Embedded system 实验一

在开发板上安装$Linux$系统

171180616 郭成伟 NJUEE

背景介绍

我们知道，嵌入式$linux$系统有很多优点。

1. 开放源码
2. 所需容量小（最小大约需要2MB）
3. 成熟与稳定
4. 支持良好

基于这些优点，市场上有很多基于嵌入式$Linux$系统的产品，诸如智能手机，路由器设备，无人机等等，大部分产品都离不开嵌入式开发。据统计，2017年，使用基于$Linux$的$Android$系统的手机与平板占市场份额60%以上，此外服务器等大型计算机也大部分都使用$Linux$开发。由此，我们学习嵌入式$Linux$开发是很有必要的。

嵌入式$Linux$系统的典型层次有

- $bootloader$以及$Uboot$
- $kernel$
- $root\quad filesystem$
- $application$

在我们这一系列的实验中，我们对$Bootloader$部分没有要求。

将电源线、网线、串口线连接到嵌入式处理器上。

串口线连接方式

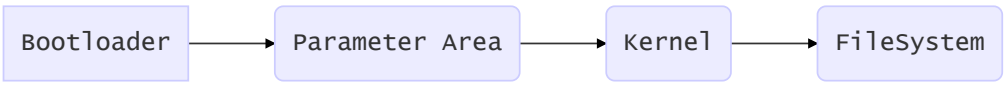
在板上有六个端口

- 1号口 黑色地线
- 4号口 绿色发送线
- 5号口 白色接收线
- 红色电源线（此实验中我们不使用）

连接好了之后，开发板与主机已经连接完毕。

开发板的启动

开发板启动后，$Linux$ 系统的一般启动过程包括引导内核、启动内核和启动初始化程序这几个阶段：加电后，处理器自动去读存储器的特定位置(硬盘的主引导记录MBR或者磁盘的第一个扇区)，实现目标板硬件的初始化，并跳转到内核的开始地址。典型的内存结构如下：



内核初始化之后，挂载根文件系统，从文件系统中启动$init$程序以完成剩下的系统启动过程。

串口设置

在主机中，使用$minicom$与开发板建立通信 （异步串行通信）

运行$minicom$后，$Ctrl + A + o$进入配置界面，其中我们进入 $serial \quad port \quad setup$进行一些通信协议的配置。

其中，A项 $serial \quad device$是串行通信口的选择。

我们需要把串口通信口设置成$/dev/ttyUSB0$ ，因为我们通过USB转$RS232$接入电脑，所以设备需要如前述设置。

E项$Bps/Par/Bits$则是串口参数的设置。设置通信波特率（并非比特率）、数据位、奇偶校验位以及停止位。本实验设置为：$115200Bps$，8位数据位，无校验，一个停止位。

奇偶校验指的是，发射数据途中，发射一个校验位来标志数据长度的奇偶性，此类校验只能提示出错否，并没有纠错功能。

设置完成后，我们可以在 $minicom$ 主界面上看到开发板的启动信息。在开启的小段时间内，通过键盘干预,进入 $U-Boot$。退出$minicom$的命令是$Ctrl + a + x$ 调出全部功能菜单$Ctrl + a + z$

环境变量设置

我们要建立开发板与主机之间的连接，就要主动分配ip地址。

```
setenv ipaddr 192.168.208.133  #分配ip地址给开发板
setenv serverip 192.168.208.12  #主机地址
```

$bootargs$启动参数，

```
set bootargs root=/dev/ram initrd=0x88080000 console=ttyO0,115200
#root文件系统位于flash的ram上
#当你使用ramdisk启动系统的时候需要指定initrd=r_addr, r_addr表示initrd在内存中的位置。
#串口设备的终端是ttyO0，波特率是115200

```

$tftp$

$tftp$ 是基于UDP 协议的文件传输协议。开发板作为客户机，$bootloader$默认采用 $tftp$协议。在配置好上述ip地址后，可通过$tftp$命令传输文件（需指定传输后板上保存地址）。此协议设计的时候是进行小文件传输的。因此它不具备通常的FTP的许多功能，它只能从文件服务器上获得或写入文件，不能列出目录，不进行认证。

1. TFTP可用于UDP环境；比如当需要将程序或者文件同时向许多机器下载时就往往需要使用到TFTP协议。
2. TFTP代码所占的内存较小，这对于我们嵌入式开发来说很有意义。通常我们的设备的内存空间或者硬盘空间并不大。设备不需要硬盘，只需要固化了TFTP、UDP和IP的小容量只读存储器即可。当电源接通后，设备执行只读存储器中的代码，在网络上广播一个TFTP请求。网络上的TFTP服务器就发送响应，其中包括可执行二进制程序。设备收到此文件后将其放入内存，然后开始运行程序。这种方式增加了灵活性，也减少了开销。
3. TFTP初始连接时候需要发出WRQ（请求写入远程系统）或RRQ（请求读取远程系统），收到一个确定应答，一个确定可以写出的包或应该读取的第一块数据。通常确认包包括要确认的包的包号，每个数据包都与一个块号相对应，块号从1开始而且是连续的。因此对于写入请求的确定是一个比较特殊的情况，因此它的包的包号是0。如果收到的包是一个错误的包，则这个请求被拒绝。创建连接时，通信双方随机选择一个TID，因为是随机选择的，因此两次选择同一个ID的可能性就很小了。每个包包括两个TID，发送者ID和接收者ID。这些ID用于在UDP通信时选择端口，在第一次请求的时候会将请求发到TID 69，也就是服务器的69端口上。应答时，服务器使用一个选择好的TID作为源TID，并用上一个包中的TID作为目的ID进行发送。这两个被选择的ID在随后的通信中会被一直使用。

传输时，在主机上须将文件放入$tftp$文件夹之中（$/srv/tftpboot$），在$minicom$中输入命令：

```

tftp 0x82000000 zImage4.4 #将主机上的zImage4.4内核映像传入目标开发板地址0x82000000
tftp 0x88080000 ramdisk_img.gz#以下同理
tfpt 0x88000000 am335x-boneblack.dtb
#加载文件
bootz 0x82000000 0x88080000:0x1996a1 0x88000000
#以压缩方式载入文件
```

如果文件正确，那么内核就能正确加载进入开发板并且进入开发板$Linux$系统的命令行$console$

进入系统之后，我们可以设置开发板的网络，使之可以连接外网：

```
route add default gm +ip #设置路由地址
telnet +ip #使用talnet连接外部网络。
```

至此我们实验的第一部分已经结束了：使用提供的映像文件完成安装。

上文所述文件均为教师提供。接下来我们逐步将上述文件替换成我们编译产生的新文件。

内核编译

$Linux$内核源代码可以从网上下载。在这边我们使用的$linux$版本是$4.4.155$

可以用如下命令查看

```
cat /proc/version
```

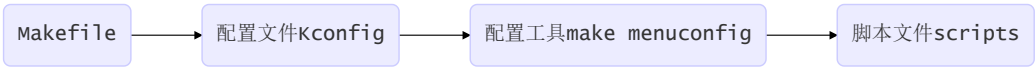
我们知道，$Linux$文件系统按树形结构进行组织，这些目录按照约定俗成分别存放不同的内容。在嵌入式开发的过程中，我们需要对这些规则有一定了解，主动将相应文件存放入相应文件夹，以便其他人使用时更为方便。下文便简单地介绍一些文件夹的区别。

- $arch$:（$architecture$）$arch$子目录包括所有与体系结构有关的内核代码。$arch$的每一个子目录都代表一个linux所支持的体系结构。
- $include$：包括一些编译内核所需要的头文件

- $init$：包含内核的初始化代码，其中包含的$main.c$文件是研究$linux$内核的起点
- mm：该目录包含所有独立于CPU体系结构的内存管理代码，如页式存储管理、内存的分配、释放等。
- $kernel$：包括主要的内核代码。
- $drivers$：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. $block$：块设备驱动文件
    2. $char$：字符设备驱动程序。如鼠标、串口
    3. $cdrom$： 包含Linux所有的$CD-ROM$代码
    4. pci：PCI卡驱动程序代码
    5. net：网络设备驱动程序
    6. $sound$：声卡设备驱动程序
- lib：放置内核的库代码
- net：包含内核与网络的相关的代码
- ipc：包含内核进程通信的代码
- fs：包含所有的文件系统代码和各种类型的文件操作代码
- $scripts$：包含用于配置内核的脚本文件等。内核源码每个目录下都有一个$Kconfig$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，他们通过脚本文件发生作用。

配置内核

$Linux$内核的配置系统由四个部分组成：



- $Makefile$：分布在$Linux$内核源码中的$Makefile$ 定义了$Linux$ 内核的编译规则。顶层 $Makefile$ 是整个内核配置、编译的总体控制文件；
- 配置文件$Kconfig$：给用户提供配置选择的功能。`.config` 是内核配置文件，包括由用户选择的配置选项，用来存放内核配置后的结果。
- 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。一般有4种设定配置的方法（`make config`、`make oldconfig`、`make menuconfig`、`make xconfig`）。在这里我们使用`make menuconfig`
- 脚本文件：在`/script`目录下

在命令行我们使用`make menuconfig`根据自己的要求对Linux内核进行配置。

```
ls arch/arm/configs/ #官方对开发板的配置文件
make bb.org_defconfig ARCH=arm # 将上述目录下的配置文件更新到内核的配置文件中去
make ARCH=arm menuconfig #基于上述配置文件再进行客制化配置
```

值得一提的是，为了满足接下来我们编译文件系统的需要我们需要配置 一些环境：

1. 在 “Block devices→” 中选中 “device driver”在其目录下的“disk ram block device support” 。
2. 在 “General setup” 设置分支里选中 “Initial RAM filesystem andRAM disk (initramfs/initrd) support”

在配置完成之后配置工具会生成一个`.config`文件，并建立过程中其余步骤将用到的一些符号链接和头文件，并将之前的配置文件被分到`.config.old`,以便用户可能需要恢复上一次的配置。

可以看到我们在上述过程中均使用了声明：`ARCH=arm`

ARCH 变量用来指明要为哪种架构建立内核，内核源码树里各个 Makefile 会根据此变量来选择它准备使用的与架构相关的目录。当为目标板编译内核的时候必须将 ARCH 变量设成目标板的架构，对这个以 ARM 为基础的用户接口模块，应将 ARCH 设为 arm。

当然我们也可以使用别的方法达到同样的效果。``

```
export ARCH=arm
#或者 直接更改Makefile文件
vim Makefile
```

再配置好我们客制化的内核之后，就可以开始编译过程了。

交叉编译

交叉编译是指在一个平台上生成另一个平台上的可执行代码。在我们这个例子上，我们在Linux x86系统上编译生成 Linux arm 的内核文件。

为什么我们需要交叉编译呢?

在嵌入式开发中，我们很难直接在目标板上直接编译可执行文件。基于前文所述嵌入式开发的优点，同时也限制了一些我们的开发条件。有时是因为目的平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；有时是因为目的平台上的资源贫乏，无法运行我们所需要编译器；有时又是因为目的平台还没有建立，连操作系统都没有，根本谈不上运行什么编译器。所以我们采用交叉编译的方式编译内核。

```
make ARCH=arm CROSS_COMPILE=arm-noe-linux-gnueabi- -j8 #j8多線程編譯
```

使用多线程编译可以加快我们的编译速度。

编译完成后，生成zImage内核映像文件，位于`linux/arch/arm/boot`，我们把这个文件拷贝到我们的tftp服务器之中等待传送。

根文件系统制作

根文件系统首先是一种文件系统，文件系统是对一个存储设备上的数据和元数据进行组织的机制。这种机制有利于用户和操作系统的交互。对 Linux 来说，文件是用户与操作系统交互所采用的主要工具。

在上述内核编译模块中，我们简要介绍了一些目录。

而Linux根文件系统中一般有如下的几个目录：

- **/bin目录**

  由于该目录下的命令在挂接其它文件系统之前就可以使用，所以/bin目录必须和根文件系统在同一个分区中。

  /bin目录下常用的命令有：cat、chgrp、chmod、cp、ls、sh、kill、mount、umount、mkdir等等

- **/sbin 目录**

存放基本的系统命令，只有root权限的用户可以使用，一半用于启动和修复系统，在挂接其他文件系统之前就可以使用/sbin，所以/sbin目录必须和根文件系统在同一个分区中。

/sbin目录下常用的命令有：shutdown、reboot、fdisk、fsck、init等。

- **/dev目录**

  存放与设备和设备接口相关的文件。在$Linux$系统下，通常以文件的方式访问各种设备。比较重要的文件有/dev/null, /dev/zero, /dev/tty等。

- **/etc目录**

  该目录下存放着系统主要的配置文件，例各种账号密码文件等等

- **/lib目录**

  该目录下存放一些库文件，共享库用于启动系统，运行根文件系统中的可执行程序。

- **/proc目录**

  这是一个空目录，常作为proc文件系统的挂接点，proc文件系统是个虚拟的文件系统，它没有实际的存储设备，里面的目录，文件都是由内核临时生成的，用来表示系统的运行状态，也可以操作其中的文件控制系统。

- **/mnt目录**

  用于临时挂在某个文件系统的接点。

我们利用Busybox制作根文件系统就是创建这上面的这些目录，和这些目录包含的各种文件。

根文件系统和普通文件系统的不同之处在于，内核启动后第一个挂载的是根文件系统。系统引导启动程序会在根文件系统挂载之后把一些初始化的服务加载到内存中取运行。若系统不能从指定设备上挂载根文件系统，则系统会出错而退出启动。成功之后可以自动或手动挂载其他的文件系统。因此，一个系统中可以同时存在不同的文件系统。

Busybox简介

BusyBox将许多具有共性的小版本的UNIX工具结合到一个单一的可执行文件。这样的集合可以替代大部分常用工具比如的GNU fileutils， shellutils等工具。BusyBox提供了一个比较完善的环境，并且由于采用了模块化设计，使之可以进行功能上的裁剪以及瘦身，因此可以适用于任何小的嵌入式系统。

其特点是所有命令都编译成一个文件：BusyBox，其他命令工具（例如sh、cp、ls 等）都是指向 BusyBox 文件的链接。在使用BusyBox生成的工具时，会根据工具的文件名链接到特定的处理程序，这样，所有这些程序就只需要被加载一次，而所有的BusyBox 工具组件都可以共享相同的代码段，这在很大程度上节省了系统的内存资源，也提高了应用程序的执行速度。

下载好BusyBox的安装包后，进入解压目录，仿照内核配置过程输入 `make menuconfig`,进入安装配置。

- 在`busybox setting/build option`菜单下，选择静态库编译方式
- 设置安装文件夹,文件系统以这个路径为起点。

设置完成之后，输入`make install`安装。

安装完毕之后，开始文件系统的构建。

文件系统构建

- 构建`/etc`目录，在etc下创建`inittab`、`rc`、`motd`三个文件

`inittab`文件是由系统启动程序init读取并解释执行。

rc文件要求可执行属性，用命令`chmod +x rc`修改其属性。 shell脚本文件开头`#!/bin/sh`并不是注释不可缺少。

motd文件不影响系统正常工作，由rc文件调用并且打印显示在终端上，motd意思是Message of today

- 在`/etc`目录下再创建init.d目录，并将 /etc/rc 向 /etc/init.d/rcS 做符号链接。此文件为 inittab 指定的启动脚本。
- 创建 `/dev` 目录, 并在该目录下建立必要的设备。创建设备的操作需要在映像中进行。
- 建立 `/proc` 空目录, 供 proc 文件系统使用。
- 建立`/sbin`空目录
- 建立`/mnt`空目录
- 建立 `/lib` 目录, 将交叉编译器链接库路径（`/opt/armhf-linux-2018.08/arm-none-linux-gnueabi/lib`）下的下面几个库复制到 lib 目录:

  `ld-2.25.so, libc-2.25.so, libm-2.25.so` 并做相应的符号链接

  ```
  ln -s ld-2.25.so ld-linux-armhf.so.3 #链接
  ln -s libc-2.25.so libc.so.6
  ln -s libm-2.25.so libm.so.6
  ```

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

文件系统目录至此构造完毕。

映像文件制作

文件系统格式的选择

主流的文件系统类型有

1. ext文件系统

   是Linux2.4版本的标准文件系统。ext2fs继承了扩展文件系统 (extfs)的一些优点，并且具有很好的稳定性、可靠性和健壮性。

   - 支持4TB上限的内存
   - 文件名称最长可达1024个字符
   - 创捷文件系统时，可设置页面大小。
   - 可实现快速符号链接，提高访问速度
2. NFS文件系统

   可以使文件实现共享，在不同的系统之间使用。
3. JFFS2文件系统

   基于FLASH的日志文件系统。掉电不影响文件的可靠性。支持数据压缩，提供损耗平衡支持，支持多种节点类型，降低内存损耗。但是当文件系统已满或者接近满时，由于垃圾收集问题会大大影响运行速度。
4. YAFFS2文件系统

   是专门针对 NAND 闪存设计的嵌入式文件系统，更多使用在大容量 NAND FLASH 之中。

考虑到我们是制作一个根文件系统，也就是说，这个根文件系统并不需要提供写入能力：在制作完成后根文件系统的内容并不需要更新。与此同时也不需要提供断电保护的功能。为了在内存空间有限的嵌入式系统中使用这个根文件系统，最好需要具备压缩的功能。综上所述，我们使用在Ramdisk上的ext2文件系统。

Ramdisk将一部分固定大小的内存当作分区使用，系统比较简单。由于是保存在内存之中，运行速度比较快，内容为只读属性，可以提高系统的性能。

Ramdisk的生成

在主机上创建一个空文件并且将之格式化成ext2fs的文件系统映像。

```
dd if=/dev/zero of=ramdisk_img bs=1k count=8192
# intput file /dev/zero这个设备专门用于创建虚拟文件
# output file  输出文件
# 每个block 为1k大小
# 总共大小为8M
mke3fs ramdisk_img #格式化
mount ramdisk_img #挂载
```

格式化产生文件夹`/lost+found`

挂载后目录位于`/mnt/ramdisk_img`，我们需要将我们在安装目录下创建的文件系统拷贝到该目录下。使用`cp`命令。由于我们拷贝了一些库进入文件系统中，所以我们生成的8M映像空间不够。

使用strip命令对这些库文件进行瘦身。

```
arm-linux-strip ld-2.25.so
arm-linux-strip libm-2.25.so
arm-linux-strip libc-2.25.so  #需要使用export改变环境变量才能直接使用这个命令
```

拷贝完成之后，我们可以在映像上的`/dev`文件夹下创建一些设备文件。

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

至此，我们的映像文件已经基本完成。

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

`umount ramdisk_img`之后，为了节省内存占用空间，我们需要进行压缩。

`gzip ramdisk_img`

将压缩后的映像文件转移到tfpt服务器即可。

至此我们的文件已经准备完毕。

```
setenv ipaddr 192.168.208.133
setenv serverip 192.168.208.12
tftp 0x82000000 zImage
tftp 0x88080000 ramdisk_img.gz
tftp 0x88000000 am335x-boneblack.dtb
set bootargs root=/dev/ram rw initrd=0x88080000 console=ttyO0,115200
bootz 0x82000000 0x88080000:0x8769d8 0x88000000 #以压缩方式载入文件
#自己版本内核大小0x 8769d8
```

系统正常启动

```
        ARM-LINUX WORLD
        BB -- BLACK
        by 13
================================

and this is a test! XD

Please press Enter to activate this console. ls -l
/ #
/ # ls -l
total 1866
drwxrwxr-x    2 1001     1001          3072 Nov 12  2019 bin
drwxr-xr-x    4 0        0             2640 Jan  1  1970 dev
drwxrwxr-x    3 1001     1001          1024 Nov 12  2019 etc
drwxrwxr-x    2 1001     1001          1024 Nov 12  2019 lib
-rwxr-xr-x    1 1001     1001       1877664 Nov 12  2019 linuxrc
drwx------    2 0        0            12288 Nov 12  2019 lost+found
drwxrwxr-x    2 1001     1001          1024 Nov 12  2019 mnt
dr-xr-xr-x   79 0        0                0 Jan  1  1970 proc
drwxrwxr-x    2 1001     1001          3072 Nov 12  2019 sbin
drwxrwxr-x    2 1001     1001          1024 Nov 12  2019 sys
drwxrwxr-x    4 1001     1001          1024 Nov 12  2019 usr
/ # [  149.443766] random: nonblocking pool is initialized
```

第一个实验到此完成。