵌入式目标系统的扩展空间。
2. 在 PC 上构造 extdir, 其结构应具有下面的形式:

   ```
   /srv/nfs4/extdir
       |-- bin/
       |-- lib/
       |   |-- libcrypt-2.25.so
       |   |-- libcrypt.so.1 -> libcrypt-2.25.so
       |   |-- libdl-2.25.so
       |   |-- libdl.so.2 -> libdl-2.25.so
       |   |-- libgcc_s.so -> libgcc_s.so.1
       |   |-- libgcc_s.so.1
       |   |-- libpthread-2.25.so
       |   |-- libpthread.so.0 -> libpthread-2.25.so
       |   |-- libresolv-2.25.so
       |   |-- libresolv.so.2 -> libresolv-2.25.so
   ```

```
|   |-- librt-2.25.so
|   |-- librt.so.1 -> librt-2.25.so
|   |-- libstdc++.so.6 -> libstdc++.so.6.0.27
|   |-- libstdc++.so.6.0.27
|   `-- ...
`-- sbin/
```

其中的共享库来自交叉编译工具链, 与先前制作 RAM Disk 时的共享库为同一来源。不是所有的库都会用到, 但如果运行程序时发现动态链接库缺失, 请注意及时补充缺失的库。

3. 创建 /etc/hosts 文件。hosts 负责建立 IP 地址与域名的对应关系, 是网络连接的重要文件, 通常可以按如下方式编写:

```
127.0.0.1       localhost
127.0.1.1       BeagleBone

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

其中主机 localhost 的 IP 地址对应回环设备 lo, 通过下面的命令创建:

```
# ifconfig lo 127.0.0.1
```

lo 设备在使用网络连接 SSH X11 转发时是必须的。

将配置网络、挂载网络文件系统的操作纳入系统初始化过程, 写入 /etc/rc 文件中。如果本身就是以网络文件系统作为根文件系统启动, 不需要这个步骤, 但对网络文件系统的目录结构要求不变。

5.2 建立用户环境

5.2.1 用户和组

Linux 使用 /etc/passwd 和 /etc/group 两个文件管理用户和分组。passwd 文件的每一行描述了一个用户的基本信息, 以 ":" 作为分隔, 按如下方式组织:

```
用户名:密码:用户ID:组ID:用户信息:主目录:用户环境
```

密码部分保存 `passwd` 命令生成的加密文本。用户 ID 和组 ID 是系统识别用户的重要特征, 1000 以内的 ID 留给系统使用, 其中超级用户 (习惯上以 root 为用户名) 的用户 ID 和组 ID 都是 0; "用户信息" 栏不重要, 只是一段文字信息。用户登录后的工作目录以这里给出的主目录为起点, "用户环境" 是用户工作的解释器, 通常是 /bin/sh 或 /bin/bash。

group 文件每一行记录一个组信息。相比于 passwd 文件来说要简单得多:

```
组名:密码:组ID:组员列表(用逗号分隔)
```

在 passwd 中的每一个用户, group 中都应该有一个对应同名的组名, 称为基础组, 其组 ID 应与 passwd 中对应用户的组 ID 相同。组密码通常是空的, 安全性通过操作系统的分组策略和用户自身的密码保证。

5.2.2 添加用户和组

开发者可以直接使用文本编辑器, 通过编辑 passwd 和 group 这两个文件创建用户。BusyBox 也提供了 adduser/deluser 和 addgroup/delgroup 两组命令管理用户和分组。由于 adduser/addgroup 不自动创建新文件, 在使用这组命令管理用户和分组时需要先手工创建空文件:

```
# touch /dev/passwd /dev/group
# adduser root -u 0
...
```

手工编辑 passwd 时, 密码栏应留空, 待用户登录后使用 passwd 命令修改密码, 否则用户无法登录。

5.2.3 远程登录

Linux 网络功能强大, 使用也很灵活, 有多种远程工作手段。早期曾使用 TELNET 网络服务, 目前较常用的是 SSH (**S**ecure **SH**ell), 其安全性和功能都远超 TELNET。但 SSH 需要另外移植软件。这里仅介绍 TELNET 的使用。

BusyBox 中已经包含了 TELNET 的服务器和客户端, 注意在配置 BusyBox 时选中它们:

```
Networking Utilities  --->
    [*] telnet
    [*]   Pass TERM type to remote host
    [*]   Pass USER type to remote host
    [*]   Enable window size autodetection
    [*] telnetd
    [*]   Support standalone telnetd (not inetd only)
    [*]     Support -w SEC option (inetd wait mode)
```

在 /etc/inittab 中增加一项, 系统启动时自动启动 TELNET 服务:

```
# /etc/inittab
...
::once:/sbin/telnetd -l /bin/login
```

启动 TELNET 服务也可以安排在 /etc/rc 中。

上面看出, 启动 TELNET 服务需要运行命令 `login`, 因此在配置 BusyBox 时也要把这个命令加进去。

启动 TELNET 还需要内核 devpts (**dev**ice **p**seudo **t**erminal **s**ystem) 支持, 即在配置内核时选中下面的几项:

```
Device Drivers  --->
    Character devices  --->
        [*] Enable TTY
            [*]  Virtual terminal
                [*]   Enable character translations in console
                [*]   Support for console on virtual terminal
            [*]  Unix98 PTY support
```

仿照 PROCFS 和 SYSFS 的挂载方式挂载 devpts。这个文件系统要求挂在 /dev/pts 目录下, 挂载前需要先创建这个目录:

```
# mkdir -p /dev/pts
# mount -t devpts devpts /dev/pts
```

为了让系统启动时自动完成这些工作, 可将上述两条命令写入启动脚本 /etc/rc。

5.3 实验内容

为目标系统创建用户, 设计系统启动过程, 自动完成 TELNET 服务和挂载网络文件系统的工作。在 PC 端通过 telnet 登录到目标系统。

6

图形用户界面

图形用户界面 GUI (**G**raphical **U**ser **I**nterface, 又称图形用户接口) 是一种人—机接口形式。现代计算机系统普遍采用图形界面。相比于字符界面, 图形界面操作更加简单直观。本实验是图形用户界面的基础, 帮助开发者了解图形显示的方式。

6.1 帧缓冲设备

6.1.1 色彩表示方法

计算机反映自然界的颜色是通过红绿蓝 RGB (**R**ed **G**reen **B**lue) 三色值来表示的。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。保存屏幕上所有点的一段内存称为显示缓存区 (以下简称 “显存”), 每一个像素的颜色都与显存某个固定地址存储的内容相对应。显存可以直接存放像素点的 RGB 值, 也可能存放的是调色板的索引, 通过索引值在调色板中查表, 可以间接获得像素点的 RGB 值。

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 位来表示每个点的状态, 一个字节可以表示 8 个点。单色并不只限于于黑与白两种颜色, 而是说只能以两种颜色来表示, 通常会选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。这些色位代表整个屏幕所有像素的颜色取值范围, 色位越大, 需要提供的缓存就越大, 当然显示的色彩也就越丰富。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。这些因素会影响显存空间的大小, 因为帧缓冲区空间的大小是以屏幕的大小和显示模式来决定的。如: 采用 4 位色/像素的显示模式, 显示屏上每个点能够出现的颜色种类最多只能有 16 种 (2^4), 一个 640×480 分辨率大小的显示器, 显存为 160KB。而同样大小 32 位色的显示器需要 1.2MB 显存。

彩色方式下, 不同规格的显示会给 R、G、B 分配不同的位数。目前最高规格的是 24 位或 32 位, R、G、B 各 8 位 (称为真彩色)。如果是 32 位, 多出的 8 位分配给 Alpha 通道。因此我们会看到用 RGB 或 RGBA 表示的显示格式。Alpha 通道通常可以用作透明度, 透明度可以通过硬件实现, 也可以通过软件实现。24 位的缺点是, 存储像素单元的地址不是 4 字节对齐的, 这在 32 位系统中降低了读写的效率。

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。例如, 16 位点像素模式下, 显存每两个字节对应显示器一个点的色彩, 其色彩格式可能是 RGB565 (即红绿蓝分别有 5 位、6 位、5 位, 见图 6.1), 或 RGBA5551。

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

图 6.1: RGB 565 的一种色位关系

6.1.2 Frame Buffer 简介

显示屏的整个区域与系统内一段存储空间相对应, 系统通过改变该存储空间的内容达到改变显示信息的目的。Linux 系统通过帧缓冲 (Frame Buffer) 技术实现对显示器的控制。帧缓冲技术将显示设备的物理内存映射到用户空间, 在用户空间通过存储器读写而直接反映到显示设备上。由此, 解决显示屏的显示问题, 首先就需要解决显示屏上的每一像素与缓冲区之间的映射关系问题。

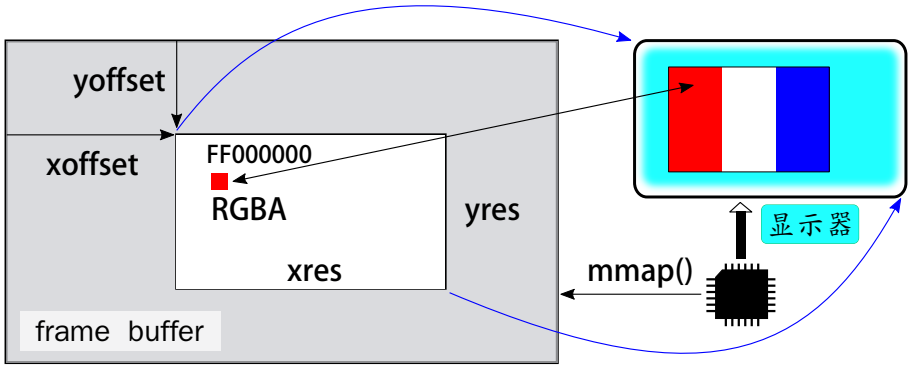


图 6.2: Frame Buffer 与显示器

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 里的数据则是按线性地址递增的顺序存放。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; 屏幕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。图6.2是一个比较完整的显存—显示器对应关系。实际上, 多数 Frame Buffer 和显示器区间的大小是一致的。

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。目前嵌入式系统普遍使用液晶显示器 (**L**iquid **C**rystal **D**isplay, LCD), 实现中间件控制功能的自然就叫做 LCDC, 它可能集成在处理器内部, 也可能是独立的器件。

另一个影响显存空间大小的因素是显示屏的单屏/双屏模式。

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。它与两个独立的显示屏拼成一个显示屏的方式不同。单看其中一部分, 它的显示方式是与单屏方式是一致的, 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, 且这两块 Frame Buffer 无需前后衔接, 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。这种结构的优点是可以将显示器分成前后台处理。因为在前台绘制一幅图耗时比较长, 程序写得不好、或者处理器性能不够时, 可以明显看出闪烁和动画效果, 而将后台切换到前台则可以瞬间完成。

6.1.3 Frame Buffer 内核支持

Linux 内核自 2.1.109 引入了 Frame Buffer 支持。大多数 Cortex-A 处理器都可以通过配置内核选项支持这项功能。在 BeagleBoard 上:

```
Device Drivers --->
   Graphics support --->
     <*> DRM Support for TI LCDC Display Controller
     ...
        Frame buffer Devices --->
          <*> DA8xx/OMAP-L1xx/AM335x Framebuffer support
```

树莓派上:

```
Device Drivers --->
   Graphics support --->
        Frame buffer Devices --->
          {*} Support for frame buffer devices --->
              <*> BCM2708 framebuffer support
```

当 Frame Buffer 驱动工作后, 可以看到设备文件 /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。存在有多个帧缓冲设备时, 次设备号依次递增。

6.2 Frame Buffer 编程

了解帧缓冲设备的参数可以通过系统功能调用ioctl() 的 FBIOGET_FSCREENINFO、FBIOGET_VSCREENINFO 命令, 前者获取固定参数 (F:Fixed), 后者获取可变参数 (V:Variable), 如:

```
1 #include <linux/fb.h>
2 ...
3     struct fb_fix_screeninfo finfo;
```

```
4     struct fb_var_screeninfo vinfo;
5     fd = open("/dev/fb", O_RDWR);
6     ioctl(fd, FBIOGET_VSCREENINFO, &vinfo);
7     ioctl(fd, FBIOGET_FSCREENINFO, &finfo);
```

可以获得显示器色位、分辨率、颜色结构等信息。fb_fix_screeninfo 和fb_var_screeninfo 结构如下:

```
1  struct fb_fix_screeninfo {
2      char id[16];                /* ID 字串 */
3      unsigned long smem_start;   /* frame buffer 物理首地址 */
4      __u32 smem_len;             /* frame buffer 长度 */
5      __u32 type;                 /* frame buffer 类型 */
6      __u32 type_aux;             /* 隔行 */
7      __u32 visual;               /* 可视模式:黑白/眞彩/伪彩... */
8      __u16 xpanstep;             /* zero if no hardware panning */
9      __u16 ypanstep;             /* zero if no hardware panning */
10     __u16 ywrapstep;            /* zero if no hardware ywrap   */
11     __u32 line_length;          /* 每行字节数 */
12     unsigned long mmio_start;   /* I/O映射内存物理首地址 */
13     __u32 mmio_len;             /* I/O映射内存长度 */
14     __u32 accel;                /* 加速器支持 */
15     __u16 capabilities;         /* 功能项 */
16     __u16 reserved[2];          /* 保留 */
17 };
18
19 struct fb_bitfield {           /* 位域结构 */
20     __u32 offset;               /* 起始位 */
21     __u32 length;               /* 位数 */
22     __u32 msb_right;            /* MSB 右对齐 */
23 };
24
25 struct fb_var_screeninfo {
26     __u32 xres;                 /* 可视分辨率 */
27     __u32 yres;
28     __u32 xres_virtual;         /* 虚拟分辨率 */
29     __u32 yres_virtual;
30     __u32 xoffset;              /* 可视域相对于虚拟域的偏移 */
31     __u32 yoffset;
```

```
32
33      __u32 bits_per_pixel;       /* 每像素点位数 */
34      __u32 grayscale;            /* 灰度级(0: 彩色, 1: 灰度) */
35      struct fb_bitfield red;     /* 位域 RED */
36      struct fb_bitfield green;   /* 位域 GREEN */
37      struct fb_bitfield blue;    /* 位域 BLUE */
38      struct fb_bitfield transp;  /* 位域 ALPHA */
39
40      __u32 nonstd;               /* != 0 非标准像素格式 */
41
42      __u32 activate;             /* 激活方式 */
43
44      __u32 height;               /* 图像高度(mm) */
45      __u32 width;                /* 图像宽度(mm) */
46
47      __u32 accel_flags;          /* 加速标志(已弃用) */
48
49      /* 时钟. 除pixclock 外, 均以pixclock 数为单位 */
50      __u32 pixclock;             /* 像素时钟 (pico seconds) */
51      __u32 left_margin;          /* 左边距 */
52      __u32 right_margin;         /* 右边距 */
53      __u32 upper_margin;         /* 上边距 */
54      __u32 lower_margin;         /* 下边距 */
55      __u32 hsync_len;            /* 水平同步长度 */
56      __u32 vsync_len;            /* 垂直同步长度 */
57      __u32 sync;                 /* see FB_SYNC_* */
58      __u32 vmode;                /* see FB_VMODE_* */
59      __u32 rotate;               /* 逆时针转角 */
60      __u32 colorspace;           /* 4字节编码格式的色彩空间 */
61      __u32 reserved[4];          /* 保留 */
62  };
```

你也可以用 BusyBox 提供的 `fbset` 命令了解这个设备的特性:

```
# fbset
mode "1024x768"
    geometry 1024 768 1024 768 16
    timings 0 0 0 0 0 0 0
    accel true
```

```
    rgba 5/11,6/5,5/0,0/0
endmode
```

以上表示显示分辨率是 1024×768, 虚拟分辨率也是 1024×768, 色位 16 位, “rgba” 表示这些色彩分量在一个字中所占据的位宽和起始位, 起始位从最低位算起。

获取 Frame Buffer 缓冲区首地址的系统功能调用是:

```
#include <sys/mman.h>

unsigned char *fbp = 0;
fd = open("/dev/fb", O_RDWR);
fbp = (unsigned char *)mmap(NULL,                    /* 通常是空 */
                            buffersize,              /* 映射长度 */
                            PROT_READ|PROT_WRITE,    /* 保护方式 */
                            MAP_SHARED,              /* 共享方式 */
                            fd,                      /* 文件描述符 */
                            0);                      /* 起始位置 */
```

buffersize 是mmap() 需要映射的显存字节数, 它可以根据显示器信息进行计算:

$$buffersize = \frac{width \times height \times bits\_per\_pixel}{8}$$

通过 mmap() 函数获得缓冲区首地址, 从该首地址开始、以buffersize 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:

```
1 /* draw a red point at (x, y) in RGB888 */
2
3     offset = y * finfo.line_length + x * vinfo.bits_per_pixel / 8;
4     *(unsigned char *)(fbp + offset + 0) = 0;
5     *(unsigned char *)(fbp + offset + 1) = 0;
6     *(unsigned char *)(fbp + offset + 2) = 255;
7     ......
```

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。由于从 8 位缩减到 5 位或者 6 位, 必然损失精度, 一般是抛弃低位。

```
1 /* draw (Red, Green, Blue) at (x, y) in RGB565 */
2 int dot(int x, int y,
3         unsigned char Red,
4         unsigned char Green,
5         unsigned char Blue)
6 {
```

```
7       ...
8       offset = y * finfo.line_length + x * vinfo.bits_per_pixel / 8;
9       color = ((Red << 8)   & 0xF800) |
10              ((Green << 3) & 0x07E0) |
11              ((Blue >> 3)  & 0x001F);
12      *(unsigned char *)(fbp + offset + 0) = color & 0xFF;
13      *(unsigned char *)(fbp + offset + 1) = (color >> 8) & 0xFF;
14      ......
15  }
```

不同位端 (endian) 的处理器、以及不同的 Frame Buffer 结构, 移位及高低字节顺序可能有所不同, 原则上应根据 `fb_bitfield` 结构信息填充色彩值。

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

使用完毕应通过 `munmap()` 释放显示缓冲区:

```
int munmap(void *addr, size_t length);
```

即使不调用`munmap()`, 进程结束时也会自动释放显示缓冲区, 但关闭文件不会导致缓冲区释放。

6.3 实验内容

6.3.1 实验方法

由于像 BeagleBoard 这类单板计算机本身不是针对桌面应用设计的, 所以都不带显示器, 但留有一些显示接口。并且 Linux 操作系统也有灵活的功能, 因此图形界面的程序设计实验可以采用不同的形式。

外接 HDMI 显示器

BeagleBoard 和树莓派都有 HDMI 输出接口。如果不在意成本, 通过 HDMI 连接显示器是最简单可靠的方法。

其他接口的显示设备

热门的板卡总不会缺少企业为它生产配件。图6.3是分别用于 Beagle Bone Black 扩展显示器和树莓派 DSI 接口的液晶屏显示器。这类显示接口不像 HDMI, 没有统一的标准, 不能互换。建议配备带有触摸屏的显示器, 便于触摸屏实验。

虚拟显示设备

目标系统开启 VNC 服务器, PC 远程登录 VNC, 以 PC 显示器的一部分作为目标系统的显示器。这种方法, 需要先移植 VNC 服务软件。

(a) BeagleBone Cape 触摸屏 (b) 树莓派 DSI 接口触摸屏

图 6.3: 两种 LCD 显示器

```
x11vnc 主页: http://www.karlrunge.com/x11vnc
x11vnc 源码: http://x11vnc.sourceforge.net/dev/x11vnc-0.9.14-dev.tar.gz
```

移植过程如下:

1. 下载 x11vnc 源码
2. 按如下方法配置、编译:

```
在 x11vnc 解压目录下执行:
$ ./configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-fbdev \
    --without-ssl \
    --without-crypt \
    --without-pthread \
    --without-jpeg \
    --without-zlib \
    --without-x
$ make -j8
```

以上省去了所有可能的依赖库, 仅通过选项`--with-fbdev` 保证支持 Frame Buffer。性能有所降低, 功能也不完整, 目的是为了简化移植过程。

编译完成后, 在 x11vnc/ 目录下生成一个可执行程序 x11vnc。把这个程序复制到目标系统, 在目标系统上启动 VNC 服务:

```
# x11vnc -rawfb /dev/fb0 -forever -shared >/dev/null 2>/dev/null &
```

将输出重定向到文件 /dev/null, 是为了避免程序运行时将我们不关心的信息打印在终端上, 干扰终端的正常使用。

PC 端使用 vncviewer, 将目标系统的显示内容展现在本地窗口:

```
$ vncviewer 192.168.2.123
```

由于是通过网络远程连接, 没有图像压缩功能、没有视频流处理能力, 帧率大概在 10fps 左右, 不能很好地显示视频; 又因为 x11vnc 没有编入输入接口的支持, 无法通过主机与目标系统进行交互操作, 只能基本满足本实验对于图像的显示的要求。

若采用虚拟显示方案, 建议由教师完成 *x11vnc* 的移植工作, 提供给实验学生使用。

6.3.2 实现基本画图功能

1. 基于 Frame Buffer 编写画点、画线的 API 函数, 以此为基础实现画任意曲线的画图功能;
2. 编写画圆/椭圆的函数;
3. * 参考 Bresenham 画图算法[1]改进你的程序, 并通过实验对比改进前后的效果。

注意代码功能模块化, 主程序和子程序分开, 使用 Makefile 管理编译过程。

出于降低能耗的目的, 在一定时间内未检测到动作, 内核会将显示器调入休眠状态 (黑屏)。这里所说的 "一定时间", 在内核源码文件 *drivers/tty/vt/vt.c* 中:

```
...
static int vesa_blank_mode;
static int vesa_off_interval;
static int blankinterval = 10*60;
...
```

将blankinterval 设为 *0*, 即取消黑屏。

如果不打算修改源码, 当显示黑屏时, 可向文件*/sys/class/graphics/fb0/blank* 中写入任意信息, 激活显示器:

```
# echo "asdf" >/sys/class/graphics/fb0/blank
```

6.3.3 软件结构设计

软件开发应考虑模块化和层次化, 这是可持续发展的基础。以下软件布局方式可供参考:

```
project/
|-- dot.c           # 画点函数 dot()
|-- line.c          # 画线函数 line(), 调用 dot()
|-- polyline.c      # 画折线或曲线, 调用 line()
```

[1]参考文献:

Bresenham, J. E., Algorithm for computer control of a digital plotter, in IBM Systems Journal, vol. 4, no. 1, pp. 25-30, 1965, doi: 10.1147/sj.41.0025.

Alois Zingl, A Rasterizing Algorithm for Drawing Curves, http://members.chello.at/~easyfilter/Bresenham.pdf (2012)

```
|-- circle.c        # 画圆/椭圆函数 circle/ellipse(), 调用dot()
|-- ...             # 其他画图功能
|-- draw.h          # 头文件, 函数原型声明
|-- main.c          # 主程序
`-- Makefile
```

层次化可以化繁为简, 模块化有利于实现功能共享。操作系统提供了大量的软件库, 就是典型的共享方式。在这个实验中, 我们可以将调用设备驱动实现的功能独立出来, 构成一个函数库, 为用户程序屏蔽底层硬件信息, 提供一些简单的画图调用。

Linux 操作系统有两种库的形式:

1. 静态库: 通过归档管理器 ar 将若干模块编译生成的 ".o" 文件合并而生成的 ".a" 文件。编译静态库的方法:

   ```
   $ arm-linux-gcc -c dot.c
   $ arm-linux-gcc -c line.c
   $ ...
   $ ar cq libdraw.a dot.o line.o ...
   ```

   独立编译后的 ".o" 文件也可以看作是函数库, 但它不能像库那样用链接器的 `-l` 选项与主程序链接。
2. 共享库, 又叫动态库, 通过链接器的 `-shared` 选项生成一个没有主函数 `main()` 的库文件。编译动态库的方法:

   ```
   $ arm-linux-gcc -shared -fPIC -o libdraw.so -Wl,soname,libdraw.so.0 dot.o
       ↪ line.o ...
   ```

库文件名前缀是固定的 "lib" 三个字母, 后缀是 ".a" 或 ".so"。链接选项 `-l` 只需要指定库名, 不需要完整的文件名。未经选项 `-static` 明示, GCC 默认链接动态库; 如果头文件和库文件不在 GCC 的默认搜索路径中, 还需要用选项 `-I`、`-L` 指明。例如

```
$ arm-linux-gcc -o main main.c -I. -L. -ldraw
```

运行动态链接的程序, 需要将动态库 ".so" 文件复制到目标系统 /usr/lib/目录, 或者通过环境变量`LD_LIBRARY_PATH` 指明动态库所在路径。如果编译动态库时有链接选项 soname, 则库文件以 soname 命名为准。

soname 并不自动生成库文件, 需要手工制作链接 (参考4.2.3节在制作根文件系统时对 Glibc 的操作)。

7

音频接口

音频信号是多媒体信息的重要组成部分。本实验学习在 Linux 操作系统上音频采集和回放的方法。

7.1 接口介绍

计算机系统中, 声卡负责将音频模拟信号转换成数字信号, 或将数字信号还原成音频模拟信号。历史上, Linux 内核支持两套音频设备驱动架构: OSS (**O**pen **S**ound **S**ystem, 开放声音系统) 和 ALSA (**A**dvanced **L**inux **S**ound **A**rchitecture, 高级 Linux 声音架构)。目前主要采用 ALSA, OSS 只作为一个向后兼容的选项而保留至今。这两种架构提供不同的接口, 编程差别较大, 表7.1是二者使用的设备文件, 内核为 OSS 分配的主设备号是 14, 为 ALSA 分配的主设备号是 116。

表 7.1: Linux 声卡驱动使用的设备文件

功能	ALSA	OSS
采样	/dev/snd/pcmC0D0c	/dev/dsp
输出	/dev/snd/pcmC0D0p	/dev/dsp
混音	/dev/snd/mixerC0D0	/dev/mixer
音序器		/dev/midi
状态		/dev/sndstat
控制	/dev/snd/controlC0	/dev/dsp

BeagleBone Black 没有单独的音频输入输出接口。处理器 AM335X 的 McASP0 (**M**ulti**c**hannel **A**udio **S**erial **P**ort, 多通道音频串口) 将数字音频信号送往 HDMI 接口 TDA19988, 由 TDA19988 负责音频信号输出。树莓派的音频设备由 Broadcom BCM2835 I2S 模块支持。配置内核时, 需要选中相关功能。当选择 ALSA 兼容 OSS 时, 可以使用 /dev/dsp 设备文件进行音频接口编程。

```
Device Drivers  --->
```

```
Graphics support  --->
    I2C encoder or helper chips  --->
        <*> NXP Semiconductors TDA998X HDMI encoder
...
<*> Sound card support  --->
    <*>  Advanced Linux Sound Architecture  --->
        [*]  Enable OSS Emulation
        <M>    OSS Mixer API
        <M>    OSS PCM (digital audio) API
        <*>  ALSA for SoC audio support  --->
            <*>  SoC Audio for Texas Instruments chips using
               ↪ eDMA
            -*-  Multichannel Audio Serial Port (McASP) support
            <*>  SoC Audio for the AM33XX chip based boards
```

7.2 OSS 软件设计

面向 OSS 的设备文件主要有两个: /dev/dsp 和 /dev/mixer。/dev/dsp 用于模拟信号的输入 (A/D 转换) 和输出 (D/A 转换), 以及 A/D、D/A 转换工作方式的设置, 包括设置采样/输出频率、通道数、数据格式等等。嵌入式系统中, 音频信号输入/输出一般都使用 $\Delta-\Sigma$ 的 A/D、D/A 转换器。模拟信号输入/输出通过读/写系统调用实现, 工作方式通过 ioctl() 控制。/dev/mixer 用于混音器设置, 但 OSS 也支持直接使用 /dev/dsp 设置混音器。OSS 重要的头文件是 linux/soundcard.h, 其中定义的常用设置命令有:

- SNDCTL_DSP_RESET, 设备复位。
- SNDCTL_DSP_SPEED, 采样/输出率, 如 8000Hz、44100Hz、48000Hz 等。
- SNDCTL_DSP_CHANNELS, 通道数, 单通道为 1, 立体声为 2。如果是双通道, 左右通道数据交替存放。
- SOUND_PCM_READ_CHANNELS, SOUND_PCM_WRITE_CHANNELS, PCM 输入、输出通道数, 通常二者是一致的。
- SNDCTL_DSP_SETFRAGMENT, 缓冲数据块大小。
- SNDCTL_DSP_SAMPLESIZE, 采样值的数据位大小。
- SNDCTL_DSP_SETFMT、SNDCT_DSP_GETFMTS, 设置/获取数据位格式。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。

上述命令通过系统功能调用 ioctl() 发给声卡设备。例如, 下面的系统功能调用将设备 /dev/dsp 的采样率设置为 8000Hz:

```
    int fd = open("/dev/dsp", O_RDWR);
    int fs = 8000;
    ret = ioctl(fd, SNDCTL_DSP_SPEED, &fs);
    if (ret < 0) {   /* ioctl() 调用出错 */
        ...
```

$\Delta - \Sigma$ 的 A/D、D/A 转换器通过晶振的有限个分频得到采样率, 不能直接实现任意频率的采样/输出。完整的做法还应根据系统功能调用的返回值判断是否设置成功。

由于硬件的局限性, 树莓派和 BeagleBone Black 仅具备通过 HDMI 的音频输出能力, 不带有音频信号的 A/D 转换功能, 必须另接声卡才能采集音频模拟信号。

混音器设备文件 /dev/mixer 用于设置音源 (例如模拟信号选择话筒输入还是线输入), 控制音源和输出的音量 (输出高低音成分、左右通道强度) 等等。混音器的常用命令有:

- 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。

清单 7.1: OSS 输出/输入示例 oss.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <sys/ioctl.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <unistd.h>
8  #include <math.h>
9  #include <sys/soundcard.h>
10
11 #define BUF_SIZE    8192
12 #define DEVICE_NAME "/dev/dsp"
13
14 int main (int argc, char *argv[])
15 {
16     int audio_fd;
17     short buffer[BUF_SIZE];
18     int speed;
```

```
19      int format;
20      int stereo;
21      int channels;
22      int vol;
23
24      if ((audio_fd = open(DEVICE_NAME, O_RDWR)) < 0) {
25          perror(DEVICE_NAME);    /* 打开设备文件出错 */
26          return EXIT_FAILURE;
27      }
28
29      /* 设置采样率 */
30      speed = 11025;
31      if (ioctl(audio_fd, SNDCTL_DSP_SPEED, &speed) < 0) {
32          perror("SNDCTL_DSP_SPEED");
33          printf("Supported sampling rate %d.\n", speed);
34      }
35
36      if (speed != 11025) {
37          /* 不支持的采样率，应根据返回值重新设置 */
38      }
39
40      /* 设置数据格式: 16bit 小端 */
41      format = AFMT_S16_LE;
42      if (ioctl(audio_fd, SNDCTL_DSP_SETFMT, &format) < 0) {
43          perror("SNDCTL_DSP_SETFMT");
44      }
45
46      if (format != AFMT_S16_LE) {
47          /* 数据格式不支持时，应重新选择数据格式设置 */
48      }
49
50      /* 设置立体声模式 */
51      stereo = 1;
52      if (ioctl(audio_fd, SNDCTL_DSP_STEREO, &stereo) < 0) {
53          /*  Fatal error */
54          perror("SNDCTL_DSP_STEREO");
55      }
56
```

```
57      if (stereo != 1) {
58          /* 不支持立体声，应选择单通道方式 */
59          channels = 1;
60          ioctl(audio_fd, SNDCTL_DSP_CHANNELS, &channels);
61      } else {
62          channels = 2;
63      }
64
65      /* 设置音量
66       * 音量原则上应通过混音器 /dev/mixer 设置，但/dev/dsp也支持
67       * 音量值 vol 的高低8位对应左右两个通道
68       * 每个通道的最大值100，最小值 0
69       * MIXER_WRITE 设置输出通道
70       * MIXER_READ 设置输入通道
71       */
72      vol = (100 << 8)|100;
73      if (ioctl(audio_fd, MIXER_WRITE(SOUND_MIXER_PCM), &vol) < 0) {
74          perror("MIXER_WRITE"); /* 设置不成功的处理过程 */
75      }
76
77      vol = (100 << 8)|100;
78      if (ioctl(audio_fd, MIXER_READ(SOUND_MIXER_MIC), &vol) < 0) {
79          perror("MIXER_READ"); /* 设置不成功的处理过程 */
80      }
81
82      /* 在缓冲区构造一个单通道、500Hz的正弦波 */
83      for (int i = 0; i < BUF_SIZE; i += channels) {
84          buffer[i] = 10000 * sin(2*M_PI*i/speed*500);
85          if (channels > 1)
86              buffer[i+1] = 0;
87      }
88
89      write(audio_fd, buffer, BUF_SIZE*2);
90      read(audio_fd, buffer, BUF_SIZE*2);
91
92      write(STDIN_FILENO, buffer, BUF_SIZE*2);
93
94      return EXIT_SUCCESS;
```

```
95 }
```

清单7.1是 OSS 示例程序, 程序输出一段短音, 再从话筒读入一段信号并打印到终端上。将输出信号接到示波器上观察, 可以看到短时间 500Hz 的正弦波。由于输入声音信号打印出的是二进制数据, 不能正常阅读, 可将其重定向到文件后再对文件进行处理。

程序使用了数学函数 `sin()`, 编译时需要链接数学函数库 (选项 `-lm`)。

7.3 ALSA

目前 Linux 内核更多的采用了 ALSA 音频驱动。与 OSS 不同的是, ALSA 应用程序通过 alsa-lib 提供的 API 实现音频输入/输出功能。alsa-lib 库对底层系统功能调用 open(), ioctl(), read(), write() 做了封装, 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。编写 ALSA 程序需要先移植 alsa-lib 库。

7.3.1 移植 alsa-lib

```
ALSA 主页: https://www.alsa-project.org/wiki/Main_Page
alsa-lib 源码: https://www.alsa-project.org/files/pub/lib/alsa-lib-1.2.3.tar.
    ↪ bz2
```

```
在 alsa-lib 解压目录下执行
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    --disable-topology \
    --without-libdl \
    --without-pthread \
    --without-librt
$ make -j8
$ make install DESTDIR=/home/student/target
```

选项`--host` 用于指定交叉编译器前缀。这里指定的 arm-linux-gcc、arm-linux-g++、arm-linux-ld 等命令应属于环境变量 `PATH` 的列表目录中, 否则 configure 无法进行; 由于采用了共享库方式 (`--enable-shared`、`--disable-static`), 编译完成后, 应将生成的共享库复制到目标系统; configure 中最后三个选项是为了避免链接 libdl、libpthread 和 librt 库, 它们对 ALSA 的功能没有直接影响。如果链接了这些库, 还需要把交叉编译器中的这些库复制到目标系统的共享库可搜索路径中。在安装目录 usr/lib/ 中有 libasound 的共享库, usr/include/ 有相关头文

件, 其中 asoundlib.h 定义了与 ALSA 编程相关的宏和 API 函数原型, 它们是二次开发的基础; usr/share/alsa/ 有 ALSA 的配置文件。共享库和配置文件需按相同的目录结构移至目标系统。

7.3.2 ALSA 程序设计

当系统支持 ALSA 时, 列设备文件清单可以看到下面的文件:

```
# ls -l /dev/snd/
total 0
drwxr-xr-x  2 root root       60 Apr 23 20:20 by-id
drwxr-xr-x  2 root root       80 Apr 23 20:20 by-path
crw-rw---T+ 1 root audio 116,  3 Jan  1  2000 controlC0
crw-rw---T+ 1 root audio 116,  6 Apr 23 20:20 controlC1
crw-rw---T+ 1 root audio 116,  2 Jan  1  2000 pcmC0D0p
crw-rw---T+ 1 root audio 116,  5 Apr 23 20:20 pcmC1D0c
crw-rw---T+ 1 root audio 116,  4 Apr 23 20:20 pcmC1D0p
crw------T  1 root root  116,  1 Jan  1  2000 seq
crw-rw---T+ 1 root audio 116, 33 Jan  1  2000 timer
```

设备文件名中的字母 pcm 表示脉冲编码调制 (**P**ulse **C**ode **M**odulation), C0、C1 表示 Card 0 和 Card 1 (BeagleBone Black 未接 USB 声卡时只有 C0); D0 表示 Device 0 (一些声卡会有多个设备); p 表示播放设备 (**p**lay 的首字母), c 表示录音设备 (**c**apture 的首字母)。BeagleBone Black 板上只有 HDMI 的音频输出, 没有输入通道, 因此不存在文件 pcmC0D0c。

在 ALSA系统中, 设备按 “plughw:c,d” 格式命名 (低配版也可以使用 “hw:c,d”), 其中 c、d 分别对应卡号和设备号; 另有一个缺省的设备 “default”, 相当于 “plughw:0,0”。清单7.2是 ALSA 提供的一组 API 函数。7.3是一个简单的 ALSA 例程, 它在 USB 声卡的耳机上输出 5 秒声音, 左右通道各 110Hz 和 220Hz。程序使用一个包装了的函数, 简化了放音参数的设置。清单7.4是一个简单的音量调整程序。

清单 7.2: ALSA 主要 API 函数

```
1  #include <alsa/asoundlib.h>
2  ...
3      snd_pcm_t *handle;              /*设备文件句柄 */
4      snd_pcm_hw_params_t *hwparams;
5      char *device = "default";
6
7      /* 分配内存 */
8      snd_pcm_hw_params_alloca(&hwparams);
9
10     /* 打开 PCM 播放设备 */
11     err = snd_pcm_open(&handle, device, SND_PCM_STREAM_PLAYBACK, 0);
```

```
12      /* 打开 PCM 录音设备 */
13      snd_pcm_open(&handle, device, SND_PCM_STREAM_CAPTURE, 0);
14
15      /* 初始化 PCM 参数*/
16      snd_pcm_hw_params_any(handle, hwparams);
17
18      /* PCM 命令集的形式是
19              snd_pcm_hw_params_can_<capability>
20              snd_pcm_hw_params_is_<property>
21              snd_pcm_hw_params_get_<parameter>
22              snd_pcm_hw_params_set_<parameter>
23
24         一些重要的参数, 包括缓冲区大小、通道数、采样格式、速率等,
25         可以通过 snd_pcm_hw_params_set_<parameter> 调用实现 */
26
27      /* 设置数据格式: 16bit 小端、有符号数 */
28      snd_pcm_hw_params_set_format(handle,
29                                   hwparams,
30                                   SND_PCM_FORMAT_S16_LE);
31
32      /* 设置通道数据存储方式: INTERLEAVED or NONINTERLEAVED */
33      snd_pcm_hw_params_set_access(handle,
34                                   hwparams,
35                                   SND_PCM_ACCESS_RW_INTERLEAVED);
36
37      /* 设置通道数: 2 (stereo), 1 (mono) */
38      snd_pcm_hw_params_set_channels(handle,
39                                     hwparams,
40                                     2);
41
42      /* 设置采样率 44.1kHz */
43      val = 44100;
44      snd_pcm_hw_params_set_rate(handle,
45                                 hwparams,
46                                 val,
47                                 0);
48      /* 设置近似采样率 40kHz */
49      val = 40000;
```

```
50      dir = 0;
51      snd_pcm_hw_params_set_rate_near(handle,
52                                      hwparams,
53                                      &val,
54                                      &dir);
55
56      /* 数据输出 (DAC), 函数返回实际写入的帧数 */
57      ret = snd_pcm_writei(handle, buffer, num_of_frames);
58      /* NONINTERLEAVED 使用 snd_pcm_writen() */
59
60      /* 数据输入 (ADC), 函数返回实际读出的帧数 */
61      ret = snd_pcm_readi(handle, buffer, num_of_frames);
62      /* NONINTERLEAVED 使用 snd_pcm_readn() */
63
64      /* 使用结束后释放资源 */
65      snd_pcm_hw_params_free(hwparams);
66      snd_pcm_close(handle);
67
68      /* 混音器操作使用 snd_mixer_<parameter/property> 一组函数 */
69      snd_mixer_t *h_mixer;               /* 混音器句柄 */
70      snd_mixer_elem_t *elem;
71      snd_mixer_selem_id_t *sid;
72
73      /* 分配内存 */
74      snd_mixer_selem_id_alloca(&sid);
75
76      /* 打开混音器设备 */
77      err = snd_mixer_open(&h_mixer, 0);
78
79      /* 关联声卡 */
80      snd_mixer_attach(h_mixer, device);
81      snd_mixer_selem_register(h_mixer, NULL, NULL);
82      snd_mixer_load(h_mixer);
83
84      /* 从第一个元素开始 */
85      elem = snd_mixer_first_elem(h_mixer);
86      /* 持续到下一个元素 */
87      elem = snd_mixer_elem_next(elem);
```

```
88
89      /* 获得输出音量控制范围 */
90      long volMin, volMax, leftVal, rightVal, vol;
91      snd_mixer_selem_get_playback_volume_range(elem, &volMin,
92                                                &volMax);
93
94      /* 获得当前左右输出通道音量 */
95      snd_mixer_handle_events(handle);
96      snd_mixer_selem_get_playback_volume(elem,
97                          SND_MIXER_SCHN_FRONT_LEFT,
98                          &leftVal);
99      snd_mixer_selem_get_playback_volume(elem,
100                         SND_MIXER_SCHN_FRONT_RIGHT,
101                         &rightVal);
102
103     snd_mixer_selem_set_playback_volume_all(elem, vol);
104
105     /* 判断是否单声道 */
106     ret = snd_mixer_selem_is_playback_mono(elem);
107
108     /* 设置左右输出通道音量 */
109     /* 对于录音参数, 函数名中的 playback 替换成 capture */
110     snd_mixer_selem_set_playback_volume(elem,
111                         SND_MIXER_SCHN_FRONT_LEFT,
112                         vol);
113     snd_mixer_selem_set_playback_volume(elem,
114                         SND_MIXER_SCHN_FRONT_RIGHT,
115                         vol);
116
117     /* 关闭混音设备 */
118     snd_mixer_close(h_mixer);
```

清单 7.3: ALSA 输出示例 alsa.c

```
1  #include <alsa/asoundlib.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  #define FS      16000               /* 采样率 */
```

```
6
7  int main (int argc, char *argv[])
8  {
9      short buffer[FS];
10     snd_pcm_t *handle;
11     snd_pcm_sframes_t frames;
12     char *device = "plughw:1,0";
13     int err;
14
15     err = snd_pcm_open(&handle, device, SND_PCM_STREAM_PLAYBACK, 0);
16     if (err < 0) {
17         perror("Device Open.");
18         return EXIT_FAILURE;
19     }
20
21     snd_pcm_set_params(handle,
22                 SND_PCM_FORMAT_S16_LE,          /* 数据格式      */
23                 SND_PCM_ACCESS_RW_INTERLEAVED,  /* 间隔存取      */
24                 2,                              /* 双通道        */
25                 FS,                             /* 采样率        */
26                 1,                              /* 允许重采样    */
27                 50000                           /* 缓冲延迟(us) */
28             );
29
30     /* 构造双通道数据: 110Hz, 220Hz */
31     for (int i = 0; i < FS; i += 2) {
32         buffer[i  ] = 12000 * sin(2*M_PI*110*i/FS);
33         buffer[i+1] = 12000 * sin(2*M_PI*220*i/FS);
34     }
35
36     for (int i = 0; i < 10; i++) {
37         frames = snd_pcm_writei(handle, buffer, FS/2);
38         if (frames < 0)
39             frames = snd_pcm_recover(handle, frames, 0);
40         if (frames < 0)
41             break;
42     }
43
```

```
44      snd_pcm_close(handle);
45      if (frames < 0) {
46          printf("snd_pcm_writei failed: %s\n", snd_strerror(frames));
47          return EXIT_FAILURE;
48      }
49
50      return EXIT_SUCCESS;
51  }
```

清单 7.4: ALSA 输出音量控制示例 mixer.c

```
1   #include <alsa/asoundlib.h>
2   #include <stdio.h>
3
4   int main (int argc, char *argv[])
5   {
6       snd_mixer_t *hmixer;
7       snd_mixer_elem_t *elem;
8       snd_mixer_selem_id_t *sid;
9       long volMin, volMax, volLeft, volRight;
10      int err;
11
12      if (argc == 2) {
13          volLeft = volRight = atoi(argv[1]);
14      } else {
15          printf("run: %s volume, volume from 0 to 100\n", argv[0]);
16          return EXIT_FAILURE;
17      }
18
19      err = snd_mixer_open(&hmixer, 0);
20      snd_mixer_attach(hmixer, "hw:1");
21      snd_mixer_selem_register(hmixer, NULL, NULL);
22      snd_mixer_load(hmixer);
23
24      snd_mixer_selem_id_alloca(&sid);
25
26      elem = snd_mixer_first_elem(hmixer);
27      while (elem) {
28          /* 获得音量范围 */
```

```
29          snd_mixer_selem_get_playback_volume_range(elem, &volMin,
30                                                    &volMax);
31          snd_mixer_handle_events(hmixer);
32          volLeft = volMax * (volLeft/100.0);
33          volRight = volMax * (volRight/100.0);
34          snd_mixer_selem_set_playback_volume_all(elem, volLeft);
35          /* 设置左右通道音量 */
36              snd_mixer_selem_set_playback_volume(elem,
37                                          SND_MIXER_SCHN_FRONT_LEFT,
38                                          volLeft);
39              snd_mixer_selem_set_playback_volume(elem,
40                                          SND_MIXER_SCHN_FRONT_RIGHT,
41                                          volRight);
42              elem = snd_mixer_elem_next(elem);
43      }
44
45      snd_mixer_close(hmixer);
46      return EXIT_SUCCESS;
47  }
```

编译 ALSA程序需要链接 asound 库, 交叉编译时还要指定依赖库的路径。下面是编译 ALSA 程序的 Makefile:

```
CC = arm-linux-gcc
CFLAGS = -I/home/student/target/usr/include
LDFLAGS = L/home/student/target/usr/lib
LIBS = -lasound -lm

all: alsa mixer

alsa: alsa.c
   $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS) $(LIBS)

mixer: mixer.c
   $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS) $(LIBS)
```

7.4 实验内容

7.4.1 内核支持

选定一个 USB 声卡, 插入 PC, 查看它的型号:

```
$ dmesg | tail
...
usb 1-5: New USB device found, idVendor=8086, idProduct=0808
usb 1-5: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-5: Product: USB PnP Sound Device
usb 1-5: Manufacturer: C-Media Electronics Inc.
input: C-Media Electronics Inc. USB PnP Sound Device as /devices/pci0000
    ↪ :00/0000:00:14.0/usb1/1-5/1-5:1.3/0003:8086:0808.0004/input/input16
hid-generic 0003:8086:0808.0004: input,hidraw3: USB HID v1.00 Device [C-Media
    ↪ Electronics Inc. USB PnP Sound Device] on usb-0000:00:14.0-5/input3
usbcore: registered new interface driver snd-usb-audio
```

根据声卡型号, 配置目标系统内核。找到内核对应选项, 将其编入内核, 并使用该内核启动系统。如果是即插既用型声卡 (USB PnP Sound Device), 只需要选中 “USB sound devices” 即可, 否则还应在子项中选择对应型号。

OSS 支持项是 “ OSS Mixer API” 和 “OSS PCM API”。该选项在声卡驱动后生成 /dev/dspx、/dev/mixerx、/dev/audiox 设备文件。

声卡支持项也可以编译成模块, 在系统启动后手工加载。

```
Device Drivers  --->
    <*> Sound card support  --->
        <*>  Advanced Linux Sound Architecture  --->
            [*]  Enable OSS Emulation
            <M>    OSS Mixer API
            <M>    OSS PCM (digital audio) API
            ...
            [*]  USB sound devices  --->
```

系统启动后, 对比 USB 声卡插入前后 /dev/snd/ 目录下的变化。如果插入声卡后, 该目录下多出一组设备文件, 基本说明声卡可以工作。

7.4.2 编写应用程序

- 参考示例, 编写放音程序, 输出波形自定。用示波器观察声卡输出端信号验证你的结果。
- 编写录音程序, 利用已有的画图程序, 将录制的数据用波形图展示。

7.4.3 研究内容

- 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?

8

触摸屏

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。本实验学习触摸屏支持库的移植。触摸屏库除了用于支持图形接口环境以外, 它本身也可以作为应用软件编程的学习范例。

8.1 触摸屏的原理

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。目前应用于计算机系统的触摸屏, 按照其感应方式和传输介质的不同，可以分为电阻式、电容式、红外线式以及表面声波式等。每一种类型都有其各自的特点。

8.1.1 电阻式触摸屏

这是较早出现的一种触摸屏。它由两层透明的金属氧化物导电层构成 (见图 8.1), 通电后各自形成均匀的电场, 相当于均匀分布的电阻。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。测量 Px 和 Py 的分压值便可计算出触摸点的位置。嵌入式系统中通常采用 A/D 转换器测量这两个电压。

这类触摸屏工艺简单, 成本很低, 但由于精度低、难以实现多点触控、透光性能差等诸多问题, 现已被更先进的技术所替代。

8.1.2 电容式触摸屏

电容触摸屏利用人体电流感应进行工作 (见图8.2)。触摸屏的四个角上引出电极。当手指在触摸屏上触碰时, 人体电容和触摸板形成耦合, 在接触点吸入一个小电流, 这个电流分从触摸屏的四角上的电极中流出, 电流大小与触点到电极的距离有严格的对应关系。控制器通过精确计算这四个电流比例关系, 便可得到触摸点的位置。由于需要产生感应电流, 因此必须通过导体触摸。

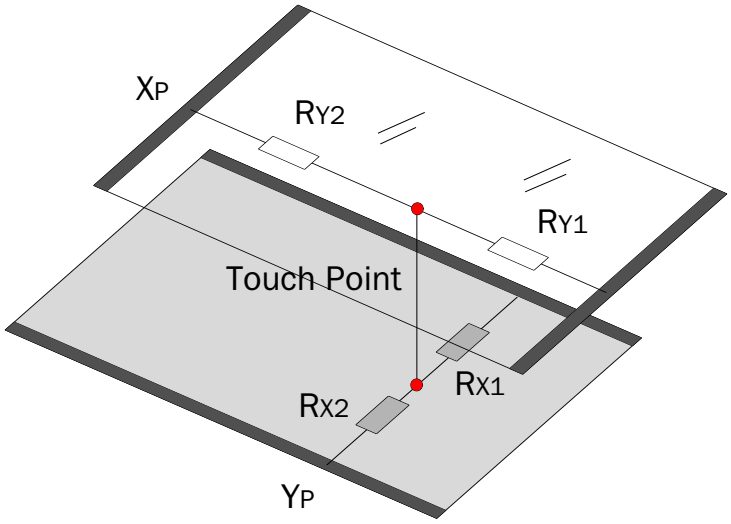


图 8.1: 电阻式触摸屏原理

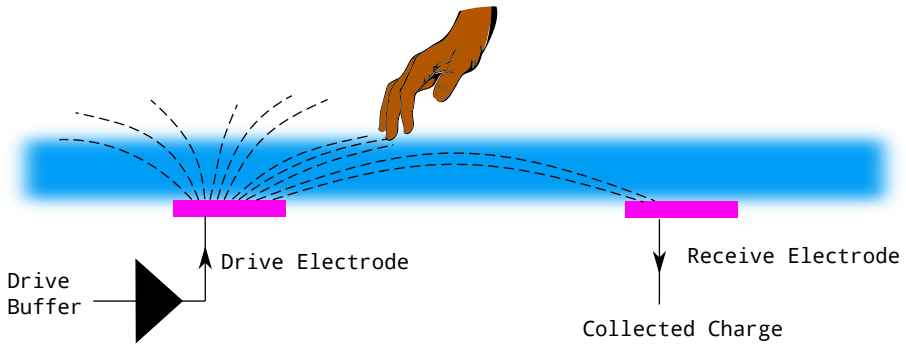


图 8.2: 电容式触摸屏原理

8.1.3 红外触摸屏

红外触摸屏利用 X、Y 方向上密布的红外线矩阵来检测并定位用户的触摸点。红外触摸屏在显示器的前面安装一个电路板外框，电路板在屏幕四边排布红外发射管和红外接收管，一一对应形成横竖交叉的红外线矩阵。用户在触摸屏幕时, 手指会挡住经过该位置的横竖两条红外线，因而可以判断出触摸点在屏幕的位置。

8.1.4 声表面波触摸屏

声表面波是在介质表面传播的超声波。这种类型的触摸屏, 三个角分别装有 X、Y 方向的发射换能器和接收换能器, 四个边刻着反射表面超声波的反射条纹 (见图 8.3)。工作时, 发射换能器产生的机械波沿玻璃表面传播 (声表面波), 经反射体反射后汇聚到接收换能器。当被触碰时, 部分声波能量被吸收, 改变了信号的特征。接收换能器收到的信号经过控制器处理便可以算出触点的坐标。根据接收信号的衰减量, 声表面波触摸屏还可以感知压力量。

声表面波触摸屏不受电信号干扰, 持久耐用, 容易制作大尺寸触摸屏。它不像电阻式或电容式触摸屏那样需要额外的感应层, 因此透光性好。由于它利用的是声波的吸收特性, 对硬物触摸

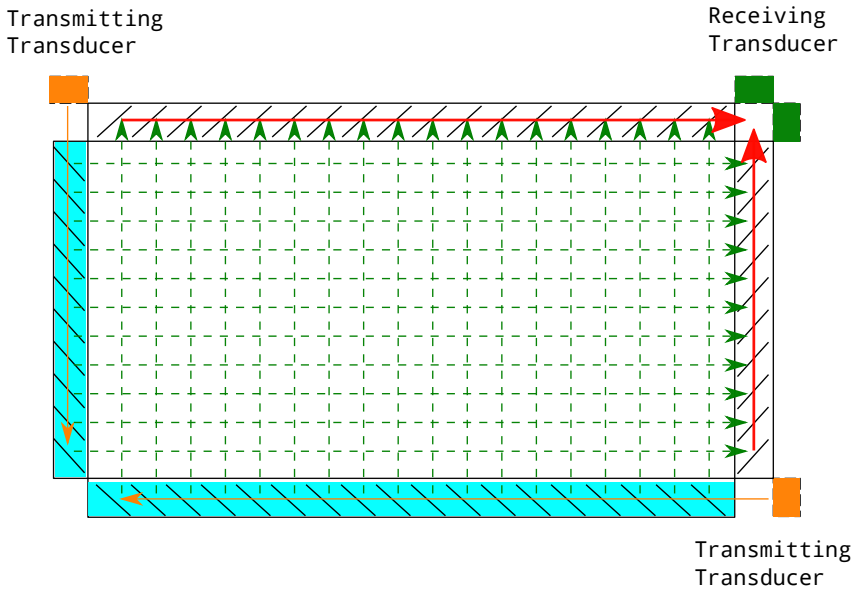


图 8.3: 声表面波触摸屏原理

无感, 易受水滴油渍干扰。

以上提到的几种触摸屏, 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 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:

```
Device Drivers  --->
    Input device support  --->
        <*>   Event interface
        ...
        [*]   Touchscreens  --->
            <*>   Ilitek ILI210X based touchscreen
```

8.2.1 触摸屏库 tslib

```
tslib 主页: http://www.tslib.org/
tslib 源码: https://github.com/libts/tslib/releases/download/1.21/tslib-1.21.
    ↪ tar.gz
```

下载、配置、编译、安装触摸屏库的完整过程如下:

```
$ wget https://github.com/libts/tslib/releases/download/1.21/tslib-1.21.tar.gz
$ tar xf tslib-1.21.tar.gz
$ cd tslib-1.21
$ mkdir build_arm && cd build_arm
$ ../configure \
   --host=arm-linux \
   --prefix=/usr \
   --enable-shared \
   --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

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, 否则配置选项 `--host=arm-linux` 设置的编译器无法正确加入。
- `--prefix` 选项用于安装目录的前缀, 缺省方式会将安装目录前缀设置为/usr/local/, 这是 Linux 系统的三级目录结构。
- `--enable-shared` 和`--disable-static` 用于设置库的生成形式: 是共享库还是静态库。大多数软件都提供这两种库形式的选项。如果不指定, 则同时生成共享库和静态库。共享库有助于减少多个软件依赖时的存储器资源占用, 但由于程序运行时的依赖, 需要一些额外的操作, 让程序能够找到共享库。
- DESTDIR 设置安装目录的起点, 请选择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`--prefix` 指定的目录, 不建议普通开发人员将文件复制到这个目录下, 特别是交叉编译时更不应该将目标平台的软件复制到主机的系统目录。

正确编译后, 会在安装目录下生成 usr/etc/、usr/bin/、usr/lib/、usr/include/、usr/share/ 等子目录, 详情如下:

```
target/
  `-- usr/
      |-- bin/
      |   |-- ts_calibrate              (校准程序)
      |   |-- ts_test                   (测试程序)
      |   `-- ...
      |-- etc/
      |   `-- ts.conf                   (触摸屏配置文件)
      |-- include/
      |   `-- tslib.h
      |-- lib/
      |   |-- libts.la                  (库链接信息)
      |   |-- libts.so -> libts.so.0.10.3
      |   |-- libts.so.0 -> libts.so.0.10.3
      |   |-- libts.so.0.10.3
```

```
|   |-- pkgconfig/
|   |   `-- tslib.pc              (软件包配置文件)
|   `-- ts/                       (插件目录)
|       |-- dejitter.la
|       |-- dejitter.so
|       |-- variance.la
|       |-- variance.so
|       `-- ...
`-- share/                        (文档目录)
        `-- man/
            |-- man1
            |   |-- ts_calibrate.1
            |   `-- ...
            `-- ...
```

etc/ts.conf 是库的配置文件; bin/ 下面是可执行程序, 包括触摸屏校准和测试工具; lib/ 目录下是触摸屏的共享库和模块插件; include/ 目录下的头文件、lib/ 下的库文件和.la 文件、lib/pkgconfig/ 下的.pc 文件用于二次开发; share/ 目录下通常是数据和文档。

8.2.2 触摸屏库的安装和测试

将 bin/ 目录下的程序、etc/ 下的配置文件和 lib/ 目录下的共享库 (包括子目录 ts/ 下的插件库) 文件按目录对应关系复制到目标系统的 /usr/ 目录下, 在目标系统中按下面的方式设置环境变量:

- 触摸屏设备文件 TSLIB_TSDEVICE

  由于触摸屏已加入 event 接口。event 设备的主设备号是 13, 次设备号从 64 开始编号。开发者可以通过查阅文件 /proc/bus/input/devices 获得触摸屏的设备情况, 与 /dev/input/ 目录下的设备文件属性对照, 确定触摸屏对应的设备文件。用如下命令设置环境变量:

  ```
  # export TSLIB_TSDEVICE=/dev/input/event2
  ```

  不确定的情况下, 在终端执行下面的命令:

  ```
  # cat /dev/input/event2
  ```

  然后触碰触摸屏, 看看是否有反应;

- 触摸屏配置文件 TSLIB_CONFFILE

  ```
  # export TSLIB_CONFFILE=/usr/etc/ts.conf
  ```

  触摸屏函数会根据这项设置读取配置文件。配置文件来自触摸屏库源码, 可在此基础上进行修改;

- 触摸屏模块插件目录 TSLIB_PLUGINDIR

  ```
  # export TSLIB_PLUGINDIR=/usr/lib/ts
  ```

  它是插件模块文件 (.so) 所在目录;

- 终端设备 TSLIB_CONSOLEDEVICE

```
# export TSLIB_CONSOLEDEVICE=none
```

缺省的终端设备是/dev/tty;

- 帧缓冲设备文件 TSLIB_FBDEVICE

```
# export TSLIB_FBDEVICE=/dev/fb0
```

- 校准文件 TSLIB_CALIBFILE

```
# export TSLIB_CALIBFILE=/etc/pointercal
```

早期触摸屏由于工艺原因, 装配到显示屏上以后, 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。这个触摸屏库本身带有校准程序 ts_calibrate。校准的原理是, 程序在显示屏已知位置上画个标记, 操作员在标记处触摸, 程序将触摸屏读取的数据与显示屏坐标进行对比, 由此建立显示屏和触摸屏的坐标对应关系。

还要根据要求编辑配置文件 /usr/etc/ts.conf, 一般需要保留这几个模块:

- module_raw input, 它表明使用 Event interface;
- pthres, 压力测量;
- dejitter, 去除抖动, 保持触点坐标平滑变化;
- linear, 线性坐标变换。

以上准备工作就绪后, 尝试执行 /usr/bin/ts_test。电阻式触摸屏还需要先运行校准程序 ts_calibrate, 根据屏幕提示操作, 生成校准文件, 否则无法正确建立显示屏和触摸板的坐标对应关系。

在目标系统上, 缺少动态链接库的程序运行时, 会出现类似如下的错误提示:

```
# ts_test
error while loading shared libraries: libdl.so.2: cannot open shared
   ↪ object file: No such file or directory.
```

这种情况, 需要把交叉编译器的共享库 libdl 复制到目标系统。

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。

9

媒体播放器

9.1 软件简介

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格式, 以 GNU 通用公共许可证发布, 可在各主流操作系统使用, 曾经是 Linux 系统中最重要的播放器之一。MPlayer 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, 但也也可以编译成 GUI 方式。不同操作系统、不同发行版可以为它配置不同的图形界面皮肤, 使其外观多姿多彩。

```
zlib 主页: http://www.zlib.net
zlib 源码: http://zlib.net/zlib-1.2.11.tar.xz
libmad 主页: http://libzplay.sourceforge.net/LIBMAD.html
libmad 源码: https://downloads.sourceforge.net/mad/libmad-0.15.1b.tar.gz
Mplayer 主页: http://www.mplayerhq.hu/design7/news.html
Mplayer 源码: http://www.mplayerhq.hu/MPlayer/releases/MPlayer-1.4.tar.xz
```

MPlayer 除了可以播放一般的磁盘媒体文件外, 还支持 CD、VCD、DVD 等多种物理设备和多种网络媒体 (rtp://、rtsp://、http://、mms:// 等)。播放视频时, 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 项目的 libavcodec 函数库原生支持。FFmpeg (**F**ast **F**orward **mpeg**) 是一个独立的多媒体编解码开源库。MPlayer 源码内置 FFmpeg, 无需单独编译编解码库。对于那些没有开源解码器的格式, MPlayer 使用二进制的函数库, 据称它能直接调用 Windows 的动态链接库。

9.2 编译

下载 zlib、libmad 和 MPlayer 源代码。

9.2.1 编译 zlib

zlib 是 Linux 系统的压缩/解压基础库, 也是最常用的压缩算法。压缩 RAM Disk 用到的 gzip/gunzip 命令就是基于 zlib 库的算法。下面是编译、安装 zlib 的过程:

```
在 zlib 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ CC=arm-linux-gcc ../configure --prefix=/usr
$ make -j8
$ make install DESTDIR=/home/student/target
```

执行安装命令`make install` 会将库文件复制到 /home/student/target/usr/lib 目录下, 同时将头文件复制到 /home/student/target/usr/include, 用作二次开发。共享库还需复制到目标系统的 /usr/lib/ 目录下, 程序运行时需要用到。使用共享库可以节省目标系统的存储空间。

9.2.2 编译 libmad

libmad (mad: **m**PEG **a**udio **d**ecoder) 是高品质全定点算法的 MPEG 音频解码库。MPlayer 对于 libmad 的依赖不是强制性的。libmad 对于定点处理器的 MPEG 算法效率非常重要, 在浮点处理器上也可起到降低功耗的作用。

```
在 libmad 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-static \
    --enable-fpm=arm
$ (修改文件 Makefile)
$ make -j8
$ make install DESTDIR=/home/student/target
```

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

9.2.3 编译 MPlayer

```
在 mplayer 解压目录下执行:
$ ./configure \
   --cc=arm-linux-gcc \
   --target=arm-linux \
   --prefix=/usr \
   --enable-mad \
   --enable-ossaudio \
   --extra-cflags="-I/home/student/target/usr/include" \
```

```
    --extra-ldflags="-L/home/student/target/usr/lib"
$ make -j8
$ make install INSTALLSTRIP="-s --strip-program=arm-linux-strip" DESTDIR=/home
    ↪ /student/target
```

`configure` 的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安装路径。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 和 mencoder。最后通过`make install` 安装到变量 `DESTDIR` 指定的目录。安装过程的 strip 需要用交叉编译工具中的命令, 不能使用 PC 默认的编译工具处理, 这里用了一个比较复杂的参数设置。由于此时可执行文件 mplayer 和 mencoder 已经生成, 因此也可以跳过安装过程, 直接使用交叉编译工具链的 strip 命令处理可执行文件, 然后再用 cp 命令复制。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 /usr/lib/ 目录里。

9.3 实验内容

尝试在目标系统上移植 MPlayer, 用 mplayer 播放一些音频文件和视频文件。试分析视频卡顿的原因, 还有哪些系统资源没有充分利用?

借助 mplayer/mencoder 的帮助文档, 学习使用 mencoder 完成一些音视频转码工作。

10

系统功能调用

用户空间的程序向操作系统内核请求服务, 这些服务通常具有更高的权限, 此时需要通过系统功能调用来实现。系统功能调用提供用户程序与操作系统之间的接口。

10.1 Linux 系统功能调用

Linux 内核中设置了一组用于实现各种系统功能的子程序, 称为系统功能调用, 或简称系统调用。一些操作系统的系统调用是通过特定的软中断实现的, 例如 DOS 操作系统的 DOS 功能调用 “int 0x21”, Linux 操作系统在 X86 平台的系统调用中断号是 0x80。用户可以通过系统调用函数在自己的应用程序中调用它们。实际上, Linux 应用程序使用的系统调用函数是 Glibc 对内核函数的包装, 在内核中, 系统调用对应某个软中断的序号。

从形式上看, 经过 Glibc 包装的系统功能调用和普通的函数调用没有差别, 它们之间真正的差别在于运行空间不同: 系统调用由操作系统核心提供, 运行于内核空间, 而普通函数调用由函数库或用户自己提供，运行于用空间。目前, Linux 内核提供包括存储管理、文件系统、进程管理、I/O 接口及网络管理约 380 个左右的系统功能调用。

系统调用在 Linux 系统中有着重要的意义。如果没有系统调用, 应用程序就失去了内核的支持。一些用户空间的函数, 如`fopen()` 功能, 最终也需要通过调用系统函数`open()` 实现。

10.2 给内核增加系统调用

增加系统功能调用需要向内核中添加功能代码。内核文件 include/uapi/asm-generic/unistd.h 中记录了每个系统调用的调用号和函数入口。`__NR_syscalls` 是系统调用的总数, 不同系统、不同版本中, 这个数字是不同的。下面假设当前内核已有 398 个系统调用, 在此基础上增加一个。

1. 修改 unistd.h 文件:

```
...
#define __NR_sayhello   398
```

```
__SYSCALL(__NR_sayhello, sys_sayhello)

#undef __NR_syscalls
#define __NR_syscalls   399
...
```

原有的系统功能调用号为 0 到 397。此处增加一个系统调用, 总数变为 399。新的系统调用编号为 398。该系统调用对应的实现函数入口是`sys_sayhello`。

2. 添加功能代码, 在 kernel 目录下添加一个文件 sayhello.c, 见清单10.1。

清单 10.1: 系统功能调用 sayhello.c

```
1 #include <linux/syscalls.h>
2
3 SYSCALL_DEFINE3(sayhello, char *, s, int, a, int, b)
4 {
5     printk("THIS IS NEW SYSCALL\n");
6     printk("%s\n", s);
7     printk("%d %d\n", a, b);
8     return a + b;
9 }
```

`SYSCALL_DEFINEx` 是内核定义的系统调用函数模板, 数字表示函数参数的数量, 其参数形式为:

```
    函数名, 参数1类型, 参数1名称, 参数2类型, 参数2名称...
```

3. 修改该目录下的 Makefile, 将这个文件加入编译项:

```
obj-y =      fork.o exec_domain.o panic.o \
...
obj-y += sayhello.o
...
```

10.3 用户层调用

内核函数并没有暴露到用户空间, 因此用户程序不能直接通过内核的函数名来调用系统函数。用户程序可以仿照下面的方法实现系统调用:

```
#define __NR_sayhello   398
...
int ret = syscall(__NR_sayhello, "hello", 100, 234);
```

程序执行后, 可用命令`dmesg` 查看内核打印信息, 以及`syscall()` 函数的返回值, 与程序功能对照结果。

10.4 实验内容

为内核增加一个系统功能调用, 该系统功能调用可获取内核的时间。尝试在用户空间访问这个函数。

11

设备驱动

Linux 内核按模块的方式组织功能, 系统调用、设备驱动都可以作为模块。系统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, 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。

11.1 设备驱动程序结构

设备驱动常常被设计成可加载模块。加载时, 它先检查并设置设备的工作状态, 然后注册一些设备操作的功能 (如读、写、I/O 控制等等), 接着就是等待被用户程序调用执行。

设备驱动可以编译到内核中, 在系统启动后便具有这项功能, 这种方法在嵌入式 Linux 系统中经常被采用, 它减少了系统启动后的模块加载工作。而在开发阶段, 设备驱动的动态加载则更为普遍, 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。

11.1.1 模块的加载和卸载

可加载模块有两个函数: `init_module()` 和 `cleanup_module()`。前者在执行系统命令`insmod` (**ins**tall **mod**ule) 时运行, 它通常完成模块的注册功能; 后者在执行系统命令`rmmod` (**rm**ove **mod**ule) 时运行, 将模块功能从内核中移除。设备驱动在 `init_module()` 中向内核注册一个主设备号和一组设备操作方法, 设备操作方法封装在 `file_operations` 结构中。

清单 11.1: 设备驱动程序结构 newdriver.c

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/fs.h>
4
5 int dev_open(struct inode *inode, struct file *file)
6 {
7     return 0;
```

```
8  }
9
10 int dev_release(struct inode *inode, struct file *file)
11 {
12     ...
13     return 0;
14 }
15
16 ssize_t dev_write(struct file *file,
17                   const char *buffer,
18                   size_t length,
19                   loff_t *offset)
20 {
21     ...
22     return length;
23 }
24
25 ssize_t dev_read(struct file *file,
26                  char *buffer,
27                  size_t length,
28                  loff_t *offset)
29 {
30     ...
31     return length;
32 }
33
34 struct file_operations fops = {
35     owner:   THIS_MODULE,
36     open:    dev_open,
37     release: dev_release,
38     read:    dev_read,
39     write:   dev_write,
40 };
41
42 int init_module(void)
43 {
44     if(register_chrdev(123, "module name", &fops)){
45         return -1;          /* 注册不成功，模块退出 */
```

```
46      } else {
47          return 0;           /* 模块驻留 */
48      }
49  }
50
51  void cleanup_module(void)
52  {
53      unregister_chrdev(123, "module name");
54  }
55
56  MODULE_LICENSE("GPL");
```

11.1.2 编译模块

编译内核模块依赖内核源码, 它通过下面的 Makefile 完成编译:

```
1  obj-m       := newdriver.o
2  KDIR        := /home/student/linux-4.14
3  PWD         := $(shell pwd)
4  default:
5          $(MAKE) -C $(KDIR) M=$(PWD) modules
6  clean:
7          $(MAKE) -C $(KDIR) M=$(PWD) clean
```

此处的目标文件名 newdriver.o 与 C 源文件名对应, KDIR 是指向内核源码路径的变量, PWD 指向当前路径。交叉编译时, 还要向 GNU Make 传递处理器架构名称和编译器名称:

```
$ ARCH=arm CROSS_COMPILE=arm-linux- make
```

正确编译后, 会生成模块文件 newdriver.ko。

11.1.3 设备驱动的使用

在目标系统上, 通过下面的命令加载设备驱动:

```
# insmod newdriver.ko
```

为了使用设备驱动程序提供的功能, 还需要创建一个设备文件。设备文件的 主设备号是模块注册设备时申请的设备号, 每一类设备的主设备号是唯一的, 否则不能成功注册; 次设备号由驱动程序内部决定。设备文件通常放在/dev 目录中。可按下面的命令创建一个字符设备文件:

```
# mknod /dev/testdev c 123 0
```

设备文件名只起到提供给应用程序识别的作用, 真正起作用的是设备号。应用程序通过系统功能调用实现设备驱动提供的方法, 从而实现对设备的控制。图11.1描述了用户程序如何通过设备文件与驱动程序发生关系的。

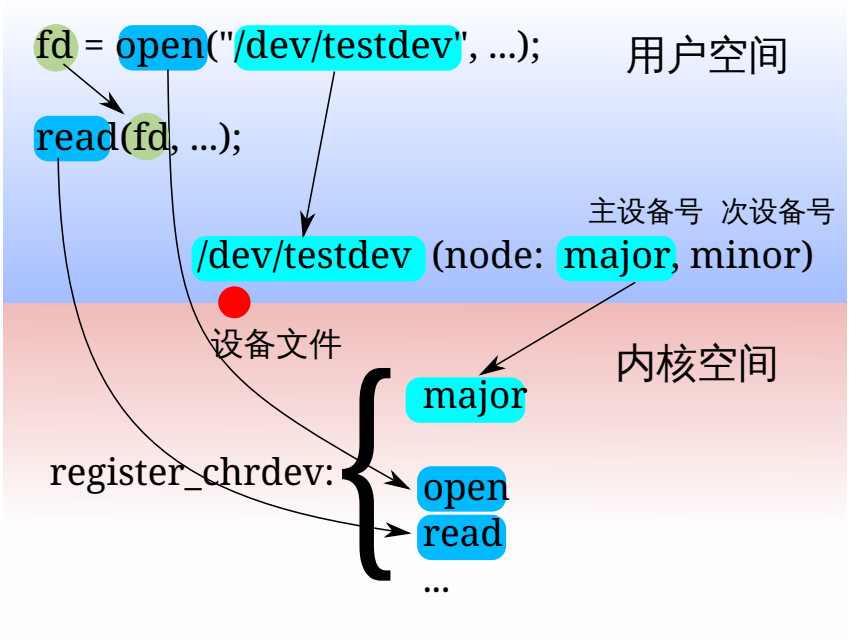


图 11.1: 用户程序、设备文件与设备驱动的关系

11.2 嵌入式处理器的片上设备

处理器内部的可编程 I/O 接口设备称为片上设备。嵌入式处理器片内集成了大量的片上设备, 为灵活构成各种应用系统提供了极大的便利。这些设备包括中断控制器、可编程定时器、GPIO、串行接口等等。其中 GPIO 是最简单也是最常用的一类接口。

AM335x 实现 4 组 GPIO 模块、每组有 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。很多 GPIO 的引脚可能与其他功能复用。

设备驱动程序负责将应用程序的读、写等操作传递给相关的硬件, 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。

本实验学习如何通过 GPIO 控制设备, 掌握 Linux 设备驱动的基本方法。

11.2.1 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)
void iounmap(void * addr);
```

`ioremap()` 的两个参数分别是需要映射的首地址和长度, 返回虚拟地址; `iounmap()` 释放被映射的虚拟地址。

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。

11.2.2 LED 控制

在 BeagleBone 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, 电路原理图见11.2。当用于控制的某个 GPIO 引脚输出高电平时, 对应的三极管导通, 电流流过 LED 而被点亮。我们可以通过下面的方式控制 usr0 LED (GPIO1_21):

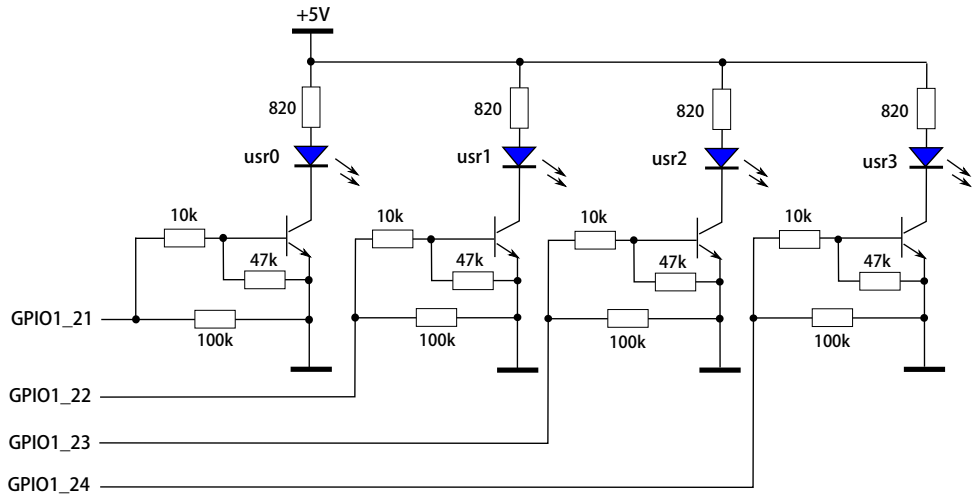


图 11.2: BeagleBone Black LED

```
1 #define GPIO1_BASE      0x4804C000
2 #define GPIO_OE         (GPIO1_BASE+0x134)
3 #define GPIO_IN         (GPIO1_BASE+0x138)
4 #define GPIO_OUT        (GPIO1_BASE+0x13C)
5 ......
6
7 volatile int *pConf     = ioremap(GPIO_OE, 4); /* 映射方向寄存器 */
```

```
8  volatile int *pDataIn  = ioremap(GPIO_IN, 4); /* 映射输入寄存器 */
9  volatile int *pDataOut = ioremap(GPIO_OUT, 4);/* 映射输出寄存器 */
10
11     *pConf    &= ~(1<<21);                /* 将pin21设为输出      */
12     *pDataOut |=  (1<<21);                /* pin21 置高电平, 灯亮*/
13
14     *pDataOut &= ~(1<<21);                /* pin21 置低电平, 灯灭*/
15
16     ......
```

11.3 实验内容

BeagleBone 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。

12

移植 Python

Python 是多模态的编程语言, 支持完整的面向对象编程和结构化编程的特性。Python 拥有一套强大的标准库和大量的第三方模块, 它们的功能覆盖系统管理、数学计算、网络通信、文本处理、数据库、图形接口等等几乎所有的应用软件编程领域。

12.1 Python 简介

Python 是一种广泛使用的解释型高级编程语言，作者是荷兰程序员 Guido van Rossum, 创作于 1990 年。进入 21 世纪, Python 语言发展迅猛, 在 2007, 2010, 2018 和 2020 年数度成为 TIOBE[1]明星编程语言。

Python 的设计思想强调代码的可读性和简洁的语法规则，让程序员使用较少的代码表达设计思想。不论程序规模大小，Python 语言的规则都是尽可能让程序结构清晰明了。通过大量的模块化设计，Python 可以让软件开发者把注意力集中在算法逻辑上，而不是细节实现上。

Python 属于自由软件。它的版权协议几经变化，目前以 PSFL (**P**ython **S**oftware **F**oundation **L**icense, Python 软件基金会版权协议) 版权协议发布。属于开源软件协议的一种, 与 GPL 兼容。不同的是, 它不强制发布修改后的软件版本必须提供源代码。

Python 是跨平台的编程语言，它是很多操作系统标准的组件。大多数 Linux 发行版和 Mac OS 都集成了 Python, 无需单独安装即可以直接使用其基本功能。在 Linux 系统中，Python 不仅仅是一种编程语言或开发工具，很多基础软件都依赖 Python, 或者直接是由 Python 语言提供的。

12.2 Python 的移植

Python 官方网站 https://www.python.org/downloads 提供了 PC 各种操作系统的安装包，使用者可根据各自的需要安装相应的版本。但想在一些嵌入式平台上使用 Python，可能需要自己从源代码开始编译。Python 是跨平台的编程语言，但运行环境与硬件平台和操作系统都

[1]TIOBE 为一家荷兰公司, 该公司定期发布增长最快的编程语言, 称为 TIOBE 指数。

有关。

Python 的下层依赖库主要有 SQLite、OpenSSL/LibreSSL、Readline、Tcl/Tk 等。SQLite 是用于嵌入式系统的轻量级关系数据库管理系统, OpenSSL/LibreSSL 提供数据安全算法, Python 图形模块通过 Tcl/Tk 支持, 是 Python 的半标准图形库, Python 命令行环境 (Python-shell) 的行编辑功能通过 Readline 支持, 它允许用户使用上下方向键回溯命令、左右方向键和退格键编辑命令。Python 还依赖 libuuid[2], 它来自 Linux 基础系统的应用软件包 util-linux。所有这些都不是必须的选项, 用户可以根据自己需要的功能移植其中的一部分。

本实验移植一个简化的 Python, 只保留 OpenSSL/LibreSSL、Readline、util-linux 和 Python 本身。

```
openssl 主页: https://www.openssl.org
openssl 源码: https://www.openssl.org/source/openssl-1.1.1f.tar.gz
libressl 主页: https://www.libressl.org
libressl 源码: https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/libressl-2.7.2.tar
    ↪ .gz
ncurses 主页: https://invisible-island.net/ncurses/ncurses.html
ncurses 源码: https://invisible-mirror.net/archives/ncurses/ncurses-6.1.tar.gz
readline 主页: https://www.gnu.org/software/readline
readline 源码: ftp://ftp.cwru.edu/pub/bash/readline-8.0.tar.gz
util-linux 源码: https://www.kernel.org/pub/linux/utils/util-linux/v2.33/util-
    ↪ linux-2.33.2.tar.xz
python 主页: https://www.python.org
python 源码: https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tar.xz
```

12.2.1 编译 OpenSSL/LibreSSL

OpenSSL 是安全套接字层 (**S**ecure **S**ockets **L**ayer, SSL) 和传输层安全 (**T**ransport **L**ayer **S**ecurity, TLS) 协议的开源实现, 是 Linux 系统中重要的安全库, 广泛应用在互联网的网页服务器和网络通信软件上。OpenSSL 以 Apache 及 BSD 版权协议发布, 其条款与 GPL 有冲突。GPL 软件使用 OpenSSL 时需予以例外。

由于 OpenSSL 与 GPL 版权协议不兼容, GNU 项目于 2003 年 3 月启动了 GNUTLS。因开发者与 GNU 项目负责人的分歧, 该项目目前已和 GNU 项目脱离, 但仍以开源版权协议 LGPLv2.1 发布[3]。

2014 年 4 月, OpenSSL-1.0.1 版本曝出一个严重的安全漏洞 (Heartbleed), 一些 OpenBSD 的开发者由 OpenSSL-1.0.1g 分支出了 LibreSSL, 在保留原有主要功能的基础上大幅删减了来自主线的代码, 可视其为 OpenSSL 的简化版本。

编译 LibreSSL 的过程如下:

[2]UUID: **U**niversally **U**nique **ID**entifier。

[3]GNUTLS 主页: https://www.gnutls.org。

```
在 libressl 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-openssldir=/etc \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

OpenSSL 的编译方法略有不同, 下面是一个较简化的过程:

```
在 openssl 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../Configure linux-armv4 \
    CC=arm-linux-gcc \
    shared \
    --prefix=/usr \
    --openssldir=/etc/ssl
$ make -j8
$ make install DESTDIR=/home/student/target
```

参数`linux-armv4` 指明是 Arm 32 位 Linux 系统, 选项`--openssldir` 指明配置文件的安装路径, 默认的安装路径以`--prefix` 指定的路径为起点。这里按桌面系统的配置习惯, 将它改变到 FHS 的一级目录结构中。LibreSSL 的`--with-openssldir` 选项也是出于同样的考虑。

LibreSSL 比 OpenSSL 的规模要小得多, 功能基本满足要求, 建议移植 LibreSSL 而不是 OpenSSL。

12.2.2 编译 Readline

命令行中, Readline 提供命令编辑功能, 回溯操作历史, 用于改善 Python Shell 的可操作性。其中的部分函数接口来自 ncurses 库。ncurses 实现在字符终端下类似图形操作界面的效果, 如同在配置内核、BusyBox 时使用 `make menuconfig` 看到的界面。编译过程如下:

```
在 ncurses 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --without-normal \
    --without-debug \
```

```
    --with-shared \
    --without-progs
$ make -j8
$ make install DESTDIR=/home/student/target
...
在 readline 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-curses \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

12.2.3 编译 util-linux

util-linux (Linux **util**ities) 是 Linux 内核官网维护的 Linux 基础工具软件, 其中的大部分命令与 BusyBox 是重合的, 这里只需要编译出其中的一个库 libuuid:

```
在 util-linux 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-all-programs \
    --enable-shared \
    --disable-static \
    --enable-libuuid
$ make -j8
$ make install DESTDIR=/home/student/target
```

12.2.4 编译 Python

交叉编译 Python 需要主机具备与之相同的大版本环境, 即, 编译 Python-3.8.2, 主机 Python 环境至少要求 Python-3.8.0。如果主机 Python 版本过低, 有两种解决办法: 一、通过发行版的包管理器升级主机的 Python 版本, 但发行版维护的软件源通常都赶不上 Python 的升级速度; 二、将待移植的 Python 先在主机上编译、安装。主机编译比交叉编译要容易得多,

configure 时无须指定交叉编译器、无须设置依赖库路径搜索, 甚至不需要完整的功能, 只需要能运行这个版本的 Python 命令。多数情况只需要下面的几个步骤:

```
在 python 解压目录下执行:
$ mkdir build_x86 && cd build_x86
$ ../configure
$ make -j8
# make install
```

最后一步的安装命令需要超级用户权限, 安装的起点是 */usr/local/* 目录, 这是 *FHS* 标准规定的三级目录结构。建议这项工作由管理员完成。无管理员权限的用户也可以安装在自己的目录中 *(make install DESTDIR=...)*, 再通过设置环境变量 `PATH` 选择优先运行自己目录的程序。

交叉编译过程如下:

```
在 python 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --build=x86_64-linux-gnu \
    ac_cv_file__dev_ptmx=no \
    ac_cv_file__dev_ptc=no \
    ac_cv_buggy_getaddrinfo=no \
    --without-cxx-main \
    --with-ensurepip=no \
    --enable-optimizations \
    CFLAGS="-I/home/student/target/usr/include" \
    CPPFLAGS="-I/home/student/target/usr/include" \
    LDFLAGS="-L/home/student/target/usr/lib"
$ make -j8
$ make install DESTDIR=/home/student/target
```

`configure` 通过编写一些小测试程序、编译, 来确定当前的编译参数, 但在交叉编译时, 一些针对目标平台的测试在主机平台上无法进行。上面配置命令中的 `ac_cv_*` 选项通过人为设定编译参数, 跳过交叉编译器的检测。

12.3 Python 基本使用

12.3.1 交互方式

终端上键入 python3 启动交互方式[4], 出现提示符 “>>>”。在这个环境中可以直接输入 Python 语句, 即刻得到指令的执行结果。使用 CTRL+D 或者 `exit()` 退出 Python Shell:

```
# python3
Python 3.8.2 (default, Jun 14 2020, 22:51:06)
[GCC 6.1.0] on linux
Type "help", "copyright", "credits" or "license" for more
   ↪ information.
>>> print ("Hello, Python!")
Hello, Python!
>>> ^D
#
```

12.3.2 运行 Python 程序

下面通过一个简单的例子说明如何在命令行中运行 Python 程序:

清单 12.1: 一个简单 Python 程序 hello.py

```
#!/usr/bin/python3
print("Hello, Python!")
```

文件名后缀对于 Linux 系统不重要。命令行方式下, 可以有下面几种方法运行 Python 程序:

- 显式使用 python3 命令解释 Python 文件中的语句:

  ```
  $ python3 hello.py
  Hello, Python!
  $
  ```

  这种情况下, 源文件第一行可理解为注释行。

- 给 Python 源文件加上可执行属性, 将 Python 源文件当作可执行程序运行:

  ```
  $ chmod +x hello.py
  $ ./hello.py
  Hello, Python!
  $
  ```

[4]“python” 命令默认链接到 python2。Python 开发者很早就建议用户转移到 Python3 的版本, 逐渐淘汰 Python2。但直到 2020 年, Linux 仍有许多基于 Python2 版本的程序在使用中。

这种情况下, 文件开头的两个字符 “#!” 具有特定的功能, 它是脚本文件的标识。系统会根据后面指定的程序作为解释器。

12.4 实验内容

1. 移植 Python 到目标系统;
2. 编写一个简单的 Python 程序, 功能自定, 在目标系统上运行。

13

I2C 设备

I2C (**I**nter-**I**ntegrated **C**ircuit, 又写作 IIC 或 I^2C) 是由 Philips 半导体公司 (如今的 NXP 半导体公司) 在上世纪八十年代开发的一种同步串行总线, 设备之间只通过一根数据线和一根时钟线连接。因其结构简单, 广泛应用于嵌入式系统主机与低速外围设备的互联。

13.1 I2C 时序

在微机应用领域, 支持 I2C 总线的设备很多, 比较常见的有 FLASH 存储器、AD/DA 转换器、多位 LED 显示器和点阵 LED 显示器等等 (图13.1)。每一种设备都有一个固定的地址, 出厂时被固化在芯片内, 作为协议的一部分。典型的 I2C 数据传输性能在 100kb/s 到 400kb/s 之间, 最高传输速度可达 5Mb/s (超快模式)。

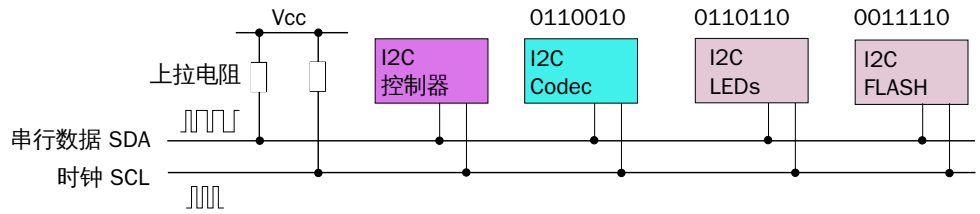


图 13.1: I2C 总线设备的连接

根据各设备在总线上的地位, I2C 总线共有四种不同的操作模式, 大部分设备只作为一种角色, 使用其中两种操作模式。这里仅以主设备收发模式为例, 说明 I2C 总线信号的传输过程, 见图13.2。

总线空闲期间, 数据线处于高电平状态。主控器发送一个起始位 START (从高到低的下降沿) 启动通信, 接着发送 7bit 地址位[1], 该地址应与其希望联系的从设备地址对应。最后再发送 1bit 的读写位, 表示在后面的时钟周期里, 主设备是从从设备里读数据, 还是向从设备里写数据。

从设备从 START 开始接收地址信息, 当监测到与自己地址匹配时, 在总线上回应一个低电平有效的应答信号 ACK, 与自己地址不匹配的设备不作应答。主设备收到应答后, 依照其自身

[1]I2C 有 7bit 和 10bit 两种地址格式, 这里仅讨论 7bit 形式。

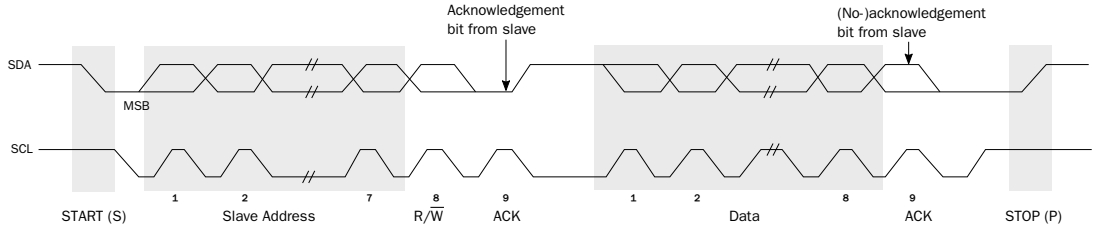


图 13.2: I2C 总线时序 (读)

设定的读写方式, 将数据线置于接收或模式发送模式, 从设备则处于对应的相反模式。地址和数据都是高位在前、低位在后的形式。如果主设备是读方式, 它将从数据线上读取一个个字符, 每接收一个字符后, 发送一个 ACK 作为应答, 接收到最后一个字符后, 发送一个从低电平到高电平上升沿的停止信息位 (STOP), 结束这一轮通信; 如果是写方式, 则是主设备发送完一个字符后, 从设备以 ACK 作为应答, 如此重复, 直到主设备发送 STOP 结束通信。

13.2 I2C 设备的使用

Linux 的 I2C 总线被驱动后, 会在/dev/目录下生成设备文件 i2c-x, “x” 是用数字 0、1、2... 标识的总线号, 它们的主设备号是 89。BeagleBone Black 板上的 HDMI TDA988X、电源管理 TPS65217C 和一片 EEPROM 通过 I2C 总线 i2c-0 控制, 外接扩展 P9 引出了两组 I2C 主控接口, 见表13.1。

表 13.1: BeagleBone Black P9 引出的 I2C

	SCL	SDA
I2C-1	pin17	pin18
I2C-2	pin19	pin20

13.2.1 嵌入式系统的 I2C 控制器

这里以 BeagleBone Black 配备的 32Kbit EEPROM (24LC32AT-I/OT) 为例说明 I2C 协议格式。24LC32 作为板载 ID, 连接在 i2c-0 设备上。该设备的 7bit 地址形式是:

1	0	1	0	A2	A1	A0

在给定地址上读取单个存储单元的内容, 时序如图13.3。24LC32 片内有一个地址计数器, 保留上一次访问的地址。每次读写单元后, 地址自动增加。因此连续读写时不需要每次写地址请求。清单13.1读入 24LC32 的起始四个字节, 根据 BeagleBone Black 文档说明, 它们应该是 “0xAA 0x55 0x33 0xEE”。

BusyBox 提供了一组 I2C 操作命令。它们的功能如下:

- i2cdetect: 检测总线上的 I2C 设备。

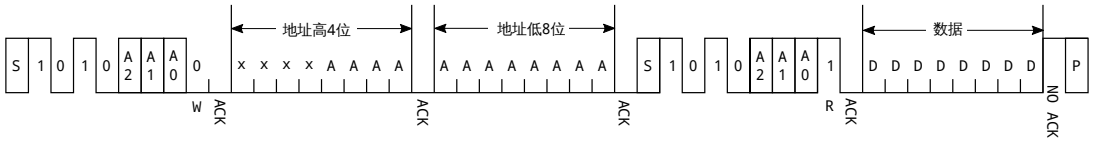


图 13.3: 24LC32 读取指定地址的一个字节

```
# i2cdetect -y 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:
10:
20: -- -- -- -- 24 -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- 34 35 36 37 -- -- --
40:
50: UU -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60:
70:
```

上面的命令检测/dev/i2c-2 总线上的设备, 发现 0x24、0x34、0x35、0x36、0x37 有效 I2C 地址。“UU” 地址被调过。

- i2cdump: 读取连接设备的 I2C 寄存器。

```
# i2cdump -y 1 0x68 i
```

注意: 错误使用`i2cdump` 会造成设备损坏, 请勿在随机地址上操作。

- i2cget: 读取 I2C/SMBus 芯片的寄存器。

```
# i2cget 1 0x68 0x20
```

上面的命令从 I2C 设备 1 的 0x68 设备 0x20 寄存器读取一个字节。

注意: 错误使用`i2cget` 会造成设备损坏。一些芯片会将读命令处理为写命令, 向寄存器写入错误的数据。

- i2cset: 向 I2C 寄存器写入数值。

```
# i2cset 1 0x68 0x20 0x50
```

上面的命令向 I2C 总线 1 的 0x68 设备地址 8 位寄存器 0x20 写入字节 0x50。

```
# i2cset -y 1 0x68 0x20 0x5000
```

上面的命令向 I2C 总线 1 的 0x68 设备地址 16 位寄存器 0x20 写入 0x5000。

注意: 错误使用`i2cset` 会造成设备损坏。错误的操作会导致数据丢失甚至更严重的后果。

- i2ctransfer: 向 I2C 设备写入一组信息。信息块大小受内核限制, 由参数`I2C_RDWR_IOCTL_MAX_MSGS` 定义。

```
# i2ctransfer 0 w1@0x50 0x64 r8
```

上面的命令对/dev/i2c-0 操作，从 I2C 的 0x50 地址 (EEPROM) 的偏移地址 0x64 读入 8 个字节。

注意: 错误使用i2ctransfer 会造成设备损坏。它同时包含读写操作, 相当于结合i2cset 和i2cget 两个命令。

清单 13.1: 通过 I2C 读取 24LC32 read_i2c.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/ioctl.h>
4  #include <fcntl.h>
5  #include <linux/i2c-dev.h>
6
7  int main (int argc, char *argv[])
8  {
9      int fd, ret;
10     char buf[128];
11
12     fd = open("/dev/i2c-0", O_RDWR);
13     if(fd < 0) {
14         perror("I2C device open failed\n");
15         return -1;
16     }
17     ret = ioctl(fd, I2C_TENBIT, 0);      /* 7bit 地址 */
18     ret = ioctl(fd, I2C_SLAVE, 0x50);    /* 从设备地址 */
19     buf[0] = 0x64;
20     write(fd, buf, 1);                   /* 写入1个字节 */
21     ret = read(fd, &buf, 4);             /* 读入4个字节 */
22     ...                                  /* 打印读入的信息 */
23
24     return 0;
25 }
```

硬件上, BeagleBone Black 的 24LC32 设置了写保护, 无法写入数据。

13.2.2 树莓派 I2C 的使用

树莓派 I2C 的内核支持选项如下:

```
Device Drivers --->
    I2C support --->
        <*> I2C device interface
        I2C Hardware Bus support --->
            <*> Broadcom BCM2835 I2C controller
```

除了内核选项支持, 还应在 Bootloader 的配置文件 config.txt 中启用 I2C 设备:

```
dtparam=i2c_arm=on
```

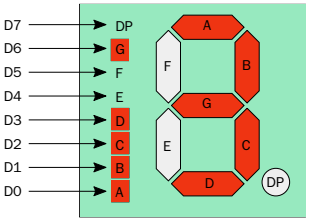
树莓派引出的 I2C 见表13.2。

表 13.2: 树莓派 I2C 引脚

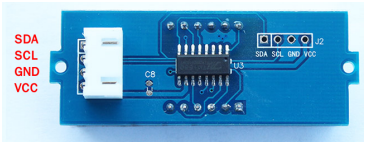
	SCL	SDA
I2C-1	pin5	pin3
I2C-0	pin28	pin27

13.2.3 使用 I2C 接口的显示器

图13.4是 TM1650 控制的四位 LED 显示器。TM1650 内部四个地址 0x34、0x35、0x36、0x37 各控制一个数码管, 另有一个控制寄存器控制数码管的工作方式, 地址是 0x24。向该地址写入的一个字节数据 D7~D0 中, D6~D4 用于亮度控制 (8 级亮度), D3 控制 7 位/8 位显示方式, D0 控制总开关 (1: 亮, 0: 灭)。字形显示为共阴极控制方式, 见右图。显示数字 “3” 的字形, 应向对应地址写入一个字节 0x4f。



(a) 正面



(b) 反面

图 13.4: 4 位 LED 数码管 (TM1650 控制器)

13.3 实验内容

1. 读取 BeagleBone Black 的 ID 信息;
2. 使用扩展接口控制多位数码管.

14

图形文件显示

计算机系统中, 表示图形的方式有两类: 一类是点阵图, 一类是矢量图。大量的图形信息以文件的形式存储。如果要将图形还原出来, 必须要对图形文件进行解析。不同的图形格式文件需要由不同的库进行处理。

14.1 BMP

BMP (**Bit**m**ap**, 位图) 图形是由 Microsoft 公司设计的一种点阵图形格式, 主要用于 Window 操作系统。大多数 BMP 图形数据按 RGB 原始格式存放, 因此也是最简单的一种图形格式。

基本 BMP 文件由三部分构成: BMP 文件头、数据信息头和数据。BMP 文件头 14 个字节, 见表14.1。

表 14.1: BMP 文件头

Offset	Bytes	Description
00H	2	ASCII 码 “BM”
02H	4	BMP 文件字节数
06H	2	保留
08H	2	保留
0AH	4	点阵数据起始位置

数据信息头描述了与设备无关的数据格式, 又称 DIB (**D**evice **I**ndependent **B**itmap) 文件格式。常见的 BMP 文件, DIB 格式由 40 个字节组成, 见表14.2, 起始偏移位置以文件头为起点。由于 BMP 头占 14 字节, 因此 DIB 从 0EH 偏移值开始。

图像高度和宽度值可以是负数, 它表示图像的伸展方向。正的宽度值表示点阵数据从左向右排列, 正的高度值表示数据从下向上伸展; 水平方向分辨率和垂直方向分辨率的单位是 pixels/meter, 正负值的含义与图像宽度/高度相同。

表 14.2: DIB 文件头

Offset	Bytes	Description
0EH	2	DIB 大小 (40 字节)
12H	4	图像宽度 (以像素为单位)
16H	4	图像高度 (以像素为单位)
1AH	2	图像面数 (总是 1)
1CH	2	每像素的位数
1EH	4	压缩方法 (0 表示不压缩)
22H	4	图像字节数
26H	4	水平方向分辨率
2AH	4	垂直方向分辨率
2EH	4	调色板色彩数量
32H	4	重要颜色的数量

如果没有调色板, 在 DIB 之后紧接着就是 RGB 数据。当图像宽度和高度值都是正数时, 数据从左下角逐层向上排列, 先行后列。当图像色彩数量很少时, 可以通过设置调色板压缩数据, 因为可以用较少的比特位为色彩编码。此时 DIB 的 1EH 地址上 (压缩方式) 可以设置不同的调色板样式。调色板紧接 DIB 之后, 调色板后面是编码的像素阵列。为了提高数据存取的效率, 一行像素数据按 32 位 (4 字节) 地址对齐, 不足时, 后面用 0 填补。

清单 14.1: BMP 文件画图 drawbmp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  #include <fcntl.h>
7
8  #define ABS(x)      ((x) > 0 ? (x): -(x))
9
10 #pragma pack(2)
11 typedef struct {
12     char         bType[2];
13     unsigned int Size;
14     short        Reserved1;
15     short        Reserved2;
16     unsigned int OffBits;
```

```
17 } BMPHeader;
18 #pragma pack()
19
20 typedef struct {
21     unsigned int Size;
22     int          Width;
23     int          Height;
24     short        Planes;
25     short        BitCount;
26     unsigned int Compression;
27     unsigned int SizeImage;
28     int          XPixelsPerMeter;
29     int          YPixelsPerMeter;
30     unsigned int ClrUsed;
31     unsigned int ClrImportant;
32 } DIBHeader;
33
34 int main(int argc, char* argv[])
35 {
36     unsigned int width, height, rowSize;
37     char *linebuffer;
38     int bmpfile = 0;
39
40     BMPHeader file_hdr;
41     DIBHeader info_hdr;
42
43     bmpfile = open("coffee.bmp", O_RDONLY);
44
45     read(bmpfile, &file_hdr, sizeof(file_hdr));
46
47     if(strncmp(file_hdr.bType, "BM", 2)) {
48         printf("File is not BMP.\n");
49         close(bmpfile);
50         return EXIT_FAILURE;
51     }
52
53     /* 读 BMP 信息头 */
54     read(bmpfile, &info_hdr, sizeof(info_hdr));
```

```
55
56      if (info_hdr.BitCount != 16 &&
57          info_hdr.BitCount != 24 &&
58          info_hdr.BitCount != 32) {
59          printf("Cannot process BMP file with bit count %d .\n",
60              info_hdr.BitCount);
61          close(bmpfile);
62          return EXIT_FAILURE;
63      }
64
65      if (info_hdr.Compression != 0) {
66          printf("This program DOES NOT process compressed file.\n");
67          close(bmpfile);
68          return EXIT_FAILURE;
69      }
70
71      width = ABS(info_hdr.Width);
72      height= ABS(info_hdr.Height);
73
74      rowSize = (info_hdr.BitCount*width + 31)/32*4;
75      linebuffer = malloc(rowSize);
76
77      for (int j = 0; j < height; j++) {
78          read(bmpfile, linebuffer, rowSize);
79          for (int i = 0; i < width; i++) {
80              /* 从 linebuffer 中提取 RGB 数据, 画点*/
81          }
82      }
83
84      close(bmpfile);
85      return EXIT_SUCCESS;
86  }
```

14.2 PNG

PNG (**P**ortable **N**etwork **G**raphics, 可移植网络图形) 是 Linux 常用的一种图形格式。开发者开发这种图形格式的最初目的是为了替代当时流行于互联网的一种私有版权协议的图像格式 GIF (**G**raphics **I**nterchange **F**ormat)。libpng 以开源版权协议发布, 它是一种无损压缩格式,

压缩算法基于 zlib 库。因此移植 libpng 需要先移植 zlib。zlib 移植过程请参考9.2.1节。

14.2.1 移植 PNG 图形库

```
libpng 主页: http://www.libpng.org
libpng 源码: https://prdownloads.sourceforge.net/libpng/libpng-1.6.37.tar.xz
```

交叉编译 libpng 过程如下:

```
在 libpng 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    CFLAGS="-I/home/student/target/usr/include" \
    LDFLAGS="-L/home/student/target/usr/lib"
$ make -j8
$ make install DESTDIR=/home/student/target
```

参数CFLAGS 和LDFLAGS 用于指导 gcc 编译器找到 zlib 的头文件和库文件。

因设置了共享库编译方式, 以上编译完成后, 应将安装目录下的共享库复制到目标系统。

14.2.2 使用 PNG 库画图

png 文件开头有 8 个固定字节: 89 50 4E 47 0D 0A 1A 0A, 可依此判断文件格式。利用 libpng 画图, 通常按以下步骤操作:

1. 创建 PNG 结构image 并初始化, 将其所有成员清零, 仅设置 image.version;
2. 调用 png_image_begin_read_from_file() 读 PNG 图像格式信息;
3. image.format 设置色彩映射表格式;
4. 调用png_image_finish_read() 读取位图数据和色彩映射表;
5. 如果是色表格式, 位图数据是色表的索引值, 需要根据索引值逐个查表, 获得每个点的 RGB 颜色值; 非色表格式, RGB 颜色值直接从位图数据中得到;
6. 根据颜色值、图像格式 (宽度和高度) 在屏幕上画出所有的像素。

清单 14.2: PNG 文件画图 drawpng.c

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
```

```
4  #include <png.h>
5
6  #include "draw.h"
7
8  int main(int argc, char *argv[])
9  {
10     png_image image;            /* png 结构 */
11     unsigned char *buffer;      /* 位图索引数据缓冲区指针 */
12     unsigned char *colormap;    /* 色表缓冲区指针 */
13     unsigned int size;
14     unsigned int x, y;
15     unsigned char r, g, b, a;
16     int ret, index;
17
18     /* 初始化 png 结构对象 */
19     bzero(&image, sizeof(image));
20     image.version = PNG_IMAGE_VERSION;
21
22     /* 命令行第一个参数是 PNG 文件名 */
23     if(argc == 2) {
24         ret = png_image_begin_read_from_file(&image, argv[1]);
25     } else {
26         printf("command: %s png-file\n", argv[0]);
27         return -1;
28     }
29
30     size = PNG_IMAGE_COLORMAP_SIZE(image);  /* 色表大小 */
31     colormap = malloc(size);
32
33     image.format |= PNG_FORMAT_RGB;         /* 设置色表格式 */
34
35     size = PNG_IMAGE_SIZE(image);           /* RGB 位图大小 */
36     buffer = malloc(size);
37
38     if(png_image_finish_read(&image,
39                              NULL,          /* 不处理Alpha */
40                              buffer,
41                              0,             /* 从上到下 */
```

```
42                              colormap) == 0) {
43          perror("Image read error\n");
44          free(colormap);
45          free(buffer);
46          return -4;
47      }
48
49      init_fb();
50      x = 20; y = 20;                         /* 画图起始点 (x, y) */
51      for (int j = 0; j < image.height; j++) {
52          for (int i = 0; i < image.width; i++) {
53              if (image.format & PNG_FORMAT_FLAG_COLORMAP) {
54                  index = *buffer++;          /* 查色表 */
55                  r = colormap[3*index + 0];
56                  g = colormap[3*index + 1];
57                  b = colormap[3*index + 2];
58              } else {
59                  r = *buffer++;              /* 读位图 */
60                  g = *buffer++;
61                  b = *buffer++;
62              }
63              dot(x + i, y + j, r, g, b);
64          }
65      }
66
67      return 0;
68  }
```

清单14.2是按上述过程完成的简化示例, 它利用6.3 设计的画图函数将 PNG 图形文件画在 Frame Buffer 上。一些 PNG 图存储的是 RGB 位图数据, 还有一些是色表格式, 需要根据不同情形分别处理。在 PC 上可以用简单的命令查看文件格式:

```
$ file tee.png
tee.png: PNG image data, 640 x 480, 8-bit/color RGB, non-interlaced
$ file coffee.png
coffee.png: PNG image data, 640 x 480, 8-bit colormap, non-interlaced
```

在 PC 上通过下面的 Makefile 完成交叉编译:

```
1  CC = arm-linux-gcc
2
```

```
3 CFLAGS = -I/home/student/target/usr/include
4 LDFLAGS = -L/home/student/target/usr/lib
5
6 drawpng: drawpng.c
7         $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS) -lpng -lz -lm -L. -ldraw
```

这里还用到了6.3节制作的 Frame Buffer 图形库 libdraw.a (或者是共享库 libdraw.so), 编译时应将它复制到当前目录, 使用选项 `-L.` 指示将当前目录加入编译器的库搜索路径。如果是共享库形式, 还应将共享库复制到目标系统的运行库搜索路径, 程序运行时依赖共享库。

14.3 JPEG

JPEG (**J**oint **P**hotographic **E**xperts **G**roup, 联合图像专家组) 是另一种常用的图形格式, 广泛用于数码相机等图像设备, 是一种有损压缩格式。它利用人眼的对图像的感知特点, 通过去除一些不敏感信息, 可获得比 PNG 格式更高的压缩率。算法基于离散余弦变换。本实验采用的图形库 libjpeg-turbo 是 libjpeg 的一个分支, 它使用 SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata, 单指令多数据) 指令对 JPEG 的编解码进行加速处理, 在 X86、Arm 等平台上可以获得比主线 libjpeg 更高的处理速度。

14.3.1 移植 JPEG 图形库

```
libjpeg-turbo 主页: https://libjpeg-turbo.org
libjpeg-turbo 源码: https://prdownloads.sourceforge.net/libjpeg-turbo/2.0.3/
   ↪ libjpeg-turbo-2.0.3.tar.gz
```

这个版本的 libjpeg-turbo 只有 cmake 配置工具, libjpeg-turbo 不依赖其他库, 编译时也不需要设置特别选项。编译过程如下:

```
在 libjpeg-turbo 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ cmake .. \
    -DCMAKE_SYSTEM_NAME=Linux \
    -DCMAKE_SYSTEM_PROCESSOR=arm \
    -DCMAKE_C_COMPILER=arm-linux-gcc \
    -DCMAKE_INSTALL_PREFIX=/usr \
    -DWITH_SIMD=ON \
    -DENABLE_STATIC=OFF
$ make -j8
$ make install DESTDIR=/home/student/target
```

14.3.2 使用 JPEG 库画图

JPEG 画图可以按下面的步骤进行:

1. 初始化 JPEG 解压结构对象, 设置错误处理例程;
2. 将 JPEG 文件与解压结构对象关联;
3. 读 JPEG 文件头, 获得图像结构参数;
4. 设置解码格式;
5. 调用函数`jpeg_start_decompress()` 开始解压;
6. 逐行读取解压数据, 逐行画点 (也可一次多行读取);
7. 调用函数`jpeg_finish_decompress()` 完成解压;
8. 释放资源。

清单14.3是一个简单的例子, 编译方法仿照 PNG 画图程序。

清单 14.3: JPEG 文件画图 drawjpeg.c

```
1  #include <stdio.h>
2  #include <setjmp.h>
3  #include <jpeglib.h>
4
5  #include "draw.h"                    /* frame buffer 库 */
6
7  struct my_error_mgr {
8      struct jpeg_error_mgr pub;     /* "public" fields */
9      jmp_buf setjmp_buffer;         /* for return to caller */
10 };
11
12 typedef struct my_error_mgr *my_error_ptr;
13
14 void my_error_exit(j_common_ptr cinfo)
15 {
16     my_error_ptr myerr = (my_error_ptr)cinfo->err;
17
18     (*cinfo->err->output_message) (cinfo);
19
20     /* 返回到 setjmp 位置 */
21     longjmp(myerr->setjmp_buffer, 1);
22 }
23
24 int main (int argc, char *argv[])
```

```
25 {
26     struct jpeg_decompress_struct cinfo;
27     FILE *infile;
28     JSAMPARRAY buffer;              /* 输出行缓冲区 */
29     int row_stride;                 /* 输出行缓冲宽度 */
30     int x, y, i, j;
31     char r, g, b;
32
33     struct my_error_mgr jerr;
34
35     if ((infile = fopen(argv[1], "rb")) == NULL) {
36         fprintf(stderr, "can't open %s\n", argv[1]);
37         return -1;
38     }
39
40     /* 设置错误处理例程 */
41     cinfo.err = jpeg_std_error(&jerr.pub);
42     jerr.pub.error_exit = my_error_exit;
43
44     if (setjmp(jerr.setjmp_buffer)) {
45         jpeg_destroy_decompress(&cinfo);
46         fclose(infile);
47         return -1;
48     }
49
50     jpeg_create_decompress(&cinfo);   /* 1. 初始化 JPEG 解压对象 */
51
52     jpeg_stdio_src(&cinfo, infile);   /* 2. 关联文件 */
53
54     jpeg_read_header(&cinfo, TRUE);   /* 3. 读文件参数 */
55
56     cinfo.out_color_space = JCS_RGB;  /* 4. 设置参数,
57                                             本程序只处理 RGB */
58     jpeg_start_decompress(&cinfo);    /* 5. 开始解压 */
59
60     /* 每次处理一行数组 */
61     row_stride = cinfo.output_width * cinfo.output_components;
62     buffer = (*cinfo.mem->alloc_sarray)
```

```
63          ((j_common_ptr)&cinfo, JPOOL_IMAGE, row_stride, 1);
64
65      init_fb();
66      x = 20; y = 20;
67      j = 0;
68
69      /* 6.  逐行读取解压数据，画图 */
70      while (cinfo.output_scanline < cinfo.output_height) {
71          jpeg_read_scanlines(&cinfo, buffer, 1);
72
73          for (i = 0; i < cinfo.output_width; i++) {
74              r = (char)buffer[0][i*3 + 0];
75              g = (char)buffer[0][i*3 + 1];
76              b = (char)buffer[0][i*3 + 2];
77              dot(x + i, y + j, r, g, b);
78          }
79          j++;
80      }
81
82      jpeg_finish_decompress(&cinfo);   /* 7. 解压完成 */
83
84      jpeg_destroy_decompress(&cinfo);  /* 8. 释放 JPEG 解压对象 */
85
86      fclose(infile);
87      return 0;
88  }
```

14.3.3 JPEG 图像处理工具

JPEG 库除了为上层应用软件提供接口函数外, 还包含一些实用工具: cjpeg/djpeg 用于将图像文件压缩成 JPEG 图像, 或将 JPEG 图像解压成 PPM、PBM、BMP 等点阵图像 (首字母 c、d 分别对应 compress 和 decompresss); rdjpgcom/wrjpgcom 用于 JPEG 文件读/写文本注释; jpegtran 用于在 JPEG 文件之间进行无损转换 (旋转、剪裁等)。

14.4 实验内容

学习图形文件的显示, 将一些不同格式的图形文件显示在屏幕上。

15

网络工具

Linux 系统提供丰富的网络功能。本实验移植下面一些常用的网络工具:

- wget, 命令行网络文件下载工具;
- openssh, 基于 Secure Shell 协议的安全网络连接应用;
- 无线网卡工具, wpa_supplicant: 无线网络设备连接配置工具; hostapd: 用户空间的无线接入守护进程, 用于管理主机的无线接入点;
- iptables, 网络数据包过滤管理工具。

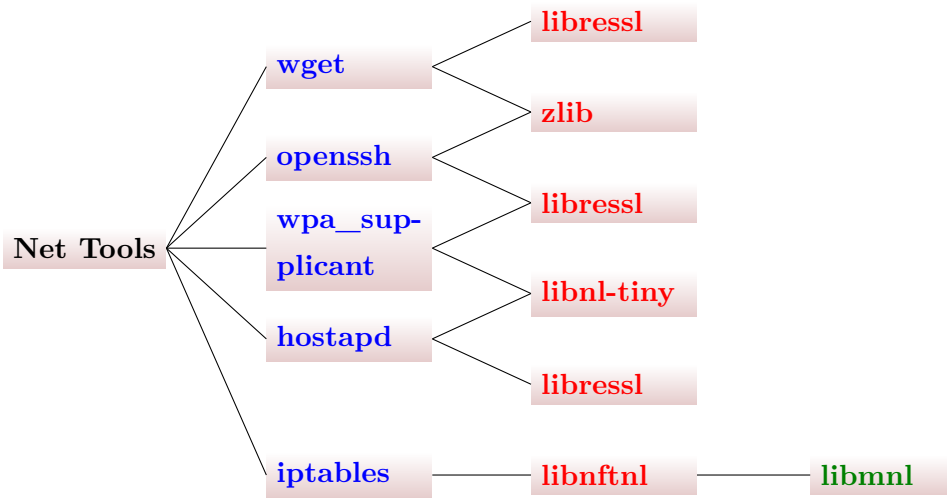


图 15.1: 常用网络工具

15.1 网络下载工具

GNU Wget 是一个下载网络文件的简单而强大的自由软件，属于 GNU 计划的一部分。软件名称取自 “World Wide Web” 和 “get” 的结合, 目前支持 HTTP、HTTPS、FTP 这三个最常见的 TCP/IP 协议, 是 Linux 系统中常用的命令行网络文件下载工具。在数据加密传输时, 加密

算法可通过 LibreSSL、OpenSSL 或 GNUTLS 库支持。有关 OpenSSL/LibreSSL 的情况请参考12.2.1节。BusyBox 中有一个简化版的 wget, 因为没有 SSL 库支持, 不支持 HTTPS 协议。

```
wget 主页: https://www.gnu.org/software/wget/
wget 源码: https://ftp.gnu.org/gnu/wget/wget-1.21.tar.gz
```

Wget 依赖 zlib 的压缩功能和 SSL 库提供的网络安全功能。编译 zlib 请参见 9.2.1节，编译 OpenSSL/LibressL 请参见12.2.1节。编译 Wget 过程如下:

```
在 wget 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-pcre2 \
    --with-ssl=openssl \
    --without-libuuid
$ make -j8
$ make install DESTDIR=/home/student/target
```

15.2 远程网络连接

OpenSSH, 基于 Secure Shell 协议的安全网络连接应用, 它包含 SSH 的服务器和客户端、文件传输及密钥管理等功能。OpenSSH 依赖 zlib 和 LibreSSL 或 OpenSSL。

```
openssh 主页: https://www.openssh.com/
openssh 源码: http://ftp.openbsd.org/pub/OpenBSD/OpenSSH/portable/openssh-7.5
   ↪ p1.tar.gz
```

15.2.1 编译 openssh

根据 PC 上的路径习惯, 使用选项`--sysconfdir` 按下面的方式配置 OpenSSH。因为缺省方式会将 SSH 的配置文件保存在 FHS 的二级目录结构 /usr/etc/ssh/ 中。

```
在 openssh 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --sysconfdir=/etc/ssh \
    --disable-strip
$ make -j8
```

```
$ make install-nokeys DESTDIR=/home/student/target
```

安装 OpenSSH 时会执行 strip 操作, 而 X86 的 strip 工具是不能正确识别 Arm 指令的。`--disable-strip` 避免在安装时 strip。安装后可以在安装目录下手工用 arm-linux-strip 处理。此外, 正常的安装过程会调用编译的程序检查配置文件, 显然交叉编译的程序是不能在 PC 上运行的, 因此使用了 “install-nokeys” 的安装方式。

15.2.2 配置 SSH 服务

根据 OpenSSH 编译配置选项, SSH 服务配置文件的目录是 /etc/ssh, 其下的 sshd_config 是服务器配置文件, ssh_config 是客户端配置文件。客户端配置文件保持不变。在目标系统上的服务器配置文件中, 去掉如下几行的注释符 “#”, 修改成:

```
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
X11Forwarding yes
PermitRootLogin yes
```

上面的参数指定密钥文件、允许 X11 转发、允许 root 登录。通常为了安全起见, 在 PC 上是不允许 root 直接远程登录的。在嵌入式系统上, 为了允许 root 登录, 应将 PermitRootLogin 这个选项设为 “yes”。X11Forwarding 允许客户端的 X 窗口延伸到服务器, 利用本地 X 服务器的图形界面执行远程图形程序。

根据配置文件设置的密钥目录, 用下面两条命令生成密钥文件:

```
# ssh-keygen -q -t rsa -f /etc/ssh/ssh_host_rsa_key
# ssh-keygen -q -t dsa -f /etc/ssh/ssh_host_dsa_key
```

上述命令执行时会要求输入 passphrase, 可直接输入回车忽略。

为支持 X11 转发, 还应在 SSH 服务器端创建回环设备 lo:

```
# ifconfig lo 127.0.0.1
```

以上工作准备就绪, 可以启动 SSH 服务了:

```
# /usr/sbin/sshd
```

在客户端 (PC) 尝试连接嵌入式目标板。假设目标板的 IP 地址是 192.168.2.123:

```
$ ssh root@192.168.2.123
```

15.3 无线网络配置工具

无线网卡有两种主要的工作模式: 客户端模式和接入点模式 (**A**ccess **P**oint, AP 模式, 又称热点模式)。前者, 网卡作为一个从设备, 连接到附件的无线路由器上, 通过路由器接入互联网; 后者将自身设置为一个无线路由器, 为附近的设备提供接入互联网功能。实现这些功能的是用户空间的两个软件 wpa_supplicant 和 hostapd:

- wpa_supplicant, 受保护的 WiFi 接入 (**W**iFi **P**rotected **A**ccess) 请求, Linux 系统中常用的无线接入管理工具。
- hostapd, 用户空间的无线接入守护进程 (**A**ccess **P**oint **D**aemon), 用于管理主机的无线接入点。

hostapd 和 wpa_supplicant 的加密算法基于 OpenSSL/LibreSSL 库。这两个软件的代码风格完全一样, 内容也有很大部分重叠, 编译方法大同小异。二者都依赖一个网络链路层协议库 libnl。libnl 来自 OpenWrt (一个基于嵌入式 Linux 的无线路由器操作系统) 项目。此处使用该库的一个小型化替代版本 libnl-tiny。

```
libnl-tiny 主页: https://wiki.openwrt.org/doc/techref/libnl
libnl-tiny 源码: http://ftp.barfooze.de/pub/sabotage/tarballs/libnl-tiny
    ↪ -1.0.1.tar.xz
hostapd 及 wpa_supplicant 主页: http://w1.fi
hostapd 源码: http://w1.fi/releases/hostapd-2.8.tar.gz
wpa_supplicant 源码: http://w1.fi/releases/wpa_supplicant-2.8.tar.gz
```

15.3.1 编译

libnl-tiny

libnl-tiny 不提供编译配置工具, 因此在使用 Makefile 编译时需要手工指定编译器, 用于替代 GNU Make 默认的 CC 变量:

```
在 libnl-tiny 解压目录下执行:
$ rm -rf include/linux
$ make prefix=/usr \
       CC=arm-linux-gcc \
       STATICLIB=""
$ make prefix=/usr \
       CC=arm-linux-gcc \
       STATICLIB="" \
       DESTDIR=/home/student/target
       install
```

以上不编译静态库, 只编译共享库, 同时供 wpa_supplicant 和 hostapd 使用。

hostapd

hostapd 没有自动配置工具, 内部预制了一个配置文件 `defconfig`。编译时使用 `.config`。交叉编译时只要做个编译器替换, 再将 `defconfig` 复制到`.config`, 原则上就可以编译了。但由于用 libnl-tiny 代替了 libnl, 还需要修改`.config` 文件中定义的依赖库。即, 将`CONFIG_LIBNL32` 这一行注释掉。

```
在 hostapd 解压目录下执行:
$ cd hostapd
$ CFLAGS="-D_GNU_SOURCE \
    -I/home/devel/target/usr/include \
    -I/home/devel/target/usr/include/libnl-tiny"
$ cp defconfig .config
$ (修改配置文件 .config)
$ make \
     CC=arm-linux-gcc \
     CONFIG_LIBNL20=y \
     CONFIG_LIBNL_TINY=y \
     LDFLAGS="-L/home/devel/target/usr/lib -lnl-tiny"
$ make install DESTDIR=/home/student/target \
     CC=arm-linux-gcc \
     CONFIG_LIBNL20=y \
     CONFIG_LIBNL_TINY=y \
     LDFLAGS="-L/home/devel/target/usr/lib -lnl-tiny" \
     BINDIR=/usr/bin
```

wpa_supplicant

编译 wpa_supplicant 与编译 hostapd 几乎完全一样, 只是由于 wpa_supplicant 默认还依赖 DBus 库, 为了避免在此时编译 DBus, 应将配置文件.config 中的 `CONFIG_CTRL_IFACE_DBUS_NEW` 和`CONFIG_CTRL_IFACE_DBUS_INTRO` 两行注释掉。

15.3.2 无线网络连接

带有板载无线网卡的单板机 (如树莓派 3 代), 通过内核的无线协议支持, 可以进行无线网络连接。没有板载无线网卡 (如 BeagleBone Black), 通过外接 USB 无线网卡, 以及内核的驱动支持, 同样可以实现 WiFi 连接。

为了使用无线网络, 需要在编译内核时选择对应型号的设备支持。下面是树莓派 3 代无线网卡和 rtL8192cu USB 网卡支持项在内核中的位置。rtl8192cu USB 网卡可以在 BeagleBoard 上使用。

```
Device Drivers  --->
    [*]   Network device support  --->
        [*]   Wireless LAN  --->
            [*]  Broadcom devices
            ...
            <M>  Broadcom FullMAC WLAN driver
```

```
            [*]  SDIO bus interface support for FullMAC driver
            [*]  USB bus interface support for FullMAC driver
            [ ]  PCIE bus interface support for FullMAC driver
            [ ]  Broadcom device tracing
            ...
            [*]  Realtek devices
            < >  Realtek 8180/8185/8187SE PCI support
            < >  Realtek 8187 and 8187B USB support
            <*>  Realtek rtlwifi family of devices  --->
                <*> Realtek RTL8192CU/RTL8188CU USB Wireless ...
```

以模块方式编译的驱动, 启动后需要通过手工加载。例如, 上面针对树莓派的 FullMAC WLAN 是以模块方式编译的, 生成 brcmfmac.ko 文件。内核及模块编译完成后, 所有模块需要复制到目标系统的 /lib/modules/ 目录下, 并保持原有的目录结构。用户通过下面的命令安装驱动:

```
# modprobe brcmfmac
```

驱动、模块之间会存在依赖关系, `modprobe` 命令可以根据内核结构自动安装有依赖的模块。

为了确定无线网卡是否已工作, 可以用 `ifconfig`查看 wlan0 设备:

```
# ifconfig -a
...
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.124  netmask 255.255.255.0  broadcast 192....
        ether 30:10:b3:99:c3:10  txqueuelen 1000
        RX packets 76724  bytes 53485912 (53.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 68102  bytes 53748267 (53.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

当可以显示 wlan0 信息时, 无论是否有 IP 地址, 都表明设备已驱动。没有 IP 地址只是表示还没有正确配置。如果不带`-a` 选项时看不到 wlan0 信息, 说明无线网卡被禁用了, 此时应通过 BusyBox 提供的 `rfkill` 命令打开射频设备:

```
# rfkill unblock wlan
```

如果插入网卡后不工作, 注意检查一下内核最后打印出的信息:

```
# dmesg | tail
...
rtl8192cu: Loading firmware rtlwifi/rtl8192cufw_TMSC.bin
usb 1-1: Direct firmware load for rtlwifi/rtl8192cufw__TMSC.bin failed with
    ↪ error -2
usb 1-1: Direct firmware load for rtlwifi/rtl8192cufw.bin failed with error -2
```

```
rtlwifi: Loading alternative firmware rtlwifi/rtl8192cufw.bin
rtlwifi: Selected firmware is not available
```

一些网卡需要额外的固件。固件不属于开源软件部分, 需要通过网卡厂商提供的资源下载。PC 桌面 Linux 发行版通常都带有这些固件, 它们一般放在 /lib/firmware/ 目录下, 与这里移植的嵌入式 Linux 的目录要求是一致。目标系统上使用这些固件, 也需要把文件放在这个目录里。

启用 wpa_supplicant 之前, 还要建立一个 WPA 请求的配置文件, 用来描述接入点 ID 和连接方式:

清单 15.1: WPA 请求配置文件 /etc/wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant

ctrl_interface_group=0
ap_scan=1
update_config=1
fast_reauth=1

network={
    ssid="NJU_WLAN"
    psk="password"
}
```

通过下面的命令接入 “NJU_WLAN” 热点:

```
# wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant.conf
# udhcpc -i wlan0
```

wlan0 是无线网卡被识别的设备名, wpa_supplicant 的选项 `-B` 表示后台运行。udhcpc 是来自 BusyBox 的 DHCP 客户端命令, 它从 DHCP 服务器获得分配的 IP 地址。由于它只能获得地址, 不能设置网卡地址, BusyBox提供了一个自动配置的 脚本文件模板 simple.script, 它在 BusyBox 源码的 examples/udhcp/ 目录下。使用时将其复制到目标系统的 /usr/share/udhcpc/ 目录, 重新命名为 default.script, 并用 `chmod +x` 命令给它加上可执行属性。

一旦 wlan0 设置了正确的 IP 地址, 便可以通过接入点 NJU_WLAN 连接因特网了。

15.3.3 WiFi 热点方式

无线网卡还有另一种工作方式, 使用 hostapd 将其设为热点, 供其他设备访问。hostapd 的配置文件如下:

清单 15.2: 接入点配置文件 /etc/hostapd.conf

```
interface=wlan0
driver=nl80211
```

```
ssid=BBBlack
hw_mode=g
channel=6
auth_algs=1
wmm_enabled=0
wpa=2
wpa_passphrase=password
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
```

启用 AP 需要下面三个步骤:

1. 设置本机 IP 地址:

   ```
   # ifconfig wlan0 192.168.0.1
   ```

2. 启动主机 AP 守护进程:

   ```
   # hostapd -B /etc/hostapd.conf
   ```

3. 开启 DHCP 服务, 为接入设备自动分配地址:

   ```
   # udhcpd -f /etc/udhcpd.conf
   ```

   选项 "-f" 表示前台运行。udhcpd.conf 是 DHCP 默认的配置文件, 该项参数也可以省略, 文件内容可以参考 BusyBox 中的文件 examples/udhcp/udhcpd.conf。清单15.3 是一个简化的版本。

   清单 15.3: DHCP 配置文件 /etc/udhcpd.conf

   ```
   start 192.168.0.20          #
   end   192.168.0.254         # 可分配 IP 地址范围
   interface wlan0             #
   remaining yes
   opt dns 8.8.8.8 4.4.4.4     # 域名服务器 (可选项)
   opt subnet 255.255.255.0    # 子网掩码 (可选项)
   opt router 192.168.0.1      # 路由地址 (可选项)
   opt lease 864000            # 占用时间 (秒数, 可选项)
   ```

   udhcpd 需要用文件/var/lib/misc/udhcpd.leases 记录数据, 需要先创建这个空文件, 否则会启动失败。创建方法如下:

   ```
   # mkdir -p /var/lib/misc
   # touch /var/lib/misc/udhcpd.leases
   ```

以上工作完成后, 使用其他无线设备 (手机、平板或带有 WiFi 的笔记本电脑), 应可以看到一个名为 “BBBlack” 的 WiFi 信号。接入, 输入密码 “password” 后, 便可以通过网络命令 (笔记本电脑的 `telnet`、`ssh`等命令) 连接到目标板。

15.4 防火墙配置工具

网络防火墙 (Firewall) 是指位于内部网和外部网之间的屏障, 它由硬件和软件两部分共同组成。软件部分又由内核和用户程序配合, 按照预设规则, 控制网络数据包的进出。内核部分由模块 NetFilter 负责, iptables 则是用户空间常用的管理配置工具。iptables 是一个面向系统管理员的命令行工具, 它允许系统管理员定义处理数据包规则链路的表项, 可以检测、修改、转发、重定向或丢弃数据包。libmnl 和 libnftnl 为 iptables 提供底层用户空间的 API 函数。

15.4.1 移植过程

软件资源如下:

```
libmnl 主页: https://netfilter.org/projects/libmnl
libmnl 源码: https://netfilter.org/projects/libmnl/files/libmnl-1.0.4.tar.bz2
libnftnl 主页: https://netfilter.org/projects/libnftnl
libnftnl 源码: https://netfilter.org/projects/libnftnl/files/libnftnl-1.1.5.
    ↪ tar.bz2
iptables 主页: https://netfilter.org/projects/iptables
iptables 源码: https://netfilter.org/projects/iptables/files/iptables-1.8.4.
    ↪ tar.bz2
```

这三个软件编译方法大体相似, 但层层依赖, 上层软件配置编译环境时需要传递下层库的路径。下面是编译过程:

```
编译 libmnl. 在 libmnl 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target

编译 libnftnl. 在 libnftnl 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
```

```
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    LIBMNL_CFLAGS="-I/home/student/target/usr/include" \
    LIBMNL_LIBS="-L/home/student/target/usr/lib -lmnl"
$ make -j8
$ make install DESTDIR=/home/student/target

编译 itpables. 在 iptables 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    --disable-ipv6 \
    libmnl_CFLAGS="-I/home/student/target/usr/include" \
    libmnl_LIBS="-L/home/student/target/usr/lib -lmnl" \
    libnftnl_CFLAGS="-I/home/student/target/usr/include" \
    libnftnl_LIBS="-L/home/student/target/usr/lib -lnftnl"
$ make -j8
$ make install DESTDIR=/home/student/target
```

iptables 配置过程中的 `--disable-ipv6` 是一个可选项, 只是为了缩减库的规模。

以上工作完成后, 将/home/student/target/ 目录平移至目标系统的根目录, 保持目录结构。即, /home/student/target/etc/ 下面的内容移至目标系统的 /etc/ 目录下, /home/student/target/usr/bin/ 下的内容移至目标系统的/usr/bin/ 目录下, /home/student/target/usr/lib/ 目录下的共享库移至目标系统的 /usr/lib/ 目录下, iptables 还有一个插件目录 usr/lib/xtables/ 也需要移到目标系统。为了减小存储占用的空间, 可以使用交叉编译工具的 strip 命令对二进制文件进行处理。静态库 (.a 文件) 及.la 文件、share 目录下的数据文件、用于二次开发的 include 目录不需要搬到目标系统。

15.4.2 无线路由器

有线网接入局域网, 无线网卡工作在 AP 模式, 经过适当的设置, 就可以作为无线路由器使用。下面是使用 iptables 设置网络地址转换 (**N**etwork **A**ddress **T**ranslate, NAT) 规则的过程。

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

```
# iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
# iptables -A FORWARD -i wlan0 -o eth0 -j ACCEPT
# iptables -I INPUT -p udp --dport 67 -i eth0 -j ACCEPT
# iptables -I INPUT -p udp --dport 53 -s 192.168.0.0/24 -j ACCEPT
# iptables -I INPUT -p tcp --dport 53 -s 192.168.0.0/24 -j ACCEPT
```

设置输入接口 wlan0, 输出接口 eth0, 转发 DHCP (67 号端口) 和域名服务系统 DNS (**D**omain **N**ame **S**ystem, 53 号端口)。通过以上设置, 内核将对 wlan0 设备的访问转换成对 eth0 的访问。只要 eth0 能访问外网, 通过 wlan0 登录的设备也可以访问外网。

搭建无线路由器还可以使用桥接的方法, 使用 BusyBox 提供的 brctl。以下是操作过程:

```
# udhcpc -i eth0                    # DHCP 方式配置 eth0 地址
# brctl addbr br0                   # br0 是桥的名字, 任意
# ifconfig br0 XXX.XXX.XXX.XXX      # 配置与 eth0 相同的IP地址
# brctl addif br0 eth0
# hostapd -B /etc/hostapd.conf      # 启动 wlan0 hostap 模式
# brctl addif br0 wlan0
```

15.5 实验内容

1. 在目标系统上移植 Wget, 利用有线网络接口尝试在目标系统上下载网络文件;
2. 在目标系统上安装 SSH 服务, 尝试从 PC 通过 SSH 访问目标系统;
3. 移植无线网络工具 wpa_supplicant, 拔除网线, 让目标系统通过无线网络接口访问互联网 (这部分实验需要实验室提供无线 WiFi 接入点);
4. 移植 hostapd 并配置接入方法, 使用移动设备连接目标系统。
5. 移植 iptables, 配置数据包的转发规则, 让目标系统成为一个无线路由器。

16

文字布局与显示

计算机显示输出主要有图形和文字两方面。计算机以二进制存储和处理信息, 文字信息也是按二进制保存和处理的。将存储在文件中的文字内容展现在计算机显示器上, 其硬件基础是显示设备, 软件则包含从文字编码、字库检索、字形构造、文本布局等多个层次。

本实验移植文本处理相关软件。

16.1 计算机文字显示

在通用计算机上显示文字, 从上到下一般要经过如下几个层级:

1. 文字编码: 通过标准化的编码方案存储文字信息, 例如英文的 ASCII 编码、国际化文字的 UTF-8 编码等等。UTF-8 是 Unicode 的一种存储格式, 兼容 ASCII 码;
2. 字体库: 存储相应编码文字的字形。在通用计算机上, 就是所谓的字体文件。软件以编码作为索引, 在字体库中找到对应的字形, 向显示设备输出;
3. 输出设备: 显示器或打印机等等能展现字形的设备。

嵌入式计算机上, 考虑到处理能力与存储空间的限制, 实现显示的过程可能不像在通用计算机上那么完善, 但大体过程如此。不同文字系统可能会有不同的处理方案, 在一些应用场合会产生偏差。例如早期的打印机本身带有字体库, 计算机显示屏上看到的字形可能与打印的结果不完全一致。

目前计算机系统的字体格式主要有两种: 点阵字体和矢量字体。点阵字体是通过点阵描述字形的一种格式。在确定字号大小的情况, 实现显示的过程相对简单, 但缩放比较困难, 除非是整倍数缩放、且不计较显示效果。清单16.1是一个显示点阵字形的小程序。在图形界面中, 将打印函数 `printf()` 换成画点的函数, 就可以在图形界面中 “画” 出字形。字库来自清单16.2 中存储的 ASCII 字形, 它是手工构造的 8×16 的点阵。将点阵数据保存到一个文件, 这个文件就是所谓的字库, 以 ASCII 码为索引。

清单 16.1: 点阵字形显示 display_char.c

```
1 #include <stdio.h>
```

```
2  #include "font_8x16.h"
3
4  int main (int argc, char *argv[])
5  {
6      int index = 'T';
7      char c;
8
9      for (int i = 0; i < 16; i++) {
10         c = font[index][i];
11         for (int j = 0; j < 8; j++) {
12             if (c & 0x80)
13                 printf("*");
14             else
15                 printf(" ");
16             c <<= 1;
17         }
18         printf("\n");
19     }
20     return 0;
21 }
```

清单 16.2: 点阵字形 font.h

```
1  #ifndef _FONT_H
2  #define _FONT_H
3
4  unsigned char font[128][16] = {
5      ...
6      /* ASCII '0', index 0x30 */
7      0b00000000,     /*          */
8      0b00000000,     /*          */
9      0b00111000,     /*   ooo    */
10     0b01101100,     /*  oo oo   */
11     0b11000110,     /* oo   oo  */
12     0b11000110,     /* oo   oo  */
13     0b11010110,     /* oo o oo  */
14     0b11010110,     /* oo o oo  */
15     0b11000110,     /* oo   oo  */
16     0b11000110,     /* oo   oo  */
```

```
    0b01101100,     /*  oo oo   */
    0b00111000,     /*   ooo    */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00000000,     /*          */

    /* ASCII '1', index 0x31 */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00011000,     /*    oo    */
    0b00111000,     /*   ooo    */
    0b01111000,     /*  oooo    */
    0b00011000,     /*    oo    */
    0b00011000,     /*    oo    */
    0b00011000,     /*    oo    */
    0b00011000,     /*    oo    */
    0b00011000,     /*    oo    */
    0b00011000,     /*    oo    */
    0b01111110,     /*  oooooo  */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00000000,     /*          */
    ...

    /* ASCII 'A', index 0x41 */
    0b00000000,     /*          */
    0b00000000,     /*          */
    0b00010000,     /*    o     */
    0b00111000,     /*   ooo    */
    0b01101100,     /*  oo oo   */
    0b11000110,     /* oo   oo  */
    0b11000110,     /* oo   oo  */
    0b11111110,     /* ooooooo  */
    0b11000110,     /* oo   oo  */
    0b11000110,     /* oo   oo  */
    0b11000110,     /* oo   oo  */
```

```
55      0b11000110,     /* oo    oo   */
56      0b00000000,     /*            */
57      0b00000000,     /*            */
58      0b00000000,     /*            */
59      0b00000000,     /*            */
60      ...
61  };
62  #endif      /* _FONT_H */
```

现代计算机大量使用了矢量字体。矢量字库中存储的不是字符的点阵, 而是字符的笔画或轮廓曲线参数 (图16.1)。无论字形几何尺寸大小如何变化, 对于同一种字体, 描述字形的数据量是确定的。字形参数可以采用数学工具进行准确的缩放和变换, 实现任意精度的显示。

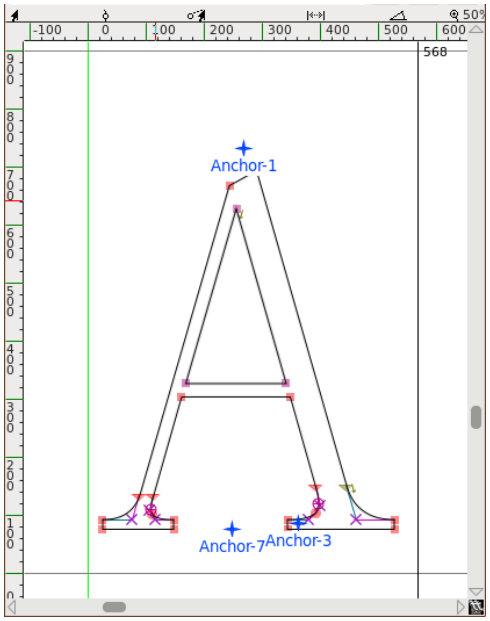


图 16.1: 文鼎明体字符 “A” 的字形描述

本实验为 Linux 系统增加文字显示功能, 目标是移植文本布局引擎 Pango。图16.2是构成 Pango 的一种层次依赖关系。虽然这套系统也支持点阵字形, 但主要面对的仍然是矢量字形显示。

16.2 编译环境

软件移植过程中的配置工作, 是检查开发环境、建立依赖关系的过程。虽然有了自动化配置脚本 (auto configure 中的 configure 命令、cmake 中的 CMakeList.txt 文件等等), 仍然需要开发人员做一些基础的引导工作。本实验需要移植的软件层次多、依赖关系复杂, 采用像15.4节那样在 configure 命令中设置参数的方法, 随着系统规模的增加, 软件依赖越来越多, 会让配置工作极其繁琐。而通过适当的环境变量设置, 配置工具会自动检查依赖关系、生成 Makefile。下面是

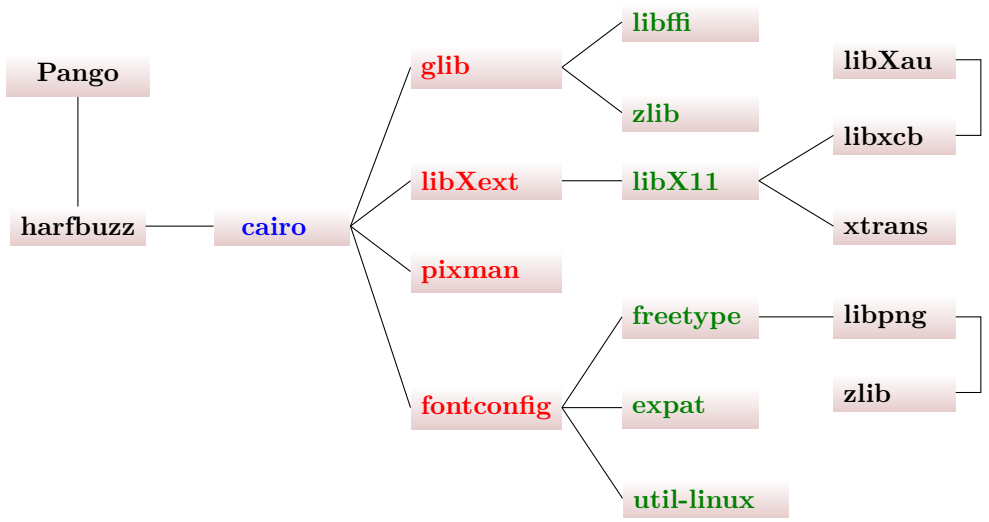


图 16.2: Pango 软件层次关系

一些重要的环境变量 (假设所有软件编译后集中安装到/home/student/target):

```
$ INSTALL_PATH=/home/student/target
$ export CFLAGS="-I$INSTALL_PATH/usr/include"
$ export CPPFLAGS="-I$INSTALL_PATH/usr/include"
$ export CXXFLAGS="-I$INSTALL_PATH/usr/include"
$ export LDFLAGS="-L$INSTALL_PATH/usr/lib"
$ export LDFLAGS="$LDFLAGS -Wl,-rpath-link=$INSTALL_PATH/usr/lib"
$ export PKG_CONFIG="/usr/bin/pkg-config"
$ export PKG_CONFIG_PATH="$INSTALL_PATH/usr/lib/pkgconfig:$INSTALL_PATH/usr/
    ↪ share/pkgconfig"
$ export PKG_CONFIG_LIBDIR="$PKG_CONFIG_PATH"
$ export PKG_CONFIG_SYSROOT_DIR="$INSTALL_PATH"
```

CFLAGS 是 C 语言头文件路径; CXXFLAGS 是 C++ 语言头文件路径, 使用 g++ 编译时会按这个路径查找头文件; CPPFLAGS 是 C/C++ 语言预处理头文件路径, 配置环境时常常会用 gcc 的选项 -E 测试某些功能是否可用; LDFLAGS 通常是库的路径, 对于间接的动态链接 (未显式出现在-l 选项中的库), 有时候还需要向链接器传递 -rpath-link 路径; 有些软件安装时, 头文件和库的路径会安装在环境变量 CFLAGS 和 LDFLAGS 设置之外的路径中, PKG_CONFIG_* 一组变量利用软件生成的.pc 文件帮助配置工具补充依赖的头文件和库文件路径。

软件安装后, 库目录中除了二次开发所需要的库以外, 还会有一些.la 文件。在多层依赖关系中, 这些文件很重要。上层软件编译时, 需要它来找到依赖库的链接路径, 并且根据它建立新的链接关系, 更上层的软件编译时通过它间接链接到下层库。例如, libxcb 依赖 libXau, 需要把 libXau.la 的路径加入 libxcb.la, 而 libX11 依赖 libcxb, 并通过 libxcb.la 找到 libXau.la。

为了能够建立完整的链条, 需要根据移植过程中设置的安装路径修改.la 文件。例如, 对于 libxcb 来说, 在安装 libxcb 后, 应将 lib/libxcb.la 的如下几行:

```
# Libraries that this one depends upon.
dependency_libs=' -L/home/student/target/usr/lib /usr/lib/libXau.la'
...
# Directory that this library needs to be installed in:
libdir='/usr/lib'
```

修改为:

```
# Libraries that this one depends upon.
dependency_libs=' -L/home/student/target/usr/lib /home/student/target/usr/lib/
    ↪ libXau.la'
...
# Directory that this library needs to be installed in:
libdir='/home/student/target/usr/lib'
```

否则, 在编译 libX11 或其他依赖 libxcb 的软件时会误以为 libXau.la 在 /usr/lib 中。其他的.la 文件也应照此处理。这类文件比较多, 建议编写一个脚本进行批量修改。

软件层次关系不是一成不变的。很多软件的依赖关系可以通过配置项改变。当配置项缺省时, 配置工具会根据当前环境自动检测依赖。因此, 在移植具有多层次关系的软件时, 如果不明确指定配置选项, 前后两次编译的结果有可能不同。

16.3 二维矢量图形库 Cairo

Cairo 项目源自 X Window 系统的 Xr/Xc, 软件名称来自 “Xr” 对应的两个希腊字母 χ、ρ 的发音。除了支持图像缓存以外, 还可以直接输出 PS、PDF 和 SVG 格式的矢量图形文件。根据图16.2给出的依赖关系看出, cairo 库由 glib、X Window 库、像素处理函数库 pixman 和字体配置工具四部分组成。

16.3.1 glib

glib 是从 GTK 分离出来的、与图形接口无关的部分, 它依赖外部函数高级语言调用接口 libffi (**F**oreign **F**unction **I**nterface, FFI)。与 GTK 相关的项目基本上都依赖 glib。注意不要将 glib 与 Glibc 混淆, Glibc 来自 GNU 项目。

```
libffi 主页: https://sourceware.org/libffi
libffi 源码: https://github.com/libffi/libffi/releases/download/v3.3/libffi
    ↪ -3.3.tar.gz
glib 主页: https://www.gtk.org
glib 源码: https://download.gnome.org/sources/glib/2.53/glib-2.53.4.tar.xz
```

编译 libffi

```
在 libffi 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    --disable-multi-os-directory
$ make -j8
$ make install DESTDIR=/home/student/target
```

移植 64 位系统 libffi 时 (`--host` 指定 aarch64-linux), 库文件会被缺省地复制到 usr/lib64/ 目录而不是 usr/lib/ 目录, 这给上层软件编译带来了一点麻烦, 因为需要为环境变量`LDFLAGS` 多增加一个目录项。配置选项 `--disable-multi-os-directory` 避免出现这种目录结构。在向 32 位系统移植时, 这个选项没有影响。

编译 glib

glib 依赖 pcre 库[1] 和 zlib。由于 glib 源码内部自带 pcre, 并且本实验移植的其他软件不依赖 pcre, 可以用 `--with-pcre=internal` 指定其内部依赖, 无需专门移植。

编译 zlib 请参考9.2.1节。下面是编译 glib 的过程。配置 glib 时, 一些环境检查在交叉编译时不能正常进行, 因此在配置选项中, 通过预设参数避开这些检查。

```
在 glib 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    glib_cv_stack_grows=no \
    glib_cv_uscore=no \
    glib_cv_va_val_copy=no \
    ac_cv_func_mmap_fixed_mapped=yes \
    ac_cv_func_posix_getpwuid_r=yes \
    ac_cv_func_posix_getgrgid_r=yes \
    ac_cv_func_snprintf_c99=yes \
    ac_cv_func_vsnprintf_c99=yes \
```

[1]pcre: **P**erl-**C**ompatible **R**egular **E**xpression (Perl 兼容的正则表达式)。Perl 是一种解释型编程语言, 正则表达式是用特定的字符描述文字特征的一种方法, 常用于文字检索、替换。

```
    --disable-libmount \
    --with-pcre=internal
$ make -j8
$ make install DESTDIR=/home/student/target
```

configure 过程中, 有些特性可以通过预编译测试获得, 有些参数需要通过程序运行得到。例如为了测试 stack_grows (堆栈增长), 配置工具会自动创建下面一段小程序:

```
volatile int *a = 0, *b = 0;
void f (int i)
{
    volatile int x = 5;
    if (i == 0) b = &x;
    else f (i - 1);
}
int main ()
{
    volatile int y = 7;
    a = &y; f (100);
    return b > a ? 0 : 1;
}
```

根据程序运行的返回值确定这项参数。但交叉编译的程序无法在主机上运行, 因此这类特性只能预设参数给交叉编译的配置工具。原则上, 应将交叉编译的程序在目标系统上运行, 根据运行结果设置这些参数。由于主机和目标机同属 Linux, 所以大多数这类特性都可以根据主机平台的结果设置, 只有极少数由内核导致的差异 (如系统功能调用) 可能失配。

16.3.2 X Window 系统

X Window 源自 MIT 的 Athena 项目, 其中的字母 “X” 在字母表中排在 “W” 之后, 意喻下一代 Window 系统。协议目前到第 11 版, 由 X.Org 基金会维护 (https://www.x.org)。

```
X Window 主页: https://www.x.org
部分X Window 库源码:
https://www.x.org/releases/individual/proto/xorgproto-2020.1.tar.gz
https://www.x.org/releases/individual/lib/libXau-1.0.9.tar.gz
https://www.x.org/releases/individual/xcb/xcb-proto-1.14.tar.gz
https://www.x.org/releases/individual/xcb/libxcb-1.14.tar.gz
https://www.x.org/releases/individual/lib/xtrans-1.4.0.tar.gz
https://www.x.org/releases/individual/lib/libX11-1.6.9.tar.gz
https://www.x.org/releases/individual/lib/libXext-1.3.4.tar.gz
```

编译 X Window 库需要先安装 xorgproto 和 xcb-proto, 它们是 Xlib 和 libxcb 的头文件。

```
在 xorgproto 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
在 xcb-proto 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

除了 libxcb, X Window 系统中库的编译方法都差不多。一些 X 库需要在配置时增加一个选项`--enable-malloc0returnsnull`, 以避开交叉编译检查。大多数 X 库都依赖 libX11, 因此编译 X Window 系统应先编译出 libX11。假设环境变量 `INSTALL_PATH` 如前所设, 以下是编译 libX11 的步骤:

1. 编译 libXau:

   ```
   在 libXau 解压目录下执行:
   $ mkdir build_arm && cd build_arm
   $ ../configure \
      --host=arm-linux \
      --prefix=/usr \
      --enable-shared \
      --disable-static
   $ make -j8
   $ make install DESTDIR=$INSTALL_PATH
   ```

2. 编译 libxcb:

   libxcb 需要用到主机的 Python 工具, Python 根据变量 `pythondir` 和 `xcbincludedir` 查找 xcb-proto 安装的文件。设置了正确的环境变量 `PKG_CONFIG_SYSROOT_DIR`, 配置工具会根据 xcb-proto 安装的.pc 文件自动找到, 否则应按下面的方法设置 `PKG_CONFIG`:

   ```
   $ PYTHON_TOOL=$INSTALL_PATH/usr/lib/python2.7/dist-packages
   $ PKG_CONFIG="/usr/bin/pkg-config"
   $ PKG_CONFIG="$PKG_CONFIG --define-variable=pythondir=$PYTHON_TOOL"
   ```

```
$ PKG_CONFIG="$PKG_CONFIG --define-variable=xcbincludedir=$INSTALL_PATH/
    ↪ usr/share/xcb"
```

编译 libxcb 过程如下:

```
在 libxcb 解压目录下执行:
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

3. 编译 xtrans:

```
在 xtrans 解压目录下执行:
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

4. 编译 libX11:

```
在 libX11 解压目录下执行:
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static \
    --enable-malloc0returnsnull
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

16.3.3 pixman

pixman (**pix**el **man**ipulation) 是底层像素处理函数库。编译 pixman 可依赖 libpng, 也可以通过选项`--disable-libpng` 使其不依赖 libpng。编译 libpng 请参考14.2节。实际上, 依赖 libpng 的只是它的测试程序, pixman 库本身的功能是独立的。

```
pixman 主页: http://www.pixman.org
pixman 源码: https://www.cairographics.org/releases/pixman-0.40.0.tar.gz
```

```
在 pixman 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-libpng \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

16.3.4 fontconfig

Fontconfig 为系统的文字显示提供配置和替换功能。它依赖下层的 FreeType 矢量字形处理库, 为上层的文字布局引擎 Pango 提供支持。Fontconfig 的配置文件使用 XML (e**X**tensible **M**arkup **L**anguage, 可扩展标记语言) 格式。Linux 系统有两套常用的 XML 库, 一个是 LibXML2, 一个是 Expat, 后者规模略小, 也是 Fontconfig 默认的依赖库。

```
freetype 主页: https://www.freetype.org
freetype 源码: https://download.savannah.gnu.org/releases/freetype/freetype
    ↪ -2.10.1.tar.bz2
expat 主页: https://libexpat.github.io
expat 源码: https://github.com/libexpat/libexpat/releases/download/R_2_2_9/
    ↪ expat-2.2.9tar.bz2
fontconfig 主页: https://www.freedesktop.org/wiki/Software/fontconfig/
fontconfig 源码: https://www.freedesktop.org/software/fontconfig/release/
    ↪ fontconfig-2.13.1.tar.gz
```

编译 freetype

FreeType 库实现矢量字体 TrueType、Type1 的显示效果支持, 在链接了 libpng 时还可以支持 PNG 压缩的点阵字体。编译 freetype 过程如下:

```
在 freetype 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
```

```
    --without-harfbuzz \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

编译 expat

```
在 expat 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

编译 fontconfig

Fontconfig 还依赖 util-linux 的 libuuid, 编译 util-linux 请参考 12.2.3节。编译 Fontconfig 过程如下:

```
在 fontconfig 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-cache-dir=/var/cache/fontconfig \
    --with-baseconfigdir=/etc/fonts \
    --with-default-fonts=/usr/share/fonts \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

16.3.5 编译 cairo

```
cairo 主页: https://cairographics.org
cairo 源码: https://cairographics.org/releases/cairo-1.16.0.tar.xz
```

```
在 cairo 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-gobject \
    --enable-xlib \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

16.4 HarfBuzz 移植

HarfBuzz 最早源自 FreeType 项目。早期的 HarfBuzz 只处理 OpenType 字体 [2]。目前的 HarfBuzz 专注于文字造型, 即从文本 Unicode 到字符型态的处理, 不提供文字布局及渲染。HarfBuzz 的开发者 Behdad Esfahbod 是伊朗裔加拿大软件工程师。软件名称源自 OpenType 的波斯语音译。

```
harfbuzz 主页: https://harfbuzz.github.io
harfbuzz 源码: https://www.freedesktop.org/software/harfbuzz/release/harfbuzz
    ↪ -2.6.4.tar.xz
```

编译 HarfBuzz 过程如下:

```
在 harfbuzz 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --without-icu \
    --with-gobject \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

选项 `--without-icu` 省去了 ICU (**I**nternational **C**omponents for **U**nicode, 国际通用字符集组件) 支持。

[2]开放字体, 由微软和 Adobe 公司联合开发的一种矢量字体, 采用 PostScript 格式, 使用文件扩展名.otf、.ttf、.ttc。

16.5 Pango 移植

Pango 是文本布局引擎库, 与字形引擎 HarfBuzz 共同实现多语种的文字渲染输出。软件名 Pango 来自希腊语 “Παν (’ 全部’ 的意思)” 和日语 “語 (音’go’)” 两个词组合的音译, 意指多语言支持。Pango 字形支持库由 FreeType、Fontconfig 和 HarfBuzz 提供, 通过图形库 Cairo 支持, 可完美实现高质量文字输出效果。

```
pango 主页: http://www.pango.org
pango 源码: http://ftp.gnome.org/pub/gnome/sources/pango/1.40/pango-1.41.0.tar
    ↪ .xz
```

```
在 pango 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=$INSTALL_PATH
```

16.6 通过 SSH 的 X11 转发

SSH 具有 X11 的转发功能, 即如果本地启动了 X 服务, 通过 ssh 命令登录到远程计算机、运行远程图形界面的程序时, 可以将远程的图形窗口显示在本地的服务器上 (图16.3)。为实现此

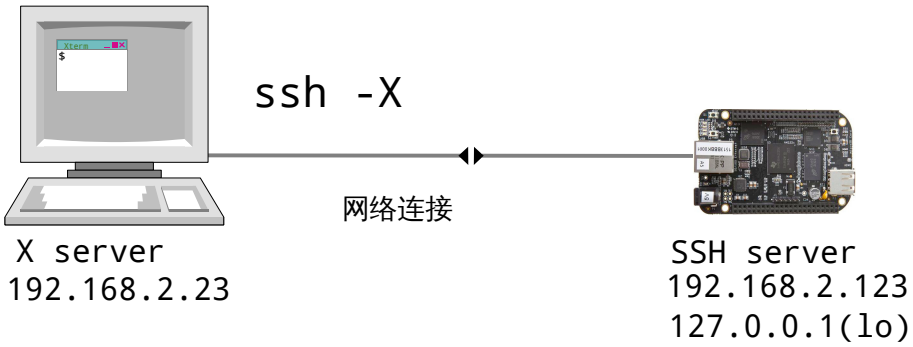


图 16.3: 通过 SSH 的 X11 转发

功能, 除了 SSH 服务以外, 还需要在远端安装一个 X 认证程序 xauth。图16.4是 xauth 的依赖关系, 相关软件清单如下:

```
部分X Window 库和应用源码:
https://www.x.org/releases/individual/lib/libICE-1.0.10.tar.gz
```

```
https://www.x.org/releases/individual/lib/libSM-1.2.3.tar.gz
https://www.x.org/releases/individual/lib/libXt-1.2.0.tar.gz
https://www.x.org/releases/individual/lib/libXmu-1.1.3.tar.gz
https://www.x.org/releases/individual/app/xauth-1.0.10.tar.gz
```

这组软件的编译方法与 libXau 相同 (配置 libXt 时需要选项 `--enable-malloc0returnsnull`)。

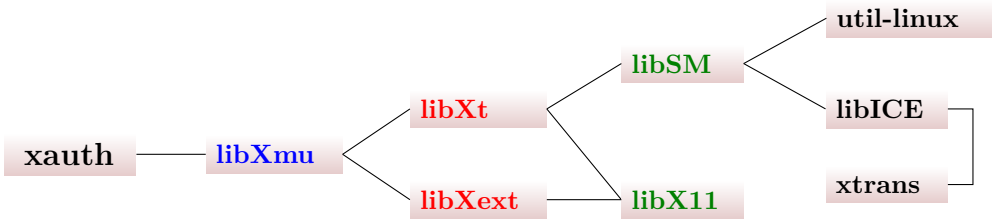


图 16.4: xauth 软件依赖关系

16.7 实验内容

1. 按图16.2的规划, 完成各个软件的交叉编译;
2. 将编译后的软件复制到目标系统;
3. 给系统安装一些字体。可以从 PC 的 /usr/share/fonts/ 目录里选择一些 TrueType 直接复制到目标系统。根据编译 Fontconfig 时的配置选项, 目标系统的字体文件目录也应是 /usr/share/fonts。配置文件目录是 /etc/fonts/, 默认的配置文件是 fonts.conf, 该文件来自 fontconfig 安装目录。此外, 还要根据 Fontconfig 配置选项的设定建立字体缓冲目录 /var/cache/fonts/。

   以上准备就绪, 用下面的命令创建字体缓冲文件:

   ```
   # fc-cache
   ```

   使用 fc-list 命令可以查看当前可用字体。fc-cache、fc-list 均移植自 Fontconfig。
4. 使用 Pango 命令 pango-view 输出 PNG 图形文件, 利用实验14编写的 PNG 图形显示程序显示字形, 见图16.5。注意画图颜色的选取, 如果不能区分前景和背景, 可能会看不到图。

   pango-view 常用的选项:

   `--text=string`: 要显示的文字, 如 `--text="Hello, World!"`。
   `--font=description`: 字体信息, 包括字体名、字形和字号。字体名可通过 fc-list 查询。
   `--rotate=degrees`: 文字旋转角度, 默认是水平方向。
   `--gravity=north/south/east/west`: 文字排版方向。
   `--markup`: 使用 pango 标记格式输入 Unicode 字符。如:

   ```
   # pango-view --markup --font="AR PL UKai CN 64" \
       --text="&#20320;&#22909;&#10;&#19990;&#30028;" \
       --output="hello.png"
   ```

`--background=#rrggbbaa/#rrggbb/transparent`: 设置背景颜色, 每个颜色分量用 2 位 16 进制符号表示。

`--foreground=#rrggbbaa/#rrggbb`: 设置前景颜色。

5. 为 Pango 增加 libXft 后端支持, 按图16.6的依赖关系重新编译 Pango。相关库源码:

```
https://www.x.org/releases/individual/lib/libXrender-0.9.10.tar.gz
https://www.x.org/releases/individual/lib/libXft-2.3.3.tar.gz
```

按15.2节完成 SSH 服务设置, 将 xauth 复制到目标系统 /usr/bin/目录, 尝试在 PC 上通过 X11 转发执行 pango-view:

```
$ ssh -X root@192.168.2.123
root@192.168.2.123's password:
# pango-view --text="Hello, World."
```

图 16.5: pango-view 输出文字的显示结果

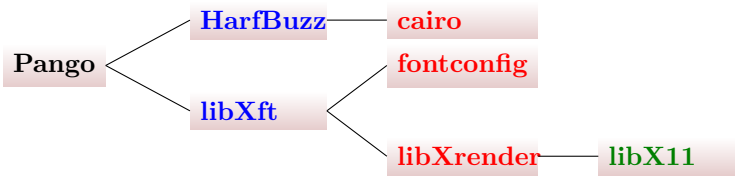


图 16.6: 重构的 Pango 软件依赖关系

本实验编译的内容较多, 实验过程也会出现多次反复。为了提高效率、减少错误, 建议将每个库的编译过程写成脚本程序。更一般地, 可编写一个较通用的脚本程序。例如清单16.3的脚本程序可适用多数 auto configure 配置方法的软件编译和安装:

清单 16.3: 编译脚本程序 build.sh

```
#!/bin/bash

INSTALL_PATH=/home/student/target
export CFLAGS="-I$INSTALL_PATH/usr/include"
export CPPFLAGS="-I$INSTALL_PATH/usr/include"
export CXXFLAGS="-I$INSTALL_PATH/usr/include"
export LDFLAGS="-L$INSTALL_PATH/usr/lib"
export LDFLAGS="$LDFLAGS -Wl,-rpath-link=$INSTALL_PATH/usr/lib"
export PKG_CONFIG_PATH="$INSTALL_PATH/usr/lib/pkgconfig:
    ↪ $INSTALL_PATH/usr/share/pkgconfig"
export PKG_CONFIG_LIBDIR="$PKG_CONFIG_PATH"
export PKG_CONFIG_SYSROOT_DIR="$INSTALL_PATH"

[ -d build_arm ] || mkdir build_arm
cd build_arm

../configure --host=arm-linux --prefix=/usr --enable-shared --
    ↪ disable-static $1
make -j8
make install DESTDIR=$INSTALL_PATH
```

配置中使用的其他参数可通过命令行添加。编译 libX11 的过程就变成了下面的样子:

```
在 libX11 解压目录下执行:
$ sh build.sh "--enable-malloc0returnsnull"
```

这样虽然做不到一劳永逸, 但也大大简化了移植过程。

17

GTK 移植

17.1 GTK 的背景

GTK 是一款跨平台的图形用户接口组件工具包, 是 Linux 系统中使用最为广泛的图形工具之一。GTK 原是 **G**IMP **T**ool**k**it 的缩写, GIMP (**G**NU **I**mage **M**anipulation **P**rogram) 是 Linux 系统重要的图像处理软件, 类似 Windows 中 photoshop 的地位。在使用面向对象的编程技术重新设计后, 软件被命名为 GTK+。开发团队决定 GTK4.0 之后不再使用 “+” 标记软件。

许多桌面系统都建立在 GTK 基础上, 如著名的 GNOME、XFCE4。

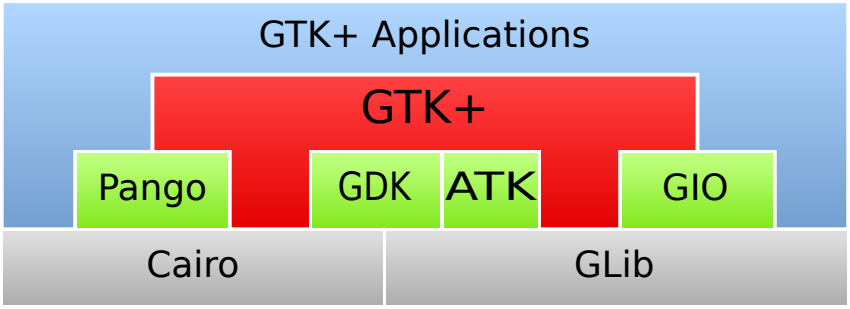


图 17.1: 基于 GTK 的软件层次结构[1]

GTK 不直接和显示硬件设备打交道。Linux 桌面系统通常基于 X Window。而在嵌入式应用中, 作为全功能 X 窗口系统的替代方案, 可以选择 DirectFB 作为后端。

从图17.1中可以看到, GTK 除了为上层应用提供接口函数库以外, 自身也依赖一系列的更底层的库。图17.2是建立 GTK 的库依赖关系, 因篇幅所限, 重复的依赖关系没有全部画出。我们需要根据这样的依赖关系逐层构造系统的 API。

本实验的最终目标是编译出 GTK3 的库以及演示程序 gtk3-demo。因涉及的库比较多, 依赖关系也比较复杂, 建议按下面的顺序编译:

1. X Window 系统;
2. 二维矢量图形库 cairo;

[1]图片引自 en.wikipedia.org/wiki/GTK。

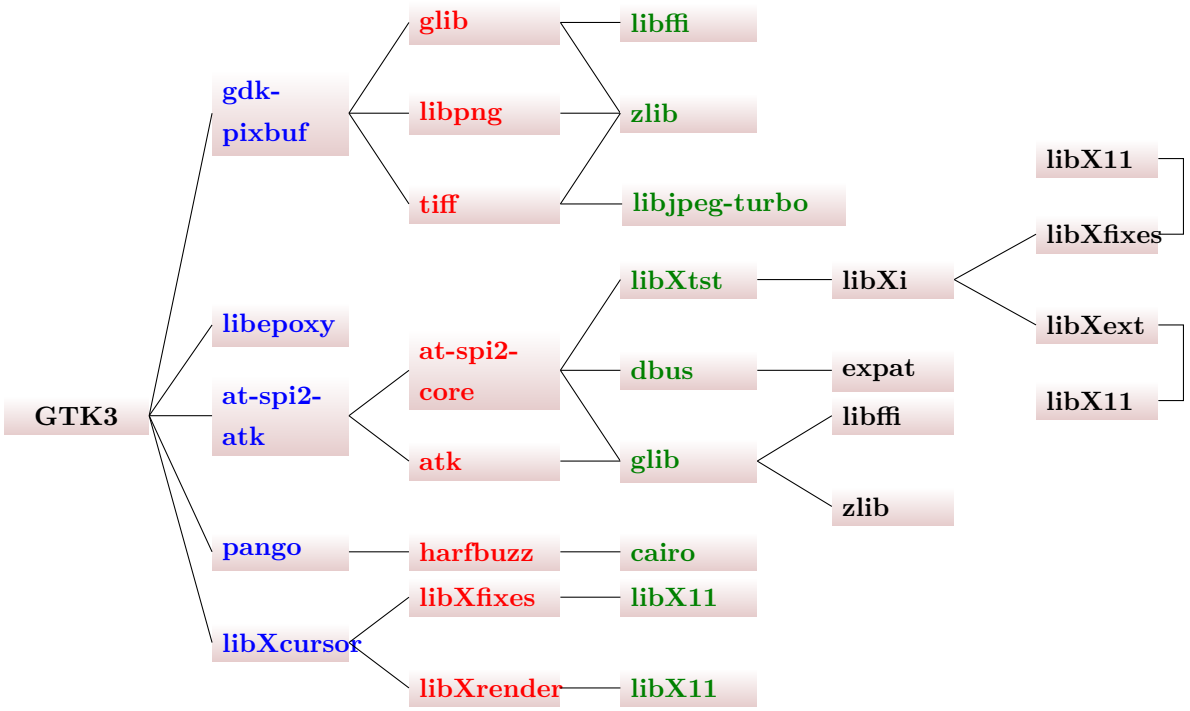


图 17.2: GTK3 的软件依赖关系

3. 文本布局引擎 pango;
4. 像素缓冲处理工具包 gdk-pixbuf;
5. 可访问性工具包 atk 及 at-spi2-atk;
6. OpenGL 函数指针管理接口库 libepoxy;
7. gtk3.

X Window 库编译方法请参考16.3.2节, 与本实验有关的 X 库有下面几个。Cairo 和 Pango 编译过程请参考第16章。

```
部分X Window 库源码:
https://www.x.org/releases/individual/lib/libXfixes-5.0.3.tar.gz
https://www.x.org/releases/individual/lib/libXi-1.7.10.tar.gz
https://www.x.org/releases/individual/lib/libXtst-1.2.3.tar.gz
```

17.2 图形工具 gdk-pixbuf

GDK-PixBuf 完成图像缓冲的处理工作, 原也是属于 GTK+ 的一部分。在 gtk+-2.21 版本之后从 GTK+ 中剥离。它依赖 libpng、libjpeg 和 tiff 等底层图形库。libpng 和 libjpeg 的移植请参考第14章。

```
tiff 主页: http://www.remotesensing.org/libtiff
tiff 源码: http://download.osgeo.org/libtiff/tiff-4.0.10.tar.gz
```

```
gdk-pixbuf 主页: https://www.gtk.org
gdk-pixbuf 源码: https://download.gnome.org/sources/gdk-pixbuf/2.36/gdk-pixbuf
    ↪ -2.36.0.tar.xz
```

17.2.1 编译 tiff

Tiff 库用于处理 TIFF (**T**agged **I**mage **F**ile **F**ormat, 带标记的图像文件格式) 文件。这种格式的文件最初用于桌面排版系统, TIFF 可以将多种图形格式包装在一个文件里。

```
在 tiff 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.2.2 编译 gdk-pixbuf

```
在 gdk-pixbuf 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    gio_can_sniff=yes \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.3 编译 libepoxy

libepoxy 是管理 OpenGL 库函数指针的接口, 默认依赖 OpenGL。mesa (https://www.mesa3d.org/) 是 OpenGL 的开源库。通常编译 libepoxy 之前需要先编译 mesa, 而 mesa 的编译和配置都比较复杂且耗时。这里作了简化处理, 通过选项 `--disable-egl` 避开这个依赖关系。

```
在 libepoxy 解压目录下执行:
$ mkdir build_arm && cd build_arm
```

```
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    X11_CFLAGS=-I/home/student/target/usr/include/X11 \
    X11_LIBS=-lX11 \
    --disable-egl \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.4 可访问性工具包 ATK

ATK (**A**ccessibility **T**ool**K**it) 提供一组接口, 形成客户机/服务器框架。客户端名称为 AT-SPI (**A**ssistive **T**echnology **S**ervice **P**rovider **I**nterface, 辅助技术服务提供接口), 下层是 X Window 和 DBus, 上层是桌面系统。

```
dbus 主页: https://dbus.freedesktop.org
dbus 源码: https://dbus.freedesktop.org/releases/dbus/dbus-1.13.10.tar.gz
atk 主页: https://developer.gnome.org/atk
atk 源码: https://download.gnome.org/sources/atk/2.26/atk-2.26.1.tar.xz
at-spi2-core 主页: https://wiki.gnome.org/Apps
at-spi2-core 源码: https://download.gnome.org/sources/at-spi2-core/2.26/at-
    ↪ spi2-core-2.26.3.tar.xz
at-spi2-atk 主页: https://wiki.gnome.org/Apps
at-spi2-atk 源码: https://download.gnome.org/sources/at-spi2-atk/2.26/at-spi2-
    ↪ atk-2.26.2.tar.xz
```

17.4.1 编译 dbus

DBus (**D**esktop **Bus**) 提供桌面系统的进程间通信机制, 它使用 XML 语言的配置文件, 因此也依赖 Expat。编译 DBus 过程如下:

```
在 dbus 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-dbus-user=dbus \
    --disable-selinux \
```

```
    --disable-tests \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.4.2 编译 atk

```
在 atk 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.4.3 编译 at-spi2-core

```
在 at-spi2-core 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.4.4 编译 at-spi2-atk

```
在 at-spi2-atk  解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --enable-shared \
```

```
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.5 编译 GTK3

```
gtk 主页: https://wiki.gnome.org/Projects/GTK
gtk 源码: https://download.gnome.org/sources/gtk+/3.24/gtk+-3.24.4.tar.xz
```

```
在 gtk3 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-xkb \
    --disable-xinerama \
    --disable-xrandr \
    --disable-xdamage \
    --disable-xcomposite
    --enable-x11-backend \
    --enable-shared \
    --disable-static
$ make -j8
$ make install DESTDIR=/home/student/target
```

17.6 实验内容

1. 根据图17.2的关系, 完成 GTK3 的编译。将安装目录 target/usr 整体平移到目标系统, 保持子目录结构不变。用下面的命令设置可加载模块缓存:

   ```
   # gdk-pixbuf-query-loaders > /usr/lib/gdk-pixbuf-2.0/2.10.0/loaders.cache
   ```

   复制主机上的一些字体库到 /usr/share/fonts/, 用 fc-cache 命令更新字体信息。尝试通过 SSH 的 X11 转发功能运行 gtk3-demo 程序, gtk3-demo 应该在 /usr/bin/ 目录下。

   ```
   $ ssh -X root@192.168.2.123
   # gtk3-demo
   ```

   由于字体和图形环境没有正确设置, 初次运行会有一些不成功的提示信息。请根据这些提示完善系统设置。

2. GTK 源码中包含一些示例程序。请尝试根据示例编写一个简单的 GTK+ 图形界面程序，编译并运行。编译时会用到 GTK+ 及其依赖的库和头文件。由于 GTK 应用程序依赖的库比较多，需要使用 pkg-config 获得库的路径，而 pkg-config 又依赖环境变量 PKG_CONFIG_PATH 和 PKG_CONFIG_SYSROOT_DIR。为避免操作上的烦琐，可以使用下面的 Makefile:

```
1 CC      = arm-linux-gcc
2
3 LDFLAGS = -L/home/student/target/usr/lib -Wl,-rpath-link=/home/
     ↪ student/target/usr/lib
4 LIBS = -luuid
5 hello: hello.c
6         $(CC) -o @< $(LDFLAGS) $(LIBS) `pkg-config --cflags --
             ↪ libs gtk+-3.0`
```

用下面的命令编译:

```
$ export PKG_CONFIG_PATH=/home/student/target/usr/lib/pkgconfig
$ export PKG_CONFIG_SYSROOT_DIR=/home/student/target
$ make
```

18

X 窗口服务器

Linux 的图形界面基于 X Window 系统。X Window 系统是一种客户端/服务器模型。Linux 操作系统的桌面环境 GNOME 和 KDE 都是以 X Window 窗口系统为基础建立的。

18.1 X Window 简介

X 窗口系统 (X Window System) 是一种以位图方式显示的软件窗口系统。“X” 在字母表中排在 “W” 之后, X Window 意思是下一代窗口系统。X Window 最初由 MIT 于 1984 年为 UNIX 系统开发, 后来成为 UNIX 及类 UNIX 操作系统一致适用的标准化软件工具包及显示架构的运作协议。1987 年 9 月, 协议发布到 11 版, 由此 X Window 系统也常称为 X11 或 X, 其中以 X.Org 基金会 (https://www.x.org) 维护的参考实现最为著名。目前软件版本形式以 “协议版本 + 发布版本” 命名, 最新的稳定版 X11R7.7 于 2012 年 6 月发布。

X 能为图形用户接口提供基本的输入输出框架: 受理鼠标、键盘的交互操作, 在屏幕上显示文字和绘制图形、移动窗口。由于 X 并不负责用户界面部分, 用户界面由其他以 X 为基础的实现体来负责, 因此以 X 为基础环境所开发的外观形式多样, 不同的程序可能有截然不同的接口呈现。

X 采用客户端/服务器架构模型 (C/S 模型), 由一个 X 服务器与多个 X 客户端程序进行通讯, 服务器接受对于图形输出（窗口）的请求并反馈用户输入 (键盘、鼠标、触摸屏等等), 服务器是一个能显示到其他显示系统的应用程序, 可能是控制某个 PC 的视频输出的系统程序, 也可能是个特殊硬件。

X 的一大特点在于 “网络透明性”, 服务器和客户端之间的通信协议的运作对计算机网络是透明的: 客户端和服务器可以在同一台计算机上, 也可以是不同的计算机, 甚至是不同的硬件架构和操作系统。

为了使远端客户程序显示到本地服务器, 用户通常需要启动一个终端窗口和到达远端计算机的登录工具 (如 SSH); 与之对应, 本地计算机上也可以执行一个连接到远端计算机的小型代理程序, 并在该端启动与运行自有需求与指定的应用程序。实际的远端客户端的例子有: 图形化管理远程计算机; 在远端计算机上运行计算密集的仿真程序并把结果显示到本地的桌面; 用一套显示器、键盘和鼠标控制同时运行在多台计算机上的图形化软件, 等等。

18.2 X 服务移植

本实验移植 X Window 的服务器部分。图18.1是最简化的依赖关系。有关 X 库的移植请参考16章和17章。本实验至少需要编译如下软件:

```
https://www.x.org/releases/individual/lib/libpciaccess-0.16.0.tar.gz
https://www.x.org/releases/individual/lib/libfontenc-1.1.3.tar.gz
https://www.x.org/releases/individual/lib/libXfont2-2.0.4.tar.gz
https://www.x.org/releases/individual/lib/libxkbfile-1.0.9.tar.gz
https://www.x.org/archive/individual/xserver/xorg-server-1.20.8.tar.bz2

https://www.x.org/releases/individual/driver/xf86-video-fbdev-0.5.0.tar.gz
https://www.x.org/releases/individual/data/xkeyboard-config/xkeyboard-config
    ↪ -2.30.tar.gz
https://www.x.org/releases/individual/app/xkbcomp-1.4.3.tar.gz
```

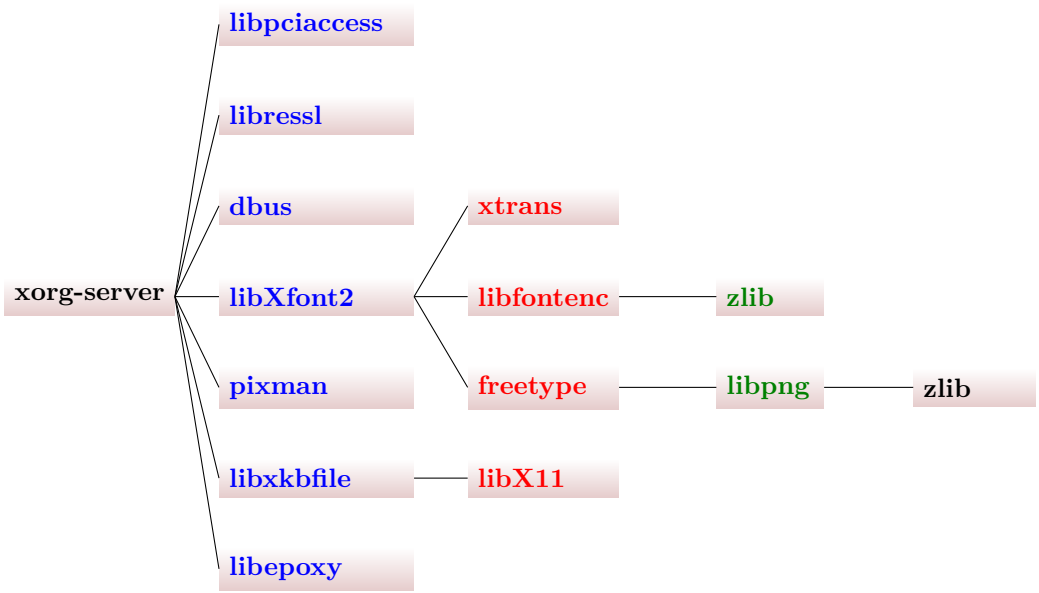


图 18.1: X Server 软件层次关系

本实验移植的软件只有 xorg-server 配置比较复杂, 其他软件都可以仿照 X11 库的方式编译。下面是编译 xorg-server 的过程。配置选项不是唯一的。当底层支持库丰富时, 其中的一些 --disable-项可以省略, 由配置工具自动设置, 或者改为 --enable-, 以明确依赖关系。

```
在 xorg-server 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
```

```
    --disable-record \
    --disable-xvmc \
    --disable-config-udev-kms \
    --disable-xdmcp \
    --disable-xdm-auth-1 \
    --disable-glx \
    --disable-xwayland \
    --disable-dri \
    --disable-glamor \
    --disable-dri2 \
    --disable-dri3 \
    --disable-libdrm \
    --disable-xshmfence \
    --disable-unit-tests \
    --without-systemd-daemon \
    --disable-systemd-logind \
    --with-log-dir=/var/log \
    --with-fontrootdir=/usr/share/fonts \
    --with-module-dir=/usr/lib/xorg/modules \
    --with-xkb-path=/usr/share/X11/xkb \
    --with-xkb-output=/var/cache/xkb \
    --with-default-font-path=/usr/share/fonts/X11/misc,built-ins \
    --with-serverconfig-path=/usr/lib/xserver
$ make -j8
$ make install DESTDIR=/home/student/target
```

启动 X 服务, 至少还需要 Frame Buffer 驱动 xf86-video-fbdev、X 键盘扩展数据 xkeyboard-config 和程序 xkbcomp。如果要支持鼠标和触摸屏这些 event 设备, 还需要移植 xf86-input-evdev 驱动, 它依赖多点触摸协议库 mtdev 和内核 event 设备接口库 libevdev。

```
mtdev 主页: https://bitmath.org/code/mtdev
mtdev 源码: https://bitmath.org/code/mtdev/mtdev-1.1.5.tar.gz
libevdev 主页: https://www.freedesktop.org/wiki/Software/libevdev
libevdev 源码: https://www.freedesktop.org/software/libevdev/libevdev-1.5.7.
    ↪ tar.xz
xf86-input-evdev 源码:https://www.x.org/releases/individual/driver/xf86-input-
    ↪ evdev-2.10.6.tar.gz
```

这些库的编译方法与 libX11 相同。

18.3 移植终端仿真器

```
xterm 主页: http://invisible-island.net/xterm/xterm.html
xterm 源码: https://invisible-mirror.net/archives/xterm/xterm-333.tgz
```

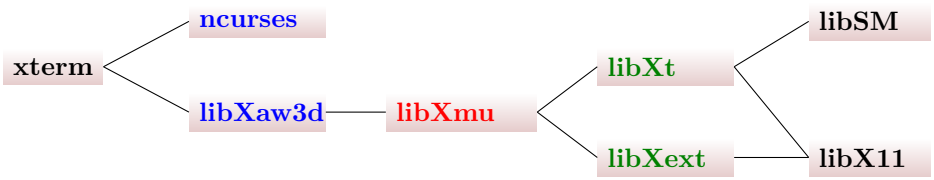


图 18.2: xterm 软件依赖关系

```
在 xterm 解压目录下执行:
$ ./configure \
    --host=arm-linux \
    --prefix=/usr \
    --with-Xawud \
    --without-xinerama
$ make -j8
$ make install DESTDIR=/home/student/target
```

18.4 启动 X 服务

软件编译后, 将它们复制到目标系统, 用下面的命令启动 X 服务:

```
# Xorg :0 &
```

Xorg 的参数形式一般用 “host:number” 表示。host 是主机地址, 缺省时表示本地。上面的操作表示在本地启动 0 号服务。同一台主机可以启动多个 X 服务, 每个 X 服务上面可以运行不同的图形工作环境。缺省参数时, 默认为 “:0”。如果启动不成功, 注意阅读日志文件 /var/log/Xorg.0.log。

X 服务启动后, 图形界面程序需要环境变量 DISPLAY 指导它运行在哪个服务上。根据 X 服务的参数, 设置环境变量如下:

```
# export DISPLAY=:0
```

下面就可以启动图形程序了。本实验移植了 xterm, 之前如果移植了 GTK, 这些图形界面程序都可以在本地 X 服务运行:

```
# xterm &
# gtk3-demo &
```

我们发现, 这些程序没有窗口, 只有界面。它们无法定位, 也无法移动。管理图形界面窗口任务的是窗口管理器。

本实验如采用虚拟显示方案，在 *PC* 上通过 *VNC* 连接目标系统，应移植基于 *X11* 的 *VNC* 服务器，按图*18.3*的依赖关系编译 *x11vnc*。

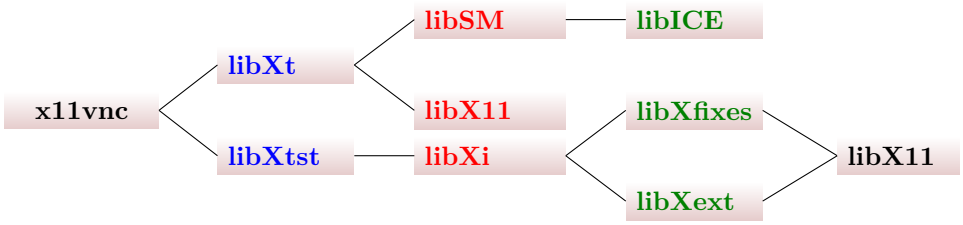


图 18.3: x11vnc 软件依赖关系

按如下选项配置编译 *x11vnc:*

```
在 x11vnc 解压目录下执行:
$ ./configure \
    --host=arm-linux \
    --prefix=/usr \
    --without-ssl \
    --without-xinerama \
    --without-xdamage \
    --without-xrandr
$ make -j8
$ make install DESTDIR=/home/student/target
```

启动 *VNC* 服务*:*

```
# x11vnc -display :0 -forever -shared >/dev/null 2>/dev/null &
```

选项 `-display` 的设置应与 *Xorg* 服务器启动时设置的显示器序号相同。

18.5 实验内容

完成 X 服务器和终端仿真器 xterm 的移植。如果通过虚拟显示连接目标系统, 可由教师完成 x11vnc 的移植并提供给学生使用。

19

窗口管理器

在 X Window 系统中, 用来管理窗口外观、布局的软件被称作 X 窗口管理器。

在苹果的 Mac OS 操作系统和微软的 Windows 操作系统中, 窗口管理都有长期固定的用户界面和操作模式, 这些都是由开发商决定的, 用户无法更换, 只能作微小调整。X 窗口管理器则提供完全开放的、和图像显示软件无关的用户界面, 用户可以自由选择不同的窗口管理器。

本实验选择窗口管理器 Fluxbox 进行移植。

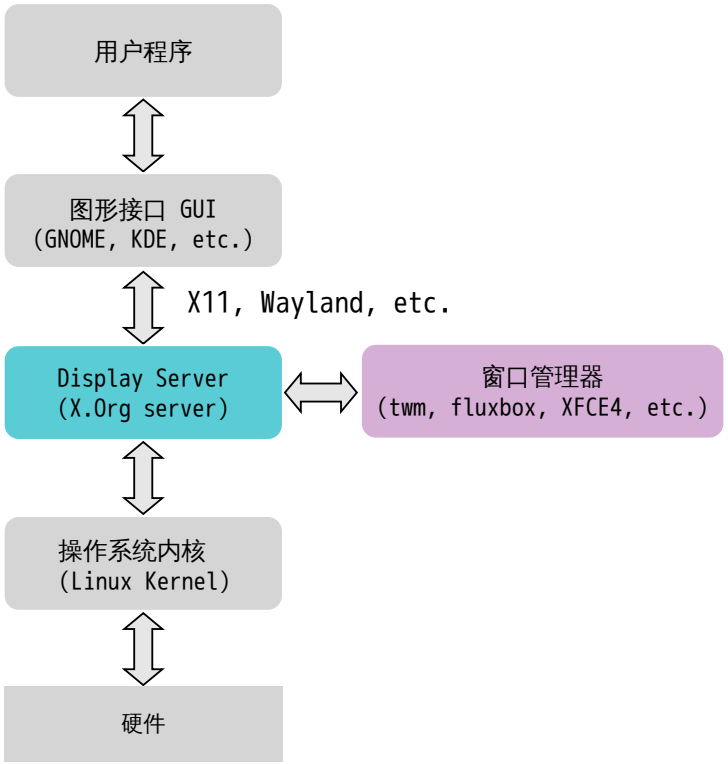


图 19.1: 图形界面与窗口管理器

19.1 窗口管理器工作过程

图19.1是窗口管理器在系统中的位置。窗口管理器启动后, X 服务器和客户端之间的交互会重定向到窗口管理器。每当创建一个新的图形界面程序, 窗口管理器将决定这个图形界面窗口的初始位置, 在图形界面的外围加上边框, 在顶部加上标题栏, 以及最大化、最小化、关闭窗口等等操作的组件。当用户点击或拖曳那些组件时，窗口管理器会执行相应的动作, 如最大化、最小化、关闭窗口结束程序, 以及移动或改变窗口的大小。

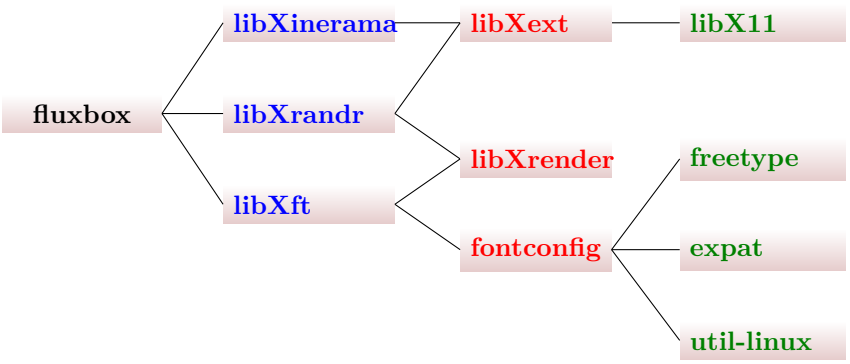


图 19.2: fluxbox 软件层次关系

19.2 移植窗口管理器

Fluxbox 是基于 X11 的轻量级窗口管理器, 用 C++ 语言编写, 由 Blackbox 分支而来。

```
https://www.x.org/releases/individual/lib/libXinerama-1.1.4.tar.gz
https://www.x.org/releases/individual/lib/libXrandr-1.5.2.tar.gz

fluxbox 主页: http://www.fluxbox.org
fluxbox 源码: http://sourceforge.net/projects/fluxbox/files/fluxbox/1.3.7/
    ↪ fluxbox-1.3.7.tar.gz
```

X 库的编译方法请参考 libX11, 这里只写出 fluxbox 的编译过程:

```
在 fluxbox 解压目录下执行:
$ mkdir build_arm && cd build_arm
$ ../configure \
    --host=arm-linux \
    --prefix=/usr \
    --disable-fribidi \
    ac_cv_func_malloc_0_nonnull=yes \
    ac_cv_func_realloc_0_nonnull=yes
$ make -j8
```

```
$ make install DESTDIR=/home/student/target
```

启动窗口管理器:

```
# Xorg &
# fluxbox &
```

随 fluxbox 安装的, 还有一些辅助程序 (启动命令 startfluxbox、颜色和壁纸设置) 以及 /usr/share/fluxbox/ 目录下的配置文件。由于缺少 X 应用程序, 部分辅助程序可能无法正常工作。首次启动窗口管理器, 配置文件会被复制到用户的主目录 ~/.fluxbox。这些配置文件都是文本文件, 用户可以手工编辑这些配置文件, 以改变窗口管理器的风格。窗口管理器风格也可以通过桌面菜单编辑, 桌面菜单用鼠标中键和右键调出, 见图19.3。

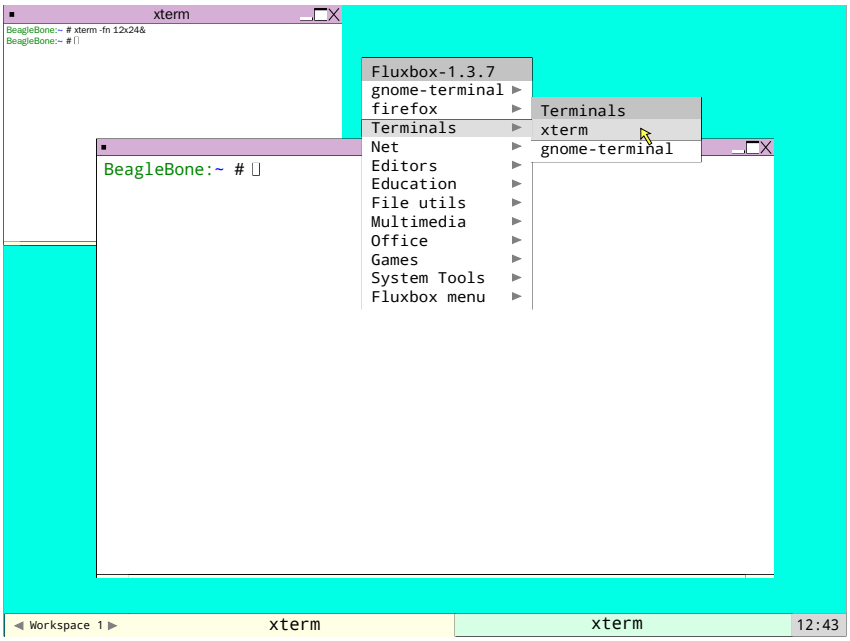


图 19.3: 窗口管理器 Fluxbox

索引