**I.mx6驱动开发过程整理**

目录

[第1章 平台及环境搭建 2](#_Toc452391249)

[1.1 软硬件版本基本信息 3](#_Toc452391250)

[1.2 基础环境搭建 3](#_Toc452391251)

[1.2.1 Ubuntu14.04 3](#_Toc452391252)

[1.2.2 Yocto建立 3](#_Toc452391253)

[第2章 烧录软件的使用以及代码解析 5](#_Toc452391254)

[2.1 MFGTOOL V2快速使用 6](#_Toc452391255)

[2.1.1 使用方法 6](#_Toc452391256)

[图2.1 MFGtool设备的识别 7](#_Toc452391258)

[图2.2 MFGtool V2界面 7](#_Toc452391259)

[图2.3 烧写过程 8](#_Toc452391260)

[图2.4 烧写成功 8](#_Toc452391261)

[图2.5 串口终端参数设置 8](#_Toc452391262)

[2.2 MFGTOOL烧写脚本代码分析 9](#_Toc452391263)

[2.3 MFGTOOL完全烧写测试 12](#_Toc452391264)

[2.4单独烧录UBOOT测试 12](#_Toc452391265)

[第3章 挂载运行环境配置 15](#_Toc452391266)

[3.1 UBOOT环境配置 15](#_Toc452391267)

[3.2 主机TFTP、NFS设置 16](#_Toc452391268)

[3.2.1 TFTP配置 16](#_Toc452391269)

[3.2.2 NFS配置 17](#_Toc452391270)

[第4章 GPIO驱动 19](#_Toc452391271)

[4.1设备树（DTS）起源 19](#_Toc452391272)

[4.2 DTS、DTSI基本语法介绍 19](#_Toc452391273)

[4.3 GPIO驱动添加 21](#_Toc452391274)

[4.3.1 GPIO初始化引脚添加 21](#_Toc452391275)

[4.3.2 GPIO引脚在驱动中的使用 21](#_Toc452391276)

[4.3.3 /dev创建设备节点过程 22](#_Toc452391277)

[第5章 USB转串驱动安装 27](#_Toc452391278)

[5.1 USB引脚配置 27](#_Toc452391279)

[5.2 USB转串口驱动（PL2303） 28](#_Toc452391280)

[第6章 SD卡驱动验证 30](#_Toc452391281)

[6.1 SD卡驱动检测 30](#_Toc452391282)

[第7章 SATA硬盘驱动验证 31](#_Toc452391283)

[7.1 SATA硬盘内核配置 31](#_Toc452391284)

[7.2 SATA硬盘硬件接口定义 31](#_Toc452391285)

[7.3 SATA硬盘接上后实际效果 33](#_Toc452391286)

[第8章 can总线调试配置 34](#_Toc452391287)

# 第1章 平台及环境搭建

## 1.1 软硬件版本基本信息

硬件版本：I.mx6q-sabresd开发板

内核版本：Linux3.14.26

Uboot版本：2014.04

编译环境：Yocto

虚拟机版本：Ubuntu14.04

虚拟机硬盘分配大小：120GB

## 1.2 基础环境搭建

### 1.2.1 Ubuntu14.04

本文所阐述的开发过程，都是基于Ubuntu14.04的基础下进行开发的。因此，我们需要在个人电脑上安装一个虚拟机。因为我们所用的Yocto编译环境所需硬件较高，所以装上虚拟机并且运行系统后可能会造成个人电脑的运行速率下降。

对于Ubuntu系统在Windows中的使用，我们一般采用虚拟机的方式。首先，下载一个VMware Workstation发行版本，自行安装破解（网上有很多教程）。随后，在http://ftp.sjtu.edu.cn/ubuntu-cd/14.04/ （这是国内的官方镜像下载地址）可以找到Ubuntu14.04 的64位ISO镜像文件， 然后按步骤进行安装即可（Ubuntu-14.04.3-desktop-amd64.iso）。

1.2.2 Yocto建立

本节主要是介绍如何通过yocto建立i.mx6q开发板少些运行所需要的镜像文件（主要内容来自Freescale\_Yocto\_project\_User\_Guide.docx）。

建立前提是虚拟机能够连网。如若不能，查看虚拟机网络适配器的模式，桥接模式不行则改为NAT模式尝试。

环境要求及主要编译过程：

1. 虚拟机硬盘分配大于等于120GB。
2. Ubuntu版本高于或等于14.04。
3. Linux虚拟机所需要安装的软件工具
4. Sudo apt-get install gawk wgetgit-core diffstat unzip texinfo \

gcc-multilib build-essential chrpathsocat

1. Sudo apt-get install libsdl1.2-dev xtermsedcvs subversion coreutils \

texi2htimldocbook-utils python-pysqlite2 help2man make gcc g++ \

desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev mercurial \

autoconfautomakegroff curl lzopasciidoc

1. Sudo apt-get install u-boot-tools
2. 安装repo工具。Repo是基于git的文件管理工具，使yocto的项目工程管理更为便捷。
3. 在当前用户的home目录下建立一个bin文件。

mkdir ~/bin(如果存在则这步省略)；

curl <http://commondatastorage.googleapis.com/git-repo-downloads/repo>

> ~/bin/repo; (下载repo至~/bin文件夹，命名为repo)

Chmod a+x ~/bin/repo

1. 添加repo至当前的环境变量

export PATH=~/bin:$PATH

1. Yocto设置。
2. mkdir fsl-release-bsp
3. cd fsl-release-bsp
4. git config --global user.name “Your Name”
5. git config --global user.email “Your Email”
6. git config --list
7. repo init –u git://git.freescale.com/imx/fsl-arm-yocto-bsp.git –b imx-3.14.28-1.0.0\_ga
8. repo sync

当以上这些步骤完成之后，yocto源码包会被提取在fsl-release-bsp/sources这个文件夹下。可以使用repo sync 去获得最新的源码。

1. 在fsl-release-bsp文件夹下使用 source setup-environment 可以查看所有支持的开发板。

若要产生u-boot或zImage文件则需要使用命令:

source setup-environment <build-dir>

<build-dir>:是指定的编译目录，编译产生的uboot和zImage在该目录下的tmp/deploy/images/imx6qsabresd文件夹内。

也可以使用以下命令：

MACHINE=<machine name> source fsl-setup-release.sh –b <build-dir> -e <backend>

其中：-b <build-dir>为设置所要编译的目录。

-e<backend>为设置终端设备。默认为x11，可选项为x11、fb(Frame Buffer)、dfb(DirectFB)和wayland。

7. bitbake options

bitbake命令用来编译可执行文件，命令的格式为

bitbake<image name>

例如：bitbake fsl-image-qt5 (这个命令只生成文件系统)

bitbake还包括一些编译参数，用来实现特殊的编译需要，格式为：

bitbake<parameter><component>

参数列表如下：

-c fetch 若下载未完成则继续下载

-c cleanall清除整个编译文件夹

-c deploy 编译一个目标文件到文件系统

-k 继续编译即使有中断发生

-c compile –f 即使目标文件已经部署，也重新编译

-g 列出目标文件的依赖文件

-DDD 每添加一个D表示开启一层仿真调试程度。

8. U-BOOT配置

在默认情况下，u-boot是从SD卡启动的，若要从eMMC启动，则需要对u-boot的启动方式进行配置，配置方式如下：

MACHINE=imx6qsabresd

Bitbake–c deploy u-boot-imx

9. 编译适用于开发板的x11镜像文件

MACHINE=imx6qsabresd source fsl-setup-release.sh –b build-x11 –e x11

Bitbake fsl-image-qt5

10. 当编译完成后，创建的镜像文件在<build directory>/tmp/deploy/imges文件夹下。每次编译会产生：一个u-boot，一个内核，和一个开发板配置文件。

11. 编译产生的镜像文件烧入开发板的方式之一是使用Manufacturing Tool ，建立一个manufactruing镜像文件使用的命令如下：

bitbake fsl-image-mfgtool-initramfs

至此，整个工程环境就编译完成了，耗时略长。我们可以在我们的目录下查看编译完毕的各类文件。

下载所用镜像文件：/fsl-release-bsp/build-x11/tmp/deploy/images/imx6qsabresd

内核源码文件：/fsl-release-bsp/build-x11/tmp/work/imx6qsabresd-poky-linux-gnueabi/linux-imx/3.14.28-r0/git

# 第2章 烧录软件的使用以及代码解析

## 2.1 MFGTOOL V2快速使用

这个章节主要介绍如何使用官网下载的imx-3.14.28\_1.0.0\_ga-mfg-tools将uboot、内核和文件系统烧入imx6开发板，以及具体的烧入过程。烧入所需的镜像文件在build\_x11/tmp/deploy/images/imx6qsabresd文件夹内。

### 2.1.1 使用方法

1. 本节使用的系统为WIN7 64位系统。在使用MFGtool之前，请确保cfg.ini 和 UICfg.ini这两个文件和mfgtool.exe在同一层文件夹下。

cfg.ini的作用是确定烧入执行的具体文本以及传入该文本的参数。cfg.ini内容为：

[profiles]

chip = Linux /\*选择当前Profiles文件夹下的Linux文件夹\*/

[platform]

board = SabreSD/\*此参数当前版本无用\*/

[LIST]

name = eMMC /\*打开之前选择的Linux文件夹下的OS Firmware/ucl2.xml文件，选择list名为eMMC的代码段作为烧录执行代码。\*/

[variable]/\*定义烧录代码中的变量\*/

board = sabresd

mmc = 0

ldo=

UICfg.ini的作用是确定同时烧入的设备数量，支持1~4。

UICfg.ini的内容为

[UICfg]

PortMgrDlg=1/\*设置同时烧入的设备数量为1\*/

1. 将开发板上的拨码开关设置为烧写模式(所有的开关设置为OFF档)。将开发板上的USB OTG和USB TO UART两个mini USB接口接入电脑，前者是烧写串口，后者是调试串口。识别成功后，可以在电脑的设备管理窗口看到已添加的设备。

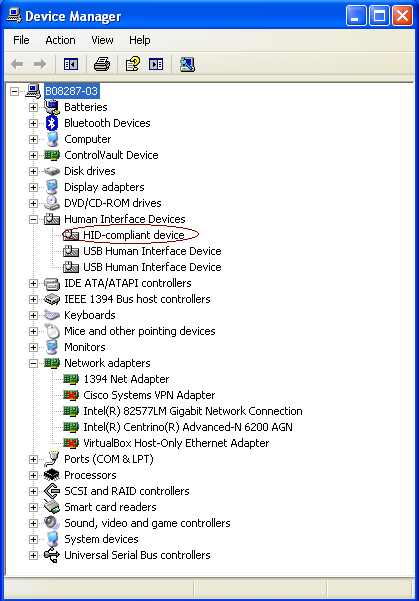


图2.1 MFGtool设备的识别

1. 双击mfgtools文件夹下的mfgtool2-yocto-mx6-sabresd-emmc.vbs(注意：请勿直接双击MfgTool2.exe)。在mfgtools文件夹下有很多.vbs文件，这些文件用VB编写，用于配置不同方式的烧入参数，用UE打开该vbs文件可查看具体的配置参数。这里选择emmc方式烧入。双击后会自动打开MfgTool2.exe。

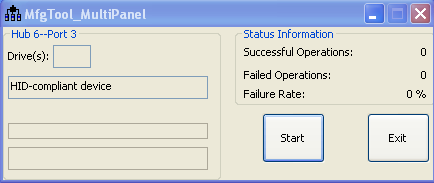


图2.2 MFGtool V2界面

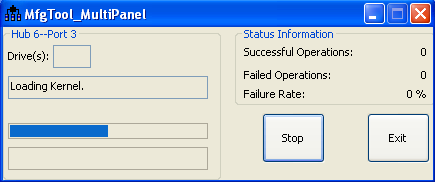


图2.3 烧写过程

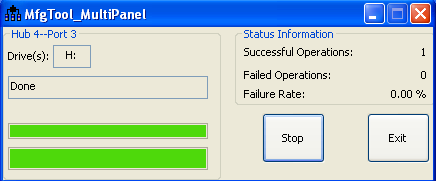


图2.4 烧写成功

1. 若开启串口终端，可以获得更多的调试信息。这里需要注意的是，串口终端的RTS/CTS选项需取消，否则在从eMMC启动时键盘无法使用。串口终端参数设置如图5。

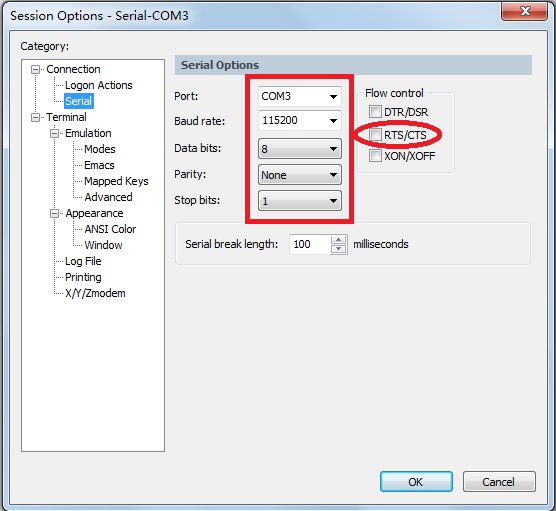


图2.5 串口终端参数设置

## 2.2 MFGTOOL烧写脚本代码分析

MFGtool的整个烧录过程是根据ucl2.xml文件中指定的代码段执行。以下的代码较原代码已删除了一些不必要的部分，由于本项目使用eMMC作为代码的存储器，因此这里针对eMMC的烧写过程进行分析。

<CFG>

<STATE name="BootStrap" dev="MX6Q" vid="15A2" pid="0054"/>

<STATE name="Updater" dev="MSC" vid="066F" pid="37FF"/>

</CFG>

<CFG></CFG>中的内容是对全局变量的配置，这里定义了设备是MX6Q，程序的执行阶段为“BootStrap”，连接至USB ，pid = 14335 ，vid = 1647 。bootStrap可以枚举多个设备，连接上USB后，程序会根据pid与vid自动选择设备。

STATE 字符声明了该段代码执行的阶段。

<LIST name="eMMC" desc="Choose eMMC as media">

这里的LIST name 与cfg.ini中的 [list] name 参数对应。

<CMDstate="BootStrap" type="boot" body="BootStrap" file ="firmware/u-boot-imx6q%board%\_sd.imx" ifdev="MX6Q">Loading U-boot</CMD>

Type 字符声明命令类型，type="boot"为下载file(uboot配置文件)到RAM

<CMD state="BootStrap" type="load" file="firmware/zImage" address="0x12000000"loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" ifdev="MX6Q">Loading Kernel.</CMD>

下载zImage到RAM，指定地址为0x12000000。loadSection：设置ROM编码参数，如果还有其他的镜像要下载，设置为“OTH”；如果是最后一个镜像则设置为“APP”；如果这个镜像中有包含flashheader，则设置为“TRUE”，否则为“FALSE”。

<CMDstate="BootStrap" type="load" file="firmware/fsl-image-mfgtool-initramfs-imx6qdlsolo.cpio.gz.u-boot" address="0x12C00000"loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" ifdev="MX6Q">Loading Initramfs.</CMD>

下载u-boot到RAM，指定地址为0x12C00000。

<CMDstate="BootStrap" type="load" file="firmware/zImage-imx6q-%board%%ldo%.dtb" address="0x18000000"loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" ifdev="MX6Q">Loading device tree.</CMD>

下载针对imx6q开发板的设备树到RAM，指定地址为0x18000000

<CMD state="BootStrap" type="jump" > Jumping to OS image. </CMD>

通知ROM跳转到RAM运行。

<!-- create partition -->

<CMD state="Updater" type="push" body="send" file="mksdcard.sh.tar">Sending partition shell</CMD>

声明命令类型为push：该命令执行者为目标板而非主机。Send：接收主机发送过来的“mksdcard.sh.tar”压缩包，并且把该压缩包的名字存入$FILE变量中。

<CMD state="Updater" type="push" body="$ tar xf $FILE "> Partitioning...</CMD>

解压缩“mksdcard.sh.tar”，这里的$是执行shell命令的意思。

<CMD state="Updater" type="push" body="$ sh mksdcard.sh /dev/mmcblk%mmc%"> Partitioning...</CMD>

运行mksdcard.sh脚本。

mksdcard.sh脚本内容为

{

#!/bin/sh

# partition size in MB

BOOT\_ROM\_SIZE=10

# call sfdisk to create partition table

# destroy the partition table

node=$1

dd if=/dev/zero of=${node} bs=1024 count=1

sfdisk --force -uM ${node} << EOF

${BOOT\_ROM\_SIZE},500,0c

600,,83

EOF

}

<!-- burn uboot -->

<CMD state="Updater" type="push" body="$ dd if=/dev/zero of=/dev/mmcblk%mmc% bs=1k seek=384 conv=fsync count=129">clear u-boot arg</CMD>

建立输出文件/dev/mmcblk%mmc%，其中的%mmc%中的参数对应于cfg.ini中的mmc。

##之后的代码为各文件的建立，目录挂载等动作，这里不再详细说明。

<!-- access boot partition -->

<CMD state="Updater" type="push" body="$ echo 0 > /sys/block/mmcblk%mmc%boot0/force\_ro">access boot partition 1</CMD>

<CMD state="Updater" type="push" body="send" file="files/u-boot-imx6q%board%\_sd.imx" ifdev="MX6Q">Sending u-boot.bin</CMD>

<CMD state="Updater" type="push" body="$ dd if=$FILE of=/dev/mmcblk%mmc%boot0 bs=512 seek=2">write U-Boot to sd card</CMD>

<CMD state="Updater" type="push" body="$ echo 1 > /sys/block/mmcblk%mmc%boot0/force\_ro"> re-enable read-only access </CMD>

<CMD state="Updater" type="push" body="$ echo 8 > /sys/block/mmcblk%mmc%/device/boot\_config">enable boot partion 1 to boot</CMD>

<!-- create fat partition -->

<CMD state="Updater" type="push" body="$ mkfs.vfat /dev/mmcblk%mmc%p1">Formatting rootfs partition</CMD>

<CMD state="Updater" type="push" body="$ mkdir -p /mnt/mmcblk%mmc%p1"/>

<CMD state="Updater" type="push" body="$ mount -t vfat /dev/mmcblk%mmc%p1 /mnt/mmcblk%mmc%p1"/>

<!-- burn zImage -->

<CMD state="Updater" type="push" body="send" file="files/zImage">Sending kernel zImage</CMD>

<CMD state="Updater" type="push" body="$ cp $FILE /mnt/mmcblk%mmc%p1/zImage">write kernel image to sd card</CMD>

<!-- burn dtb -->

<CMD state="Updater" type="push" body="send" file="files/zImage-imx6q-%board%%ldo%.dtb" ifdev="MX6Q">Sending Device Tree file</CMD>

<CMD state="Updater" type="push" body="$ cp $FILE /mnt/mmcblk%mmc%p1/imx6q-%board%.dtb" ifdef="MX6Q">write device tree to sd card</CMD>

<CMD state="Updater" type="push" body="$ umount /mnt/mmcblk%mmc%p1">Unmountingvfat partition</CMD>

<!-- burn rootfs -->

<CMD state="Updater" type="push" body="$ mkfs.ext3 -E nodiscard /dev/mmcblk%mmc%p2">Formatting rootfs partition</CMD>

<CMD state="Updater" type="push" body="$ mkdir -p /mnt/mmcblk%mmc%p2"/>

<CMD state="Updater" type="push" body="$ mount -t ext3 /dev/mmcblk%mmc%p2 /mnt/mmcblk%mmc%p2"/>

<CMD state="Updater" type="push" body="pipe tar -jxv -C /mnt/mmcblk%mmc%p2" file="files/rootfs.tar.bz2">Sending and writtingrootfs</CMD>

<CMD state="Updater" type="push" body="frf">Finishing rootfs write</CMD>

<CMD state="Updater" type="push" body="$ umount /mnt/mmcblk%mmc%p2">Unmountingrootfs partition</CMD>

<CMD state="Updater" type="push" body="$ echo Update Complete!">Done</CMD>

</LIST>

## 2.3 MFGTOOL完全烧写测试

综合第二部分，要烧入的文件有：

firmware/u-boot-imx6q%board%\_sd.imx

firmware/zImage

firmware/fsl-image-mfgtool-initramfs-imx6qdlsolo.cpio.gz.u-boot

firmware/zImage-imx6q-%board%%ldo%.dtb

files/u-boot-imx6q%board%\_sd.imx

files/zImage

files/zImage-imx6q-%board%%ldo%.dtb

我们通过yocto得到的镜像文件分为两个部分：一部分是通过

Bitbake–c deploy u-boot-imx

Bitbake fsl-image-qt5

命令得到的镜像文件，如u-boot-imx6qsabresd\_sd.imx等，另一部分通过

Bitbakefsl-imge-mfgtool-initramfs

得到的，带mfgtool的文件，如fsl-image-mfgtool-initramfs-imx6qdlsolo.cpio.gz.u-boot。

经测试，在firmware文件夹下的文件应使用yocto编译的带mfgtool的文件，在file文件夹下的文件是使用Bitbake fsl-image-qt5和Bitbake–c deploy u-boot-imx编译的文件。

烧录完成后选择eMMC启动方式:

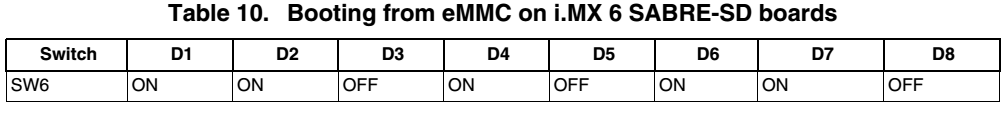


图2.6 eMMC启动拨码方式

## 2.4单独烧录UBOOT测试

在imx6开发板启动Linux内核之前，需要将一些必要的镜像文件（Uboot，Linux kernel，Device tree和Rootfs）复制到开发板的存储设备上，并设置为相应的启动方式，才能够正确启动。事实上，官方提供的开发工具MFGtool已经能够很好的解决整个系统的烧入问题，但是实际在开发的过程中，内核修改十分频繁，每一次修改就需要近10分钟的烧写（MFGtool烧写过程较慢），效率过低。因此，我们需要找一种更快的方式来烧录内核以及文件系统。Linux嵌入式开发用得最频繁的方式就是利用TFTP下载内核，然后通过NFS来挂载文件系统。但是，再做这些操作之前，我们需要达到一个前置条件，就是先将Uboot程序跑起来。本节介绍如何用MFGtool单独烧录Uboot。

系统使用uboot作为内核的引导系统，yocto编译生成的uboot镜像文件，可以通过MFGtool快速的单独烧入eMMC FLASH，但是需要对MFGtool的烧写脚本ucl2.xml进行一些修改：修改LIST name的名字；删除“<!-- burn zImage -->”及以下的部分，如图1所示。此外还需要对mfgtool2-yocto-mx6-sabresd-emmc.vbs中选择LIST name的参数做修改。烧入完成之后，拨SW6开关，重新启动就可以进入uboot了。

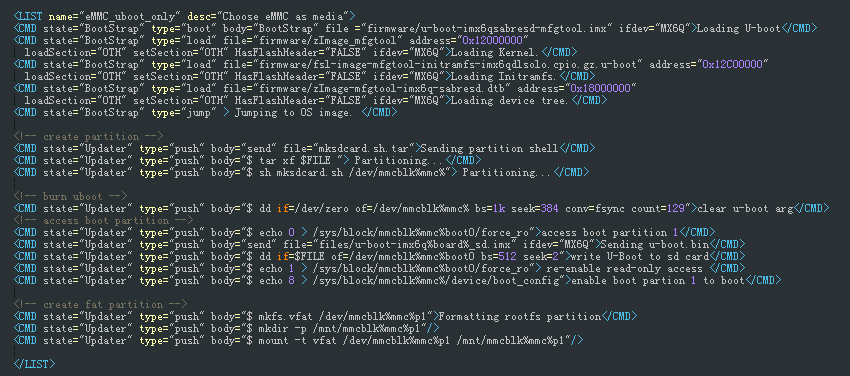


图2.7 单独烧录UBOOT的脚本

成功启动uboot时，界面参考图2.8。

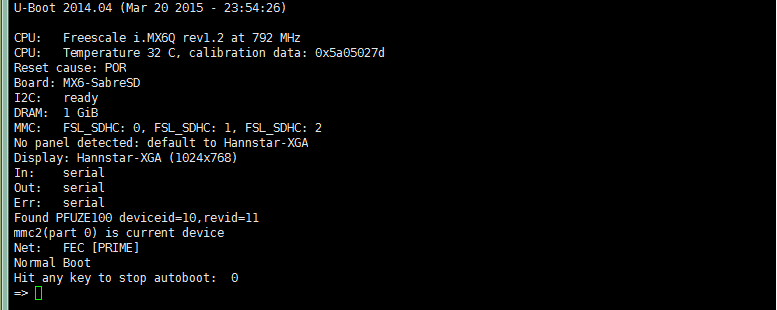


图2.8 UBOOT启动后等待界面

# 第3章 挂载运行环境配置

## 3.1 UBOOT环境配置

上一个章节的最后，学习了如何单独下载UBOOT，下载成功后也可以单独运行后。我们在图2.8的界面，输入env print或者printenv，这条命令是将uboot的已配置环境参数都打印出来，方便我们查看。找到其中开头为“netboot”的那一段，此段就是通过网络启动内核的命令序列。

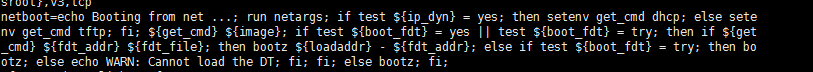


图3.1 网络启动内核命令截图

接下来就是设置uboot网络启动的一些参数。

1.首先尝试着PING一下我们的虚拟机（注意此时需要把网络模式切换成桥接模式）或者服务器。

Uboot=>ping 192.168.52.22

假设我们的服务器IP地址就是这个。目前来说也确实如此。如果正确连接，会显示alive。如果连接不上，需要检查网络或者设置IP，设置IP的命令为setenv ipaddr xxx.xxx.xxx。

设置完成后，如果还是不能ping通，有可能是网关的问题（如果不在同一网段需要有网关），我们需要对默认网关进行设置，命令为 setenv gatewayip 192.168.69.1（假设是69网段）。然后再次尝试PING主机或服务器，如若还是不行，那么很大可能是防火墙问题，或者硬件问题，需要逐一排查。

2.设置服务器主机IP（我们所需要的内核以及文件系统的存放主机）。

Uboot=>setenv serverip 192.168.52.22（也可以是自己虚拟机IP）

这个主机中需要相对应的安装TFTP以及NFS，具体方法后面章节介绍。

3.设置需要下载的内核文件名

Uboot=>setenv bootfile zImage

Uboot=>setenv Loadaddr 0x12000000 内核加载地址

文件名必须要与服务器主机中存放的文件名相同。

4.设置需要下载的设备树文件名

Uboot=>setenv fdt\_file zImage-imx6q-sabresd.dtb

Uboot=>Fdt\_addr 0x18000000 设备树加载地址

同上，文件名必须相同，并且主机中含有这个文件。

5.设置挂载的根文件系统在主机服务器上的路径。

Uboot=>setenv nfsroot /home/rootfs

这里需要注意，rootfs中的内容是Yocto目录中的

fsl-release-bsp/build-x11/tmp/deploy/images/imx6qsabresd/fsl-image-qt5-imx6qsabresd.tar.bz2

压缩包解压而来。

都设置完成后，输入Uboot=>save保存参数。

到此为止，Uboot的各个参数设置已经完成，其余的使用默认设置即可。

## 3.2 主机TFTP、NFS设置

在Ubuntu14.04中安装TFTP过程如下，在执行下列指令前，确保虚拟机或者服务器可以上网。

### 3.2.1 TFTP配置

1. 安装

$ apt-get install tftp-hpa tftpd-hpa

2. 建立目录

$ mkdir /tftpboot （这是建立tftp传输目录）

$ sudo chmod 0777 /tftpboot

$ sudo touch test.txt （test.txt文件最好输入内容以便区分，创建一个测试文件便于后面自测）

3. 配置

$ sudo vim /etc/default/tftpd-hpa

TFTP\_USERNAME="tftp"

TFTP\_DIRECTORY="/tftpboot" （这里是你的tftpd-hpa的服务目录,这个想建立在哪里都行）

TFTP\_ADDRESS="0.0.0.0:69"

TFTP\_OPTIONS="-l -c -s" （这里是选项,-c是可以上传文件的参数，-s是指定tftpd-hpa服务目录，上面已经指定）

4. 重启服务

$ sudo service tftpd-hpa restart

启动服务，这里要注意，采用的独立服务形式。

5. 测试

测试前提是在你之前设置的TFTP目录下拥有这个test.txt文件，127.0.0.1是主机回环IP地址。

$ tftp 127.0.0.1

tftp>get test.txt

tftp>q

当前目录下如果出现test.txt文件，则表明TFTP设置完成。

### 3.2.2 NFS配置

1.安装所需文件

# apt-get install nfs-common nfs-kernel-server

2.建立要分享以及挂载的目录

# mkdir /home/rootfs

3.修改配置参数

# vim /etc/exports

在最后添加  
/home/rootfs \*(rw,sync, no\_root\_squash ,no\_subtree\_check)

\*：允许所有的网段访问，也可以使用具体的IP  
rw：挂接此目录的客户端对该共享目录具有读写权限  
sync：资料同步写入内存和硬盘  
no\_root\_squash：root用户具有对根目录的完全管理访问权限。  
no\_subtree\_check：不检查父目录的权限。

4.启动NFS Server

/etc/init.d/nfs-kernel-server start

至此，所有前期工作已经设置完毕。可以利用uboot启动观察是否能够正常的TFTP下载内核并启动，然后正常的NFS挂载文件系统。如果可以，那么恭喜你，可以进入驱动的开发学习了。

# 第4章 GPIO驱动

## 4.1设备树（DTS）起源

Linux内核版本在3.X之后开始使用设备树文件来取代之前在arch/arm/plat-x xx以及arch/arm/mach-xxx的文件来描述板级细节。利用设备树文件的方式可以避免大量的冗余代码，并且能够为后面内核的移植提供很大的便利，放到一个新的硬件环境时，只要更改设备树文件即可完成移植，大大加快开发速度。

Device Tree由一系列被命名的结点（node）和属性（property）组成，而结点本身可包含子结点。所谓属性，其实就是成对出现的name和 value。在Device Tree中，可描述的信息包括（原先这些信息大多被hard code到kernel中）：

* CPU的数量和类别
* 内存基地址和大小
* 总线和桥
* 外设连接
* 中断控制器和中断使用情况
* GPIO控制器和GPIO使用情况
* Clock控制器和Clock使用情况

它基本上就是 画一棵电路板上CPU、总线、设备组成的树，Bootloader会将这棵树传递给内核，然后内核可以识别这棵树，并根据它展开出Linux内核中的 platform\_device、i2c\_client、spi\_device等设备，而这些设备用到的内存、IRQ等资源，也被传递给了内核，内核会将这些资源绑定给展开的相应的设备。

## 4.2 DTS、DTSI基本语法介绍

DTS的基本语法范例，如图3-1 所示。

它包括一系列节点，以及描述节点的属性。

“/”为root节点。在一个.dts文件中，有且仅有一个root节点；在 root节点下有“node1”，“node2”子节点，称root为“node1”和“node2”的parent节点，除了root节点外，每个节点 有且仅有一个parent；其中子节点node1下还存在子节点“child-nodel1”和“child-node2”。

注：如果看过内核/arch/arm/boot/dts目录的读者看到这可能有一个疑问。在每个.dsti和.dts中都会存在一个“/”根节点，那么如果在一个设备树文件中include一个.dtsi文件，那么岂不是存在多个“ /”根节点了么。其实不然，编译器DTC在对.dts进行编译生成dtb时，会对node进行合并操作，最终生成的dtb只有一个root node。 Dtc会进行合并操作这一点从属性上也可以得到验证。这个稍后做讲解。

在节点的{}里面是描述该节点的属性（property），即设备的特性。它的值是多样化的：

1.它可以是字符串string，如①；也可能是字符串数组string-list，如②

2.它也可以是32 bit unsigned integers，如cell⑧，用<>表示

3.它也可以是binary data，如③，用[]表示

4.它也可能是空，如⑦

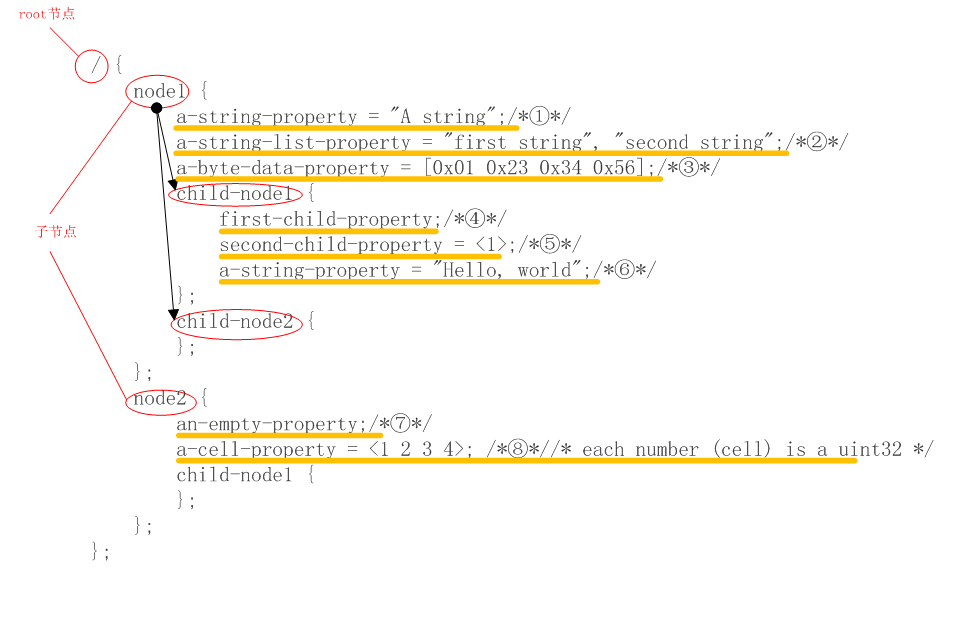


图4.1  DTS的基本语法范例

Yocto工程的DTS以及DTSI文件所在目录如下：

/fsl-release-bsp/build-x11/tmp/work/imx6qsabresd-poky-linux-gnueabi/linux-imx/3.14.28-r0/git \arch\arm\boot\dts

更多有关于设备树文件的语法以及格式需要自己去边查看实际文件边学习，这里就不一一列举了。

## 4.3 GPIO驱动添加

### 4.3.1 GPIO初始化引脚添加

在GPIO驱动编写之前，应该先对需要控制的引脚进行初始化。此次我们需要更改的设备树文件为imx6qdl-sabresd.dtsi。

打开该文件，找到iomux节点中的pinctrl\_hog:hoggrp子节点，里面添加初始化如：

MX6QDL\_PAD\_NANDF\_D0\_\_GPIO2\_IO00 0x800000

这句话的含义为将GPIO2.0这个引脚设置为输入输出功能，其中MX6QDL\_PAD\_NA NDF\_D0\_\_GPIO2\_IO00可以在同样的目录下的imx6q-pinfun c.h中找到宏定义：

#define MX6QDL\_PAD\_NANDF\_D0\_\_GPIO2\_IO01 0x284 0x66c 0x000 0x5 0x0

对应的各个数值的含义为：

<mux\_reg conf\_reg input\_reg mux\_mode input\_val>

0x80000000 表示no pin config need，用本身默认的配置。

### 4.3.2 GPIO引脚在驱动中的使用

以上操作完成了对一个引脚的IO初始化，可以在驱动代码中这样使用：

#define IMX\_GPIO\_NR(bank, nr) (((bank) - 1) \* 32 + (nr))

#define GPIO2\_01 IMX\_GPIO\_NR(2, 01)

gpio\_request\_one(GPIO2\_01, GPIOF\_DIR\_OUT, "label");

之后就可以通过gpio\_set\_value(GPIO2\_01,1)、gpio\_get\_value(GPIO2\_01)函数来对引脚进行操作。我个人不喜欢这种控制方式，因为将GPIO初始化放在刚刚这个节点，不利于DTS文件的阅读与维护。

因此，在iomux中初始化后的引脚，可以在该文件中创建一个子节点来控制。例如：

gpios{

compatible = "input-output";

dev-output{

output0 = <&gpio2 1 0>;

}

}

其中，这个节点设置完毕后可以在驱动代码中这样使用。

struct device\_node \*np;

int sg\_iGpio

np = of\_find\_node\_by\_path("/gpios/ dev-output ");//通过节点地址来申请IO

sg\_iGpio = of\_get\_named\_gpio(np, "output0", 0);

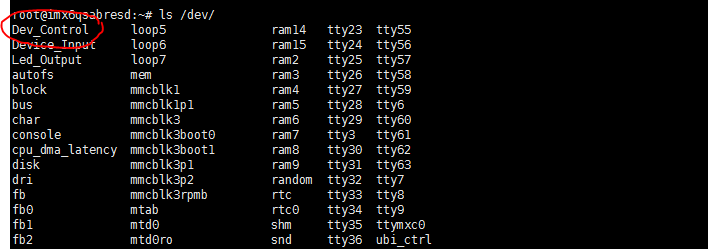
gpio\_is\_valid(sg\_iGpio);

gpio\_request\_one(sg\_iGpio, GPIOF\_DIR\_OUT, "label");

之后就可以通过gpio\_set\_value(GPIO2\_01,1)、gpio\_get\_value(GPIO2\_01)函数来对引脚进行操作。

### 4.3.3 /dev创建设备节点过程

1.目标：在/dev下生成一个设备节点：Dev\_Control



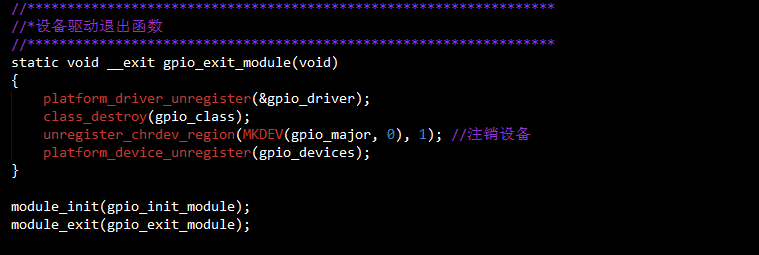
2.首先创建一个c源文件，并且命名随意，但是尽量贴近该驱动的功能。例如，我的文件起名为dev\_control.c 并把该文件放入源代码中的如下目录：/fsl-release-bsp/build-x11/tmp/work/imx6qsabresd-poky-linux-gnueabi/linux-imx/3.14.28-r0/git \drivers\char

放在driver驱动文件夹中的char文件夹是因为我们所创建的IO控制驱动属于字符设备。

3.紧接着对我们所创建的驱动文件进行代码的编写。

(1).首先是驱动的入口函数



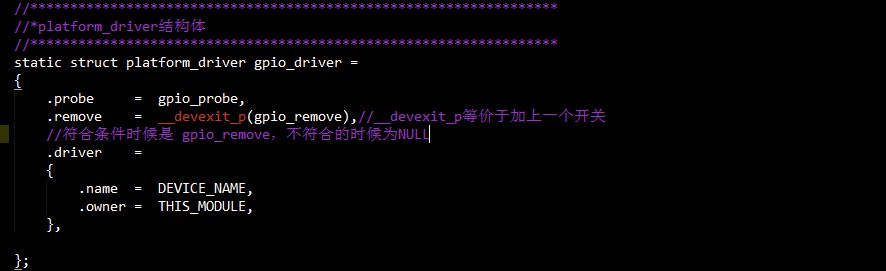


所有的驱动会有一个入口以及退出函数，当我们拿到一份代码的时候也可以从这个入口开始跟踪，一步一步解析。其中初始化出现的很频繁的一个常量是DEVICE\_ NAME,这个其实就是我们最终在/dev下创建设备节点所显示的名字。可在代码的其实部分加上宏定义

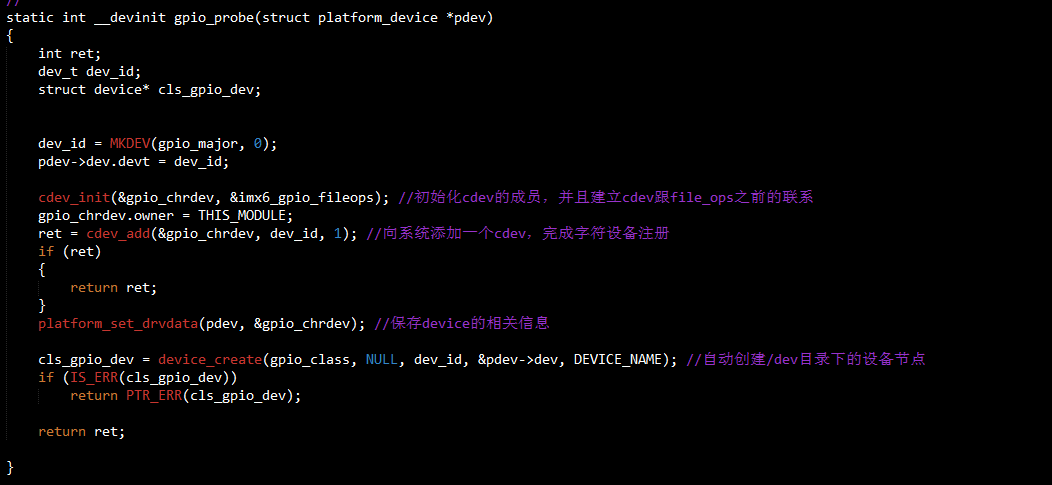
8.PNG

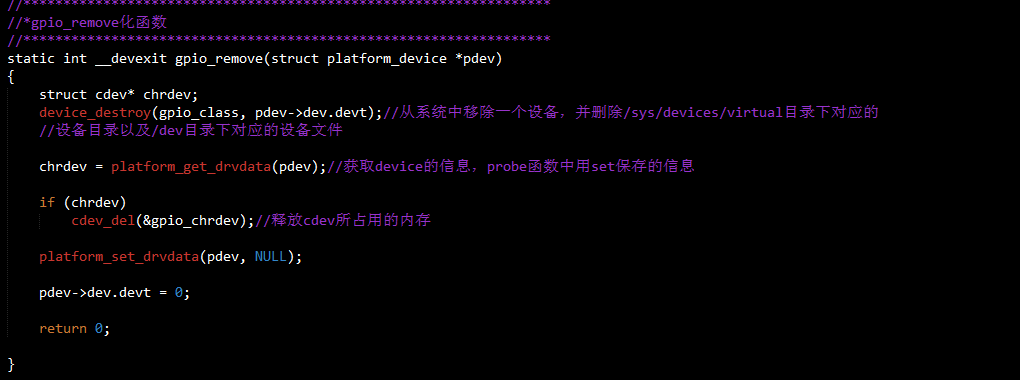
(2). 紧接着就是平台驱动platform\_driver的结构体的申请了，由于我们引脚部分的初始化通过DTS的方式，所以这里的驱动并没有平台设备platform\_device

的存在，也不需要。

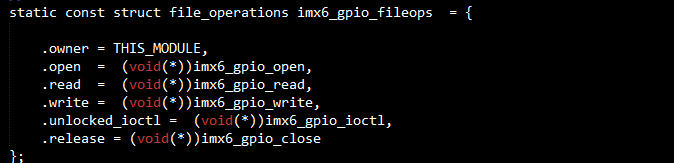


结构体当中声明了几个回调函数，其中驱动注册的时候调用的是probe，移除的时候调用remove,而driver中则是声明了驱动信息。

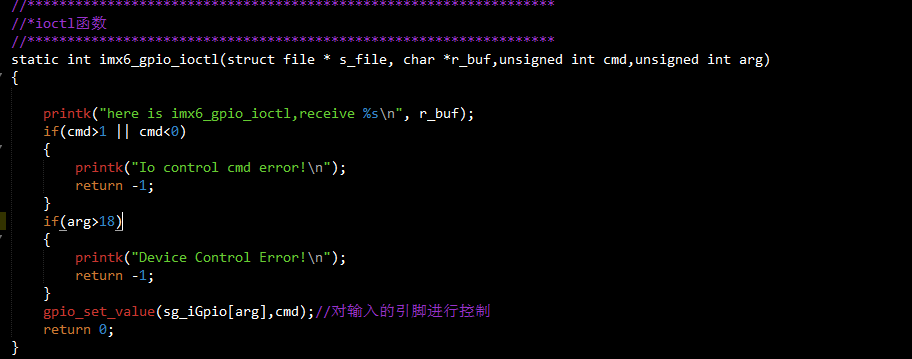


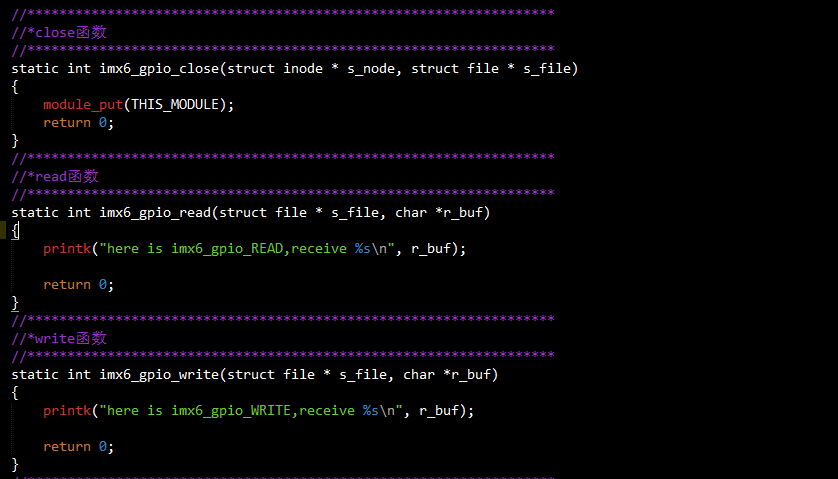


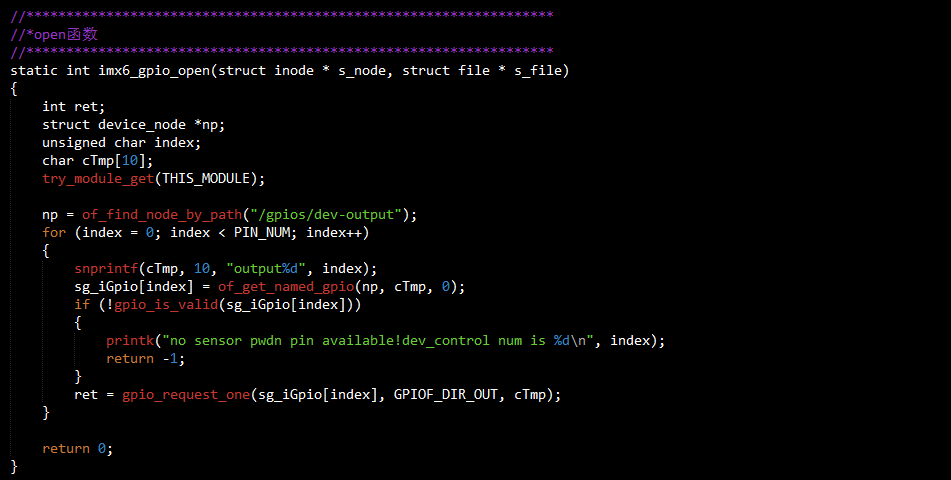
(3).前面都是在阐述底层的一些注册、移除方面的代码。那我们如何在应用层对我们的驱动进行控制呢？说白了，如何对IO口进行操作。我们都知道，在应用中，我们只要先open（/dev/xxx）然后在调用read()或则write()、ioctol()等就可以完成一次动作。但是，这是如何实现的呢？其实这些都是通过file\_operations联系到一块儿的。具体过程可自行了解。



其中这些回调函数分别是







其中，open()函数表示在调用设备驱动前进入的初始化函数，所以我们将一些对IO口的初始操作都放在这个函数中。这些语句在前文已经有过介绍，就不一一阐述了。

(4).代码写好后，我们需要在.config配置文件以及Makefile中加入我们刚刚添加的代码，这样最终才可以编译进入内核当中。先进入

fsl-release-bsp/build-x11/tmp/work/imx6qsabresd-poky-linux-gnueabi/linux-imx/3.14.28-r0/git/drivers/char目录中，然后编辑Kconfig文件，再该文件最后添加

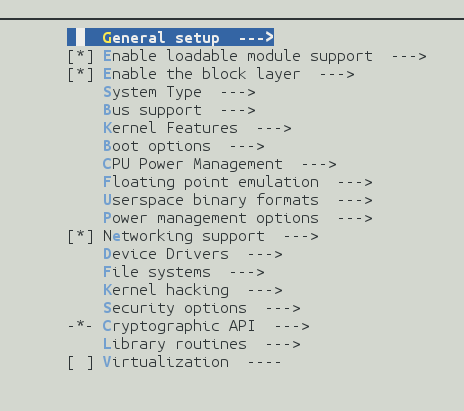
config DEV\_CONTROL\_DRIVER

tristate "DEV Control Driver"

保存并退出，然后输入指令

bitbake linux-imx -c menuconfig

这条指令相当于make menuconfig，进入内核配置窗口，它的最终结果是保存在源文件中的.config配置文件中，所以也可以直接去该文件修改，但是不推荐初学者这样使用。



进入Device Drivers-> Character devices，然后找到刚刚添加的DEV Control Driver，选中<\*>，然后保存并退出。

接近着更改Makefile文件，使用命令 vim Makefile，进入最底下，加入这条语句

obj-$(CONFIG\_DEV\_CONTROL\_DRIVER) += dev\_control.o

其中CONFIG\_DEV\_CONTROL\_DRIVER是根据我们在Kconfig中添加的名称编写的，另外dev\_control.o也是与我们创建的驱动文件名字相同。

这样都添加完毕以后，就可以分别敲指令

bitbake -c compile -f linux-imx

bitbake -c deploy -f linux-imx 进行编译，最后挂载运行即可得到我们要的结果了。

# 第5章 USB转串驱动安装

由于USB驱动一般都是由芯片厂商进行维护，并且内核源码相对复杂，就不在源码上进行过多阐述了。这里介绍有关于USB在DTS中的配置以及USB转串口驱动的linux中的安装。

## 5.1 USB引脚配置

reg\_usb\_otg\_vbus: regulator@0 {

compatible = "regulator-fixed";

reg = <0>;

regulator-name = "usb\_otg\_vbus";

regulator-min-microvolt = <5000000>;

regulator-max-microvolt = <5000000>;

gpio = <&gpio3 22 0>;

enable-active-high;

};

reg\_usb\_h1\_vbus: regulator@1 {

compatible = "regulator-fixed";

reg = <1>;

regulator-name = "usb\_h1\_vbus";

regulator-min-microvolt = <5000000>;

regulator-max-microvolt = <5000000>;

gpio = <&gpio1 29 0>;

enable-active-high;

};

一个是描述OTG，另外一个描述HOST，其中的两个gpio的作用是为了让接口商的VBUS有5V电压。

pinctrl\_usbotg: usbotggrp {

fsl,pins = <

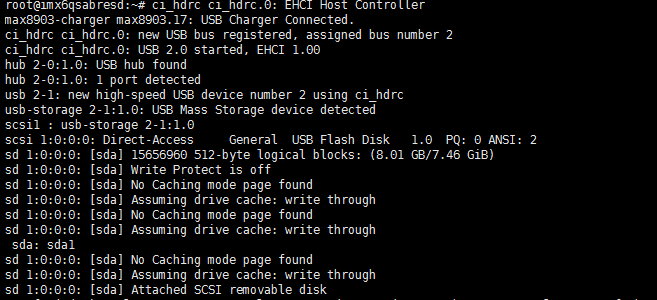
MX6QDL\_PAD\_ENET\_RX\_ER\_\_USB\_OTG\_ID 0x17059

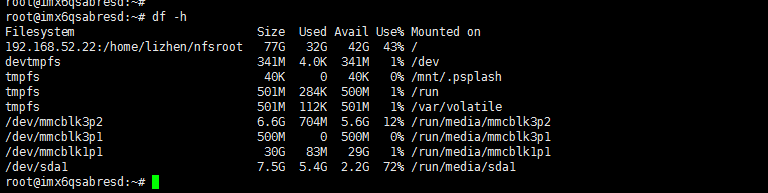
>;

};

另外这个GPIO的初始化是为了区别OTG的HOST跟DEVICE模式。

这里需要额外声明的一点是，这个USB口为什么不用配置D+ D-两个引脚呢？那是因为这两个USB所用的脚位都是没有复用功能的，也就是说他们只能作为USB的D+ D-来用，所以不用在DTS文件中配置。

当我们插上U盘时，可以在调试串口看到这样的信息：然后输入指令 df -h，可以看到USB的挂载情况以及大小。



可看到U盘的大小为7.5G，挂载在/dev/sda1目录下。

## 5.2 USB转串口驱动（PL2303）

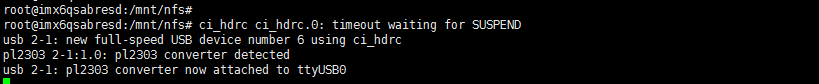
i.mx6芯片上的UART口只有4个，如果我们队UART口的需要远超出这个数值的时候，我们可以找其他方法来代替。其中，利用USB的HUB扩展，然后实现USB转UART是一个很好的方法，这样做的话最大可以额外扩充127个UART口。具体的USB转UART实现过程如下：

(1).首先，在已经配置好的环境下输入命令bitbake linux-imx -c menuconfig来打开配置窗口Device Drivers->USB support->USB Serial Converter support->USB Prolific 2303 Single Port Serial Driver<M>,然后保存退出并编译。

编译完成后，我们并不能在内核中到我们需要的驱动，这是因为我们选择的是<M>，表示编译成模块形式。我们能在源代码目录

fsl-release-bsp/build-lizhen/tmp/work/imx6qsabresd-poky-linux-gnueabi/linux-imx/3.14.28-r0/git/drivers/usb/serial中找到pl2303.ko文件，复制到我们板子中（可以通过挂载），然后在板子的该文件目录下执行 insmod pl2303.ko。如果未提示安装出错则表示安装成功。（若安装不上，则需要先安装usbserial.ko）

紧接着，我们可以直接插上USB转UART的转换线来尝试一下，注意转换线内要使用PL2303的芯片才能被识别。



由上图可以看到设备接上后，创建了一个节点ttyUSB0，我们就可以当成串口使用了。

至此，我们的USB转串的驱动就安装成功了。

# 第6章 SD卡驱动验证

## 6.1 SD卡驱动检测

SD卡驱动源代码较多，可以根据自己的需要去学习查看。这里只介绍SD卡的引脚DTS配置以及SD卡插上后的检测情况。

pinctrl\_usdhc2: usdhc2grp {

fsl,pins = <

MX6QDL\_PAD\_SD2\_CMD\_\_SD2\_CMD 0x17059

MX6QDL\_PAD\_SD2\_CLK\_\_SD2\_CLK 0x10059

MX6QDL\_PAD\_SD2\_DAT0\_\_SD2\_DATA0 0x17059

MX6QDL\_PAD\_SD2\_DAT1\_\_SD2\_DATA1 0x17059

MX6QDL\_PAD\_SD2\_DAT2\_\_SD2\_DATA2 0x17059

MX6QDL\_PAD\_SD2\_DAT3\_\_SD2\_DATA3 0x17059

MX6QDL\_PAD\_NANDF\_D4\_\_SD2\_DATA4 0x17059

MX6QDL\_PAD\_NANDF\_D5\_\_SD2\_DATA5 0x17059

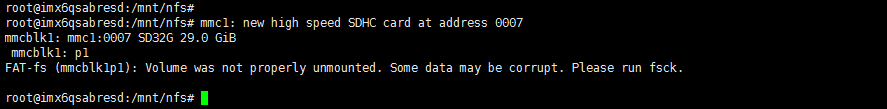
MX6QDL\_PAD\_NANDF\_D6\_\_SD2\_DATA6 0x17059

MX6QDL\_PAD\_NANDF\_D7\_\_SD2\_DATA7 0x17059

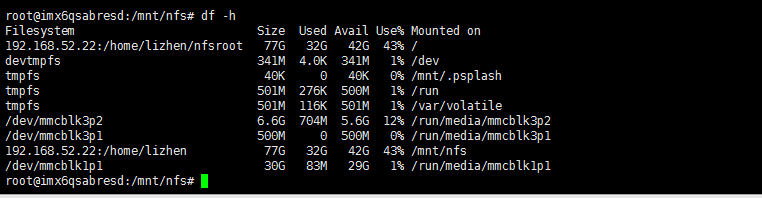
>;

};

当我们插上SD卡时，调试串口打印这样的信息



同USB一样，我们可以通过指令df -h查看挂载情况

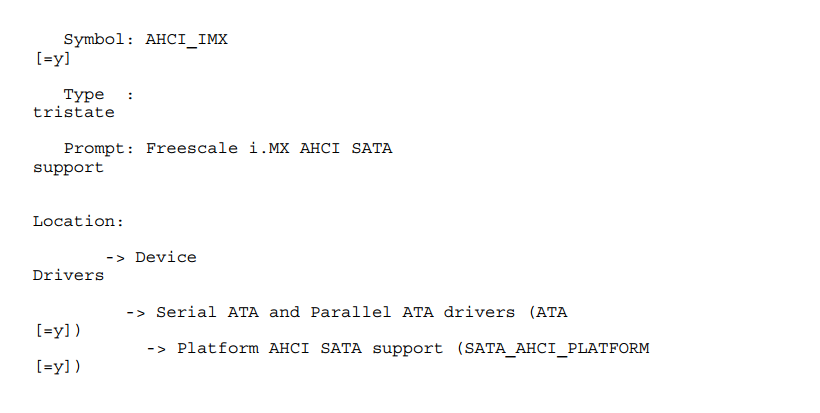


可以看出SD卡挂载在/run/media/mmcblk1p1。

# 第7章 SATA硬盘驱动验证

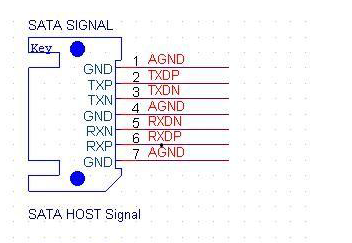
## 7.1 SATA硬盘内核配置

接上硬盘测试之前，首先要确保内核有关于SATA驱动的选项有选中。参考i.MX6 Linux® Reference Manual中的SATA Driver章节，可以得知驱动源码的位置以及内核配置选项。



## 7.2 SATA硬盘硬件接口定义

SATA接口分为数据接口以及电源接口，其中数据接口定义为：



1 GND ground(接地，一般和负极相连)

2 A+ Transmit(数据发送正极信号接口)

3 A- Transmit(数据发送负极信号接口)

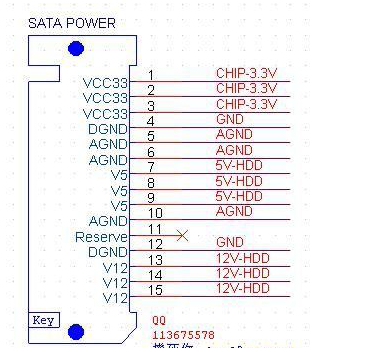
4 GND ground(接地，一般和负极相连)

5 B- Receive(数据接收负极信号接口)

6 B+ Receive(数据接收正极信号接口)

7 GND ground(接地，一般和负极相连)

电源接口定义为：



1 V33 3.3v Power （直流3.3V 正极电源针脚）

2 V33 3.3v Power （直流3.3V 正极电源针脚）

3 V33 3.3v Power, Pre-charge, 2nd mate （直流3.3V 正极电源针脚,预充电,与第二路配对）

4 Ground 1st Mate （接地，一般和负极相连,与第1 路配对）

5 Ground 2st Mate （接地，一般和负极相连,与第2 路配对）

6 Ground 3st Mate （接地，一般和负极相连,与第3 路配对）

7 V5 5v Power, pre-charge, 2nd mate （直流5V 正极电源针脚,预充电, 与第二路配对）

8 V5 5v Power （直流5V 正极电源针脚）

9 V5 5v Power （直流5V 正极电源针脚）

10 Ground 2nd Mate （接地，一般和负极相连,与第2 路配对）

11 Reserved – 保留的针脚

12 Ground 1st Mate （接地，一般和负极相连,与第1 路配对）

13 V12 12v Power, Pre-charge, 2nd mate （直流12V 正极电源针脚,预充电,与第二路配对）

14 V12 12v Power （直流12V 正极电源针脚）

15 V12 12v Power （直流12V 正极电源针脚）

电源颜色定义：

1 黄色-直流12V 正极

2 黑色-直流12V 负极(GND)

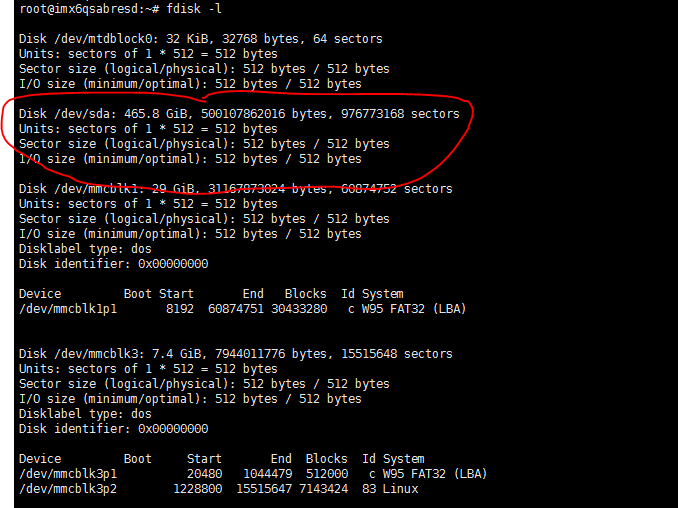
3 红色-直流5V 正极(GND,VSS)

4 黑色5V 负极(GND)

5 橙色-直流3.3V 正极

## 7.3 SATA硬盘接上后实际效果

启动后，输入fdisk -l指令，可以看到如下图所示：



可以看出，硬盘可用大小为465.8G，并且在/dev下的sda，可以通过指令

$mkfs.ext2 /dev/sda

$mkfs.vfat /dev/sda

来对硬盘进行初始化，并且确定文件格式。

通过指令 $mount -t ext2 /dev/sda /mnt/sda 进行挂载。

至此，我们就可以在/mnt/sda目录下操作硬盘了。

# 第8章 can总线调试配置

**1.在imx6qdl-sabresd.dtsi中添加代码：**

&flexcan1{

pinctl-names ="default";

pinctrl-0 = <&pinctrl\_flexcan1>;

trx-stby-gpio = <&gpio4 5 GPIO\_ACTIVE\_LOW>;

status = "okay";

};

pinctrl\_flexcan1:flexcan1grp {

fsl,pins = <

MX6QDL\_PAD\_GPIO\_7\_\_FLEXCAN1\_TX 0x80000000

MX6QDL\_PAD\_GPIO\_8\_\_FLEXCAN1\_RX 0x80000000

MX6QDL\_PAD\_GPIO\_19\_\_GPIO4\_IO05 0x80000000 //stby引脚看收发器是否有需要才配置

>;

};

在开发过程中，MX6QDL\_PAD\_GPIO\_7\_\_FLEXCAN1\_TX、MX6QDL\_PAD\_GPIO\_8\_\_FL EXCAN1\_RX放在这个位置一直不能成功配置，将其转移到IO配置位置时，就能正确配置。在内核中，可以通过readl()、writel()来读取寄存器的数值。

eg: readl(ioremap(0x020e0610,0x32));

writel(0x03, ioremap(0x020e0610,0x32));

**2.随后将menuconfig中的**

CONFIG\_CAN

CONFIG\_CAN\_RAW

CONFIG\_CAN\_BCM

CONFIG\_CAN\_VCAN

CONFIG\_CAN\_FLEXCAN

全部设置成y，然后编译下载内核并运行。

**3.启动后，打印的消息为：**

root@imx6qsabresd:~# dmesg | grep can

[ 1.464731] ahci-imx 2200000.sata: SSS flag set, parallel bus scan disabled

[ 1.512471] vcan: Virtual CAN interface driver

[ 1.518647] 2090000.can supply xceiver not found, using dummy regulator

[ 1.524700] flexcan 2090000.can: gpr /soc/aips-bus@02000000/iomuxc-gpr@020e0000 req\_gpr 0x34 req\_bit 28 ack\_gpr 0x10 ack\_bit 17

[ 1.524761] flexcan 2090000.can: device registered (reg\_base=c0ad8000, irq=142)

[ 4.671542] can: controller area network core (rev 20120528 abi 9)

[ 4.679692] can: raw protocol (rev 20120528)

[ 4.682666] can: broadcast manager protocol (rev 20120528 t)

[ 4.687055] can: netlink gateway (rev 20130117) max\_hops=1

**4.使用指令：ip link set can0 up type can bitrate 125000 来配置can0的波特率**

root@imx6qsabresd:~# ip link set can0 up type can bitrate 125000

flexcan 2090000.can can0: writing ctrl=0x0e312005

ip link set can3 up type can bitrate 125000

**5.然后启动can0： ifconfig can0 up**

调用ifconfig can0 可以得到如下信息：

root@imx6qsabresd:~# ifconfig can0

can0 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00

UP RUNNING NOARP MTU:16 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:10

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:142

**6.最后发送一条数据：**

root@imx6qsabresd:~# cansend can0 001#1122334455667788

如果一切配置正常，能够用示波器看到波形，并且can设备能够接收到数据。

# 第9章 SPI总线调试配置

9.1 设备树文件配置

①与上述其他功能类似，想要使用SPI功能之前，首先需要通过DTS文件对PIN进行初始化。如对SPI总线第一路进行初始化：

pinctrl\_ecspi2:ecspi2grp{

fsl,pins =<

MX6QDL\_PAD\_DISP0\_DAT17\_\_ECSPI2\_MISO 0x100b1

MX6QDL\_PAD\_DISP0\_DAT16\_\_ECSPI2\_MOSI 0x100b1

MX6QDL\_PAD\_DISP0\_DAT19\_\_ECSPI2\_SCLK 0x100b1

MX6QDL\_PAD\_DISP0\_DAT18\_\_GPIO5\_IO12 0x80000000 /\*SS0\*/

MX6QDL\_PAD\_DISP0\_DAT15\_\_GPIO5\_IO09 0x80000000 /\*SS1\*/

>;

};

其中后面两个引脚设置为GPIO功能 用作片选信号。其中需要注意的是检查文件其他位置是否有对所用引脚进行设置功能的语句，防止重复设置，导致出错。

②引脚配置完成后，需要对SPI总线所接设备进行更进一步的详细配置。参考配置如下：

&ecspi2 {

fsl,spi-num-chipselects = <2>;

cs-gpios = <&gpio5 12 0>,

<&gpio5 9 0>;

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_ecspi2>;

status = "okay";

sc16is752\_1@0 {

compatible = "sc16is752\_1";

spi-max-frequency = <4000000>;

reg = <0>;

interrupt-parent = <&gpio1>;

interrupts = <5 2>;

};

sc16is752\_2@1 {

compatible = "sc16is752\_2";

spi-max-frequency = <4000000>;

reg = <1>;

interrupt-parent = <&gpio1>;

interrupts = <16 2>;

};

};

如果驱动程序中不需要使用中断，则可以去除中断相关配置。如上，则对整个SPI总线在设备树文件中相关配置都已完成。

9.2驱动代码中相关匹配信息编写

static const struct spi\_device\_id sc16is7xx\_spi\_id\_table[] = {

{ "sc16is74x", (kernel\_ulong\_t)&sc16is74x\_devtype, },

{ "sc16is740", (kernel\_ulong\_t)&sc16is74x\_devtype, },

{ "sc16is741", (kernel\_ulong\_t)&sc16is74x\_devtype, },

{ "sc16is750", (kernel\_ulong\_t)&sc16is750\_devtype, },

{ "sc16is752\_1", (kernel\_ulong\_t)&sc16is752\_devtype\_chip0, },

{ "sc16is752\_2", (kernel\_ulong\_t)&sc16is752\_devtype\_chip1, },

{ "sc16is760", (kernel\_ulong\_t)&sc16is760\_devtype, },

{ "sc16is762", (kernel\_ulong\_t)&sc16is762\_devtype, },

{ }

};

如上的sc16is752\_1、sc16is752\_2为DTS上所配置的信息，匹配成功则可以加载驱动。