



# Lite200开发指南

## Lite200 Development Guide

作者：leesans (LiShanwen)

时间：2021年2月5日

版本：3.1

协议：GPL2.0



温柔正确的人总是难以生存，因为这世界既不温柔，也不正确。——比企谷八幡

## 特别声明

回首过去几年，有种莫名的失败感，

Ethan Deng

February 10, 2020

# 目录

|  |           |
|--|-----------|
| <b>1 Linux基础知识</b>                           | <b>1</b>  |
| 1.1 Linux思想：一切皆文件 . . . . .                  | 1         |
| 1.2 虚拟机使用 . . . . .                          | 1         |
| 1.3 常用命令 . . . . .                           | 1         |
| 1.4 FTP服务 . . . . .                          | 1         |
| 1.5 shell . . . . .                          | 1         |
| 1.6 Makefile . . . . .                       | 1         |
| 1.7 vim使用 . . . . .                          | 1         |
| <b>2 系统移植</b>                                | <b>2</b>  |
| 2.1 环境搭建 . . . . .                           | 2         |
| 2.1.1 交叉工具链安装 . . . . .                      | 2         |
| 2.1.2 安装下载工具 (sunxi-tools) . . . . .         | 3         |
| 2.2 u-boot移植（TF卡启动） . . . . .                | 4         |
| 2.2.1 传参 . . . . .                           | 6         |
| 2.2.1.1 bootcmd . . . . .                    | 6         |
| 2.2.1.2 bootargs . . . . .                   | 8         |
| 2.2.2 编译 . . . . .                           | 9         |
| 2.3 内核移植 . . . . .                           | 10        |
| 2.4 根文件系统移植 . . . . .                        | 17        |
| <b>3 设备树(DTS)</b>                            | <b>21</b> |
| 3.1 dts文件结构 . . . . .                        | 22        |
| 3.1.1 节点 . . . . .                           | 24        |
| 3.1.2 属性 . . . . .                           | 25        |
| 3.1.2.1 compatible . . . . .                 | 25        |
| 3.1.2.2 model . . . . .                      | 26        |
| 3.1.2.3 status . . . . .                     | 26        |
| 3.1.2.4 #address-cells #size-cells . . . . . | 26        |
| 3.1.2.5 ranges . . . . .                     | 27        |
| 3.2 设备树与驱动关系 . . . . .                       | 27        |
| <b>4 常用驱动移植</b>                              | <b>28</b> |
| 4.1 屏幕驱动移植 . . . . .                         | 28        |
| 4.2 USB驱动移植 . . . . .                        | 28        |
| 4.3 音频解码驱动移植 . . . . .                       | 28        |
| 4.4 视频解码驱动移植 . . . . .                       | 28        |

---

|                              |           |
|------------------------------|-----------|
| <b>5 常用应用移植</b>              | <b>29</b> |
| 5.1 QT5移植 . . . . .          | 29        |
| 5.2 图片浏览器移植 . . . . .        | 29        |
| 5.3 NES模拟器移植 . . . . .       | 29        |
| 5.4 视频播放器mplayer移植 . . . . . | 29        |
| <b>6 驱动开发</b>                | <b>30</b> |
| 6.1 linux驱动框架 . . . . .      | 30        |
| 6.2 字符设备驱动 . . . . .         | 30        |
| 6.3 块设备驱动 . . . . .          | 30        |
| 6.4 网络设备驱动 . . . . .         | 30        |
| <b>7 深入理解C语言</b>             | <b>31</b> |
| 7.1 C指针 . . . . .            | 31        |
| 7.2 C语言与汇编的关系 . . . . .      | 31        |
| 7.3 C语言的编译过程 . . . . .       | 31        |
| 7.4 C语言的运行过程 . . . . .       | 31        |

# 第一章 Linux基础知识

1.1 Linux思想：一切皆文件

1.2 虚拟机使用

1.3 常用命令

1.4 FTP服务

1.5 shell

1.6 Makefile

1.7 vim使用

## 第二章 系统移植

### 内容提要

- 环境搭建
- u-boot移植
- linux移植
- rootfs移植
- 深入探究

本章分三个部分来介绍Linux操作系统的移植，分别是bootloader，kernel，rootfs。其中bootloader使用最多的是u-boot，本书将使用u-boot来移植。kernel是Linux内核部分，rootfs是根文件系统。如图2.1是这三者的结构图，bootloader是SoC芯片最开始执行的代码，这个部分的主要工作是进行硬件初始化，例如时钟



图 2.1

初始化，SRAM 初始化等，同时也为后面运行C语言建立环境(初始化堆栈)。kernel在存放在bootloader之后，对于SoC来说，代码都需要在RAM中运行，这里与MCU不一样的地方就是引入了MMU（内存管理单元）。对于MCU而言，由于其执行速度低，因此运行代码都在ROM中直接运行，而对于Flash而言，其读取速度远不及RAM的速度，因此对于运行速度非常快的SoC而言，所有的代码都需要在RAM中运行。但是这里有一个问题，RAM掉电数据将会丢失，故代码保存不可能放在RAM中，当前所有的嵌入式设备而言，代码保存都是放在ROM中，因此在SoC中运行代码需要将代码搬运到RAM中然后再执行。对于bootloader来说。其代码较少，很多开发者认为再将bootloader搬到RAM中实际意义不大，除非bootloader的体积很大。实际上在u-boot 中，全志公司提供的代码是直接在ROM中运行的，当u-boot运行完大部分后会将kernel搬运到指定的RAM位置，之后将启动内核，这个启动过程后面会详细说明。对于根文件系统(rootfs)而言，由于其执行过程需要对ROM进行读写操作，因此可以不用搬运到RAM中，但是实际过程中内核启动后会产生一个虚拟的文件系统，该文件系统是挂在根文件系统的关键所在，这里不详细讲解。整体来说，大致的过程为，嵌入式设备上电后将执行bootloader，对硬件进行硬件和堆栈初始化，然后搬运内核到RAM中并启动内核，紧接着挂载根文件系统。

▲ 现在绝大多数SoC内部有一段固化的代码，该代码放在bootloader之前，也就是IROM，故实际上最开始执行的也并不是bootloader，而是芯片内部的IROM。IROM的主要目的是方便芯片的操作，例如通过USB对Flash芯片进行烧写程序。

### 2.1 环境搭建

#### 2.1.1 交叉工具链安装

所有的嵌入式设备开发都会用到各种工具，由于每个嵌入式SoC的芯片架构不同，其开发工具也不尽相同。当前用的最多的是ARM，本书中使用的Lite200主芯片的内核为ARM9，其架构使用的是ARMv5架构。最主要的工具为交叉工具链，对于F1C200S，使用的交叉工具链必须高于6.0版本，本书编译u-boot和kernel使用7.2.1版本，连接：[arm-linux-gnueabi-7.2](#) 点击即可下载。

⚠ 下载速度慢可以使用迅雷等下载软件进行下载。

下载完成后解压文件：

```
tar -vxjf gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi.tar.xz
```

然后在/usr/local目录下新建arm-linux-gcc目录

```
sudo mkdir /usr/local/arm-linux-gcc
```

进入解压目录下：

```
cd gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabi/
```

将该目录下的所有文件复制到新建的目录下

```
sudo cp -rd * /usr/local/arm-linux-gcc/
```

最后需要添加该工具链的环境变量使其可以在任何目录下执行，打开/etc/profile文件

```
sudo vim /etc/profile
```

在文件末尾添加以下内容

```
PATH=$PATH:/usr/local/arm-linux-gcc/bin
```

添加完毕，使路径生效

```
source /etc/profile
```



**笔记** arm-linux-gnueabi与arm-linux-gnueabihf的区别在于前者可以编译无浮点运算单元的SoC芯片，而后者只能编译带浮点运算单元的芯片，由于全志F1C200S内部没有浮点运算单元，因此这里必须安装前者。对于全志V3s或者H3芯片，其内部有浮点运算单元，故应该安装后者。

⚠ 上述也可以修改/etc/environment文件，在PATH值追加：

```
:/usr/local/arm-linux-gcc/bin"
```

下面验证交叉工具链是否成功安装：

在任何目录中输入arm-linux-，然后连接两次Tab键，如果出现补全交叉工具链名称，则说明安装成功，如2.2

```
lsw@lsw-VirtualBox:/usr/local/arm-linux-gcc/bin$ arm-linux-
arm-linux-addr2line      arm-linux-g++_br_real      arm-linux-gfortran.br_real      arm-linux-gnueabi-gcc-ar      arm-linux-gnueabi-nm      arm-linux-ldd
arm-linux-ar              arm-linux-gcc           arm-linux-gnueabi-addr2line    arm-linux-gnueabi-gcc-nm    arm-linux-gnueabi-objcopy   arm-linux-nm
arm-linux-as              arm-linux-gcc-6.4.0     arm-linux-gnueabi_ar        arm-linux-gnueabi-gcc-ranlib  arm-linux-gnueabi-objdump  arm-linux-objcopy
arm-linux-c++             arm-linux-gcc-6.4.0.br_real  arm-linux-gnueabi_as        arm-linux-gnueabi-gcov       arm-linux-gnueabi-ranlib  arm-linux-objdump
arm-linux-c++_br_real     arm-linux-gcc_ar       arm-linux-gnueabi_c++       arm-linux-gnueabi-gcov-dump  arm-linux-gnueabi-readelf  arm-linux-objdump
arm-linux-cc              arm-linux-gcc_br_real  arm-linux-gnueabi_c++filt    arm-linux-gnueabi-gcov-tool  arm-linux-gnueabi-size    arm-linux-ranlib
arm-linux-cc_cc_real      arm-linux-gcc_nm       arm-linux-gnueabi_cpp       arm-linux-gnueabi-gdb       arm-linux-gnueabi-readelf  arm-linux-size
arm-linux-c++filt         arm-linux-gcc_ranlib   arm-linux-gnueabi_dwp      arm-linux-gnueabi-gfortran  arm-linux-gnueabi-strings  arm-linux-readelf
arm-linux-cpp             arm-linux-gcov        arm-linux-gnueabi_elfedit   arm-linux-gnueabi_gprof     arm-linux-gnueabi-strip   arm-linux-size
arm-linux-cpp_br_real    arm-linux-gcov_dump   arm-linux-gnueabi_g++       arm-linux-gnueabi_ld       arm-linux-gnueabi_strip   arm-linux-size
arm-linux-elfedit         arm-linux-gcov_tool   arm-linux-gnueabi_gcc       arm-linux-gnueabi_ld_bfd   arm-linux-ld             arm-linux-strip
arm-linux-g++              arm-linux-gfortran   arm-linux-gnueabi_gcc-7.2.1  arm-linux-gnueabi_ld_gold  arm-linux-ldconfig
```

图 2.2

如果没有出现，则进行下面操作：

安装必要的动态链接库

```
sudo apt-get install lib32ncurses5 lib32z1
```

上面安装的动态链接库主要是因为ubuntu是64位的操作系统，而交叉工具链是32位的，因此我们需要安装32位必须的一些动态链接库。

## 2.1.2 安装下载工具 (sunxi-tools)

⚠ 该部分只针对SPI\_FLASH方式启动的用户而言，使用TF卡启动可以忽略此小结

因为后面会使用git工具，所以我们先要安装git工具，这个工具的目的就是从github上下载或者称之为克隆代码用的。使用如下命令安装git工具：

**sudo apt-get install git**

然后我们下载我们需要使用的下载工具（sunxi-tools）：

**git clone https://github.com/linux-sunxi/sunxi-tools.git -b f1c100s-spiflash**

克隆下来的工具将保存在当前目录下，进入该目录后，执行：

**make**

然后安装即可：

**make install**

**▲** 如果出现错误，这是因为有些必要的依赖库没有安装，执行如下命令即可安装：

**sudo apt-get install libusb-1.0-0-dev zlib1g-dev**

若出现无法下载的情况，请先下载该库，然后手动安装即可，步骤如下：

**wget http://packages.deepin.com/deepin/pool/main/libu/libusb-1.0/libusb-1.0-0-dev\_1.0.21-1\_amd64.deb**

下载完毕后使用如下指令安装该库：

**sudo dpkg -i libusb-1.0-0-dev\_1.0.21-1\_amd64.deb**

安装完毕后需要验证是否安装成功，在终端输入：**sunxi-fel**

如果有如下输出则说明安装成功。

```
admir@admir-PC:~/licheePI/sunxi-tools$ sunxi-fel
sunxi-fel v1.4.1-104-g11a9d20

Usage: sunxi-fel [options] command arguments... [command...]
  -h, --help                      Print this usage summary and exit
  -v, --verbose                    Verbose logging
  -p, --progress                  "write" transfers show a progress bar
  -l, --list                       Enumerate all (USB) FEL devices and exit
  -d, --dev bus:devnum            Use specific USB bus and device number
  --sid SID                        Select device by SID key (exact match)

  spl file                         Load and execute U-Boot SPL
    If file additionally contains a main U-Boot binary
    (u-boot-sunxi-with-spl.bin), this command also transfers that
    to memory (default address from image), but won't execute it.

  uboot file-with-spl             like "spl", but actually starts U-Boot
    U-Boot execution will take place when the fel utility exits.
    This allows combining "uboot" with further "write" commands
    (to transfer other files needed for the boot).
```

图 2.3

## 2.2 u-boot移植（TF卡启动）

一般来说u-boot是启动内核的关键所在，当前大部分使用的嵌入式设备都是u-boot，最新版本的u-boot几乎包含当前主流的SoC芯片，Lite200使用的芯片和licheePI nano相同，大部分硬件也是兼容的，为了快速移植该部分，这里采用licheePI nano的u-boot来进行移植。在终端输入如下命令克隆u-boot：

**git clone https://github.com/Lichee-Pi/u-boot.git -b nano-v2018.01**

克隆完毕文件会保存在当前目录下，进入该目录，

**cd u-boot**

在该文件夹下有很多分支，我们可以查看所有分支，使用如下命令：

**git branch -a**

现在我们使用的是nano开发板，所以将当前分支切换到nano分支，命令如下：

**git checkout nano-v2018.01**

u-boot默认的没有指定交叉工具链和架构，因此在编译之前需要指定交叉工具链和芯片架构，u-boot的交叉编译器在u-boot的根目录下中的Makefile文件中定义了。打开Makefile文件：

**vim Makefile**

找到CROSS\_COMPILE变量，将其改为如下：

ARCH=arm

CROSS\_COMPILE=arm-linux-gnueabi-

如图2.5所示。config目录下是板级配置文件，由于每个板子的外设不同，因此编译之前必须要对u-boot进行

```

242 #####交叉编译器设置#####
243
244 # set default to nothing for native builds
245 ARCH?=arm
246 CROSS_COMPILE ?=arm-linux-gnueabi-
247
248 KCONFIG_CONFIG ?= .config
249 export KCONFIG_CONFIG
250
251 # SHELL used by kbuild
252 CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
253         else if [ -x /bin/bash ]; then echo /bin/bash; \
254         else echo sh; fi ; fi)
255

```

图 2.4

配置。然而配置是一件比较繁琐的事情，特别是像u-boot这种比较复杂的项目而言，初学者几乎无法完成。幸运的是对于大部分开发板而言，config目录下有其配置好的默认配置文件。进入config目录中，然后执行ls查看当前所有的配置文件

**cd config**

**ls**

找到licheepi\_nano\_defconfig和licheepi\_nano\_spiflash\_defconfig，前者表示为TF卡启动，后者表示从SPI设备启动，这里有前者即可，如图2.5所示。现在回到上级目录，然后执行make licheepi\_nano\_defconfig

| config         | lager_defconfig                    | orangeipi_pc2_defconfig          |
|----------------|------------------------------------|----------------------------------|
| h_defconfig    | Lamobo_R1_defconfig                | orangeipi_pc_defconfig           |
| _defconfig     | legoev3_defconfig                  | orangeipi_pc_plus_defconfig      |
| _defconfig     | libretech_all_h3_cc_h3_defconfig   | orangeipi_plus2e_defconfig       |
| nfig           | licheepi_nano_defconfig            | orangeipi_plus_defconfig         |
| _defconfig     | licheepi_nano_spiflash_defconfig   | orangeipi_prime_defconfig        |
| _defconfig     | licheepi_zero_defconfig            | orangeipi_win_defconfig          |
| nfig           | Linksprite_pcDuino3_defconfig      | orangeipi_zero_defconfig         |
| _defconfig     | Linksprite_pcDuino3_Nano_defconfig | orangeipi_zero_plus2_defconfig   |
| defconfig      | Linksprite_pcDuino_defconfig       | origen_defconfig                 |
| _cs0_defconfig | lion-rk3368_defconfig              | ot1200_defconfig                 |
| _cs1_defconfig | liteboard_defconfig                | ot1200_spl_defconfig             |
| _defconfig     | ls1012afrdm_qspi_defconfig         | P1010RDB-PA_36BIT_NAND_defconfig |
|                | ls1012aqds_qspi_defconfig          | P1010RDB-PA_36BIT_NAND_SECBOOT   |
|                | ls1012afrdm_qspi_defconfig         | P1010RDB-PA_36BIT_NOR_defconfig  |
|                | ls1012aqds_qspi_defconfig          | P1010RDB-PA_36BIT_NOR_SECBOOT    |

图 2.5

**cd ..**

**make make licheepi\_nano\_defconfig**

配置完成后就可以进入图形界面进行配置了，执行make menuconfig命令：

```

lsw@lsw-VirtualBox:~/licheepi/u-boot/configs$ cd ..
lsw@lsw-VirtualBox:~/licheepi/u-boot$ make licheepi_nano_defconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/conf.o
SHIPPED scripts/kconfig/zconf.tab.c
SHIPPED scripts/kconfig/zconf.lex.c
SHIPPED scripts/kconfig/zconf.hash.c
HOSTCC scripts/kconfig/zconf.tab.o
HOSTLD scripts/kconfig/conf
#
# configuration written to .config
#
lsw@lsw-VirtualBox:~/licheepi/u-boot$ 

```

图 2.6

### make menuconfig

此时出现图形配置选项，如图2.7 图形配置选项中有两个非常重要的参数配置—传参，下小结重点讲解该

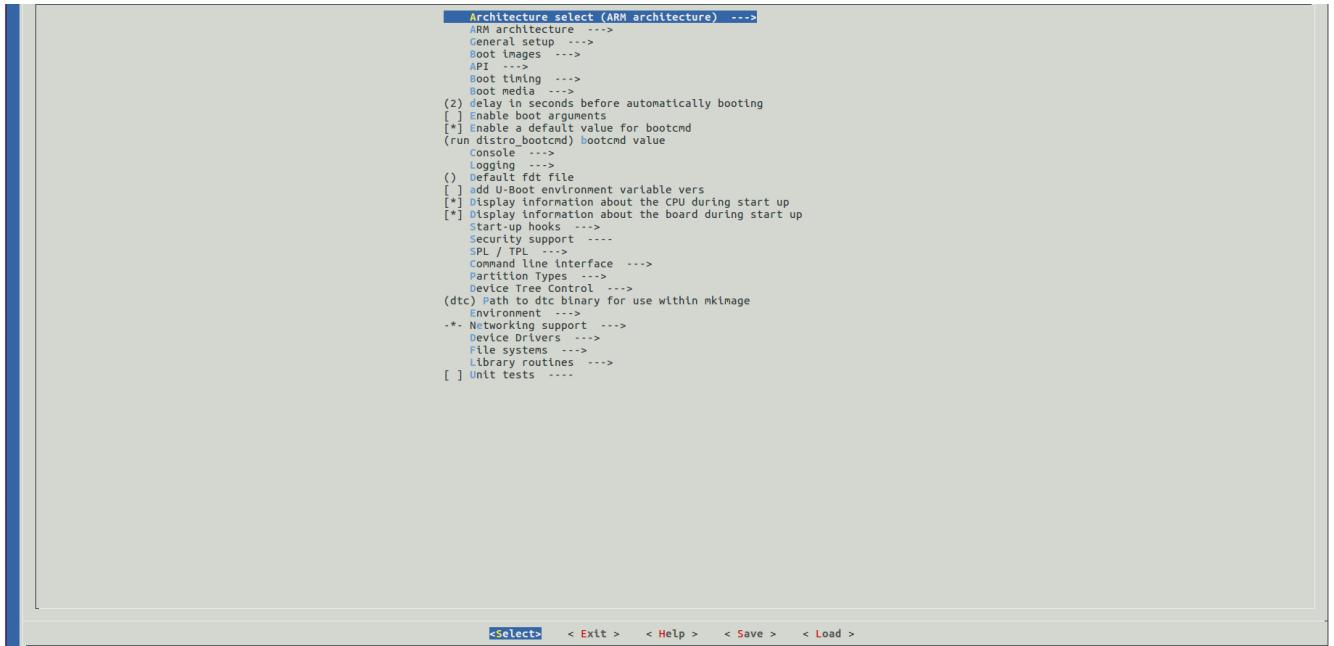


图 2.7

部分。

#### 2.2.1 传参

如图2.1所示，其中parameter就是传参需要的参数地址。这里不得不提bootargs和bootcmd，这两个环境变量可以说是所有环境变量中最重要的变量，读者看完之后就会有体会。

##### 2.2.1.1 bootcmd

在最开始提到过，内核一般不在flash中运行，这样就需要将内核搬运到内存中，这个过程需要u-boot来完成。对于mmc(TF卡)而言，在u-boot有专门的命令load mmc，该命令可以将mmc中的代码从flash搬运到指定的地址处。当环境变量bootdelay计数到0时，此时uboot就会开始执行bootcmd中的命令。

**⚠️** bootdelay这个环境变量是一个计数器，当u-boot主体运行完毕后，此时bootdelay该变量的值将会开始递减，递减时间为1s，当递减到0时，此时u-boot将会跳转到bootcmd处开始执行bootcmd命令。

load mmc命令的使用比较简单，这里以F1C200S例子来讲解

**load mmc 0:1 0x80008000 zImage**

load mmc有三个参数，第一个参数是mmc分区，第二个参数是目标地址，第三个参数是源文件。即上面的命令意思是将mmc的0:1分区中的zImage复制到内存中的0x80008000地址处。

#### ▣ TF卡属于mmc存储器的一种

上面完成了zImage的搬运任务，但是当前大部分的驱动都是基于设备树写的，关于设备树的知识，后面将单独一章节讲解，这种基于设备树(dts)的驱动使得代码重复率降低，但是内核就需要对其设备进行解析，否则设备将无法确定是哪一个驱动。这里将引入dtb文件，所谓dtb文件就是dts文件通过dtc编译器编译成的目标文件即二进制文件。内核通过加载并解析并解析该文件从而获取驱动相关的配置信息。下面给出F1C200S中使用的

**load mmc 0:1 0x80c08000 suniv-f1c100s-licheepi-nano.dtb**

上面的命令意思是将mmc的0:1分区中的suniv-f1c100s-licheepi-nano.dtb文件加载到内存中的0x80c08000地址处。关于0x80008000和0x80c08000这两个地址是如何确定的后面会详细讲到。将zImage（也就是内核镜像文件）和dtb（设备树文件）搬运到了指定内存处，此时u-boot的任务完全结束了，剩下的工作就是启动内核了，这个就需要bootm或者bootz命令。这里以bootz命令为例，如下：

**bootz 0x80008000 - 0x80c08000**

上面的意思是告诉内核镜像的起始地址为0x80008000，加载的设备树地址为0x80c08000。



1. bootz命令的格式是：bootz空格0x80008000空格-空格0x80c08000,注意-左右有空格
2. bootm命令是对没有使用设备树内核的镜像启动命令，早期版本的内核没有引入设备树，因此对于早期的内核一般使用的是bootm，其命令格式为bootm 内核地址，比如bootm x0x30008000，意思是说从0x30008000开始启动内核，启动内核的过程其实是将pc指针指向该地址，这样处理器就会从该地址处运行代码。

上面详细讲解了bootcmd环境变量，该环境变量若要执行多条命令，则每个命令之间用;隔开，例如F1C200S常用的bootcmd为：

**load mmc 0:1 0x80008000 zImage;load mmc 0:1 0x80c08000 suniv-f1c100s-licheepi-nano.dtb;bootz 0x80008000 - 0x80c08000;**

在图形配置界面中（图2.7），图2.8所示中默认开启了bootcmd，其中变量值为run

distro\_bootcmd，这个值的意思是执行一个名为distro\_bootcmd的一个脚本文件。对于一般的内核，可

```

Boot media --->
(2) delay in seconds before automatically booting
[ ] Enable boot arguments
[*] Enable a default value for bootcmd
(run distro_bootcmd) bootcmd value

Console -
Logging --->
() Default fdt file
[ ] add U-Boot environment variable vers
[*] Display information about the CPU during start up
[*] Display information about the board during start up

```

图 2.8

以直接上面的命令填写到bootcmd中，将光标移动到bootcmd value处，然后按回车键，进入编辑界面，如

图2.9：

选择【Ok】后按回车键即可保存该变量值。如图2.10

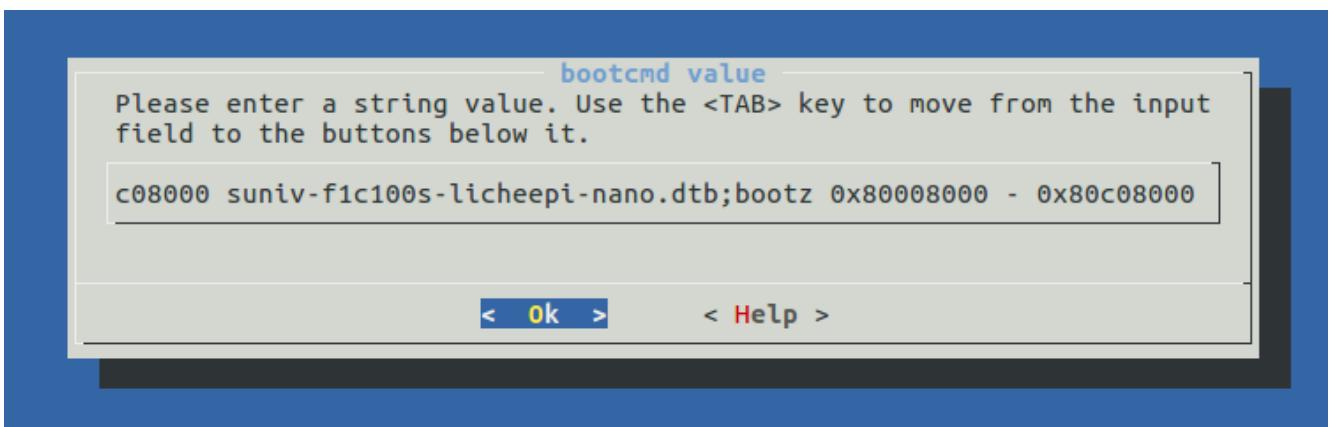


图 2.9

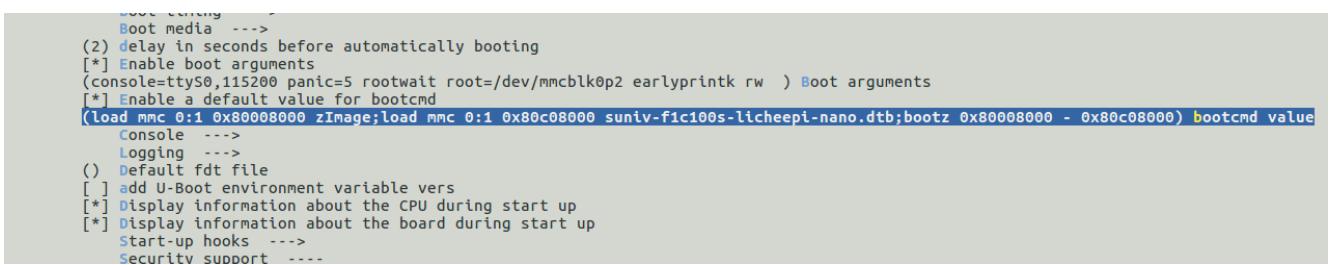


图 2.10

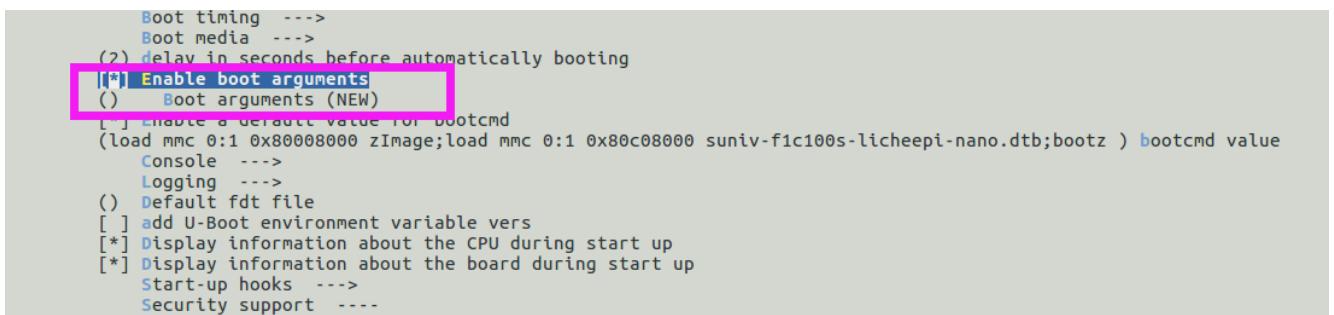
### 2.2.1.2 bootargs

bootargs也是u-boot环境变量中一个非常重要的变量，上面已经讲解了内核的启动可以通过bootcmd来完成，那接下来内核启动完毕后必须挂在根文件系统(rootfs)。但是内核并不知道根文件系统的具体位置，我们必须要告诉根文件的位置后内核才能将其挂载，这时就需要有bootargs变量。该变量的作用是告诉内核根文件系统的位置和属性以及必要的配置，这个就是图2.1中parameter的部分，u-boot会将bootargs的值（其实就是字符串）保存到事先规定好的地址处，当内核启动成功后就会获取该地址处的字符串值，然后对其进行解析并做相关的内核配置。这个过程看起来比较复杂，这里可以不用去了解细节，只需要知道bootargs是告诉内核根文件系统的位置属性以及一些配置参数。对于mmc而言，一般采用分区来划分内存区域，下面以F1C200S常用的bootargs来讲解。

```
console=ttyS0,115200 panic=5 rootwait root=/dev/mmcblk0p2 earlyprintk rw
```

上面console=ttyS0,115200表示终端为ttyS0即串口0,波特率为115200。panic=5字面意思是恐慌，即linux内核恐慌，其实就是linux不知道怎么执行了，此时内核就需要做一些相关的处理，这里的5表示超时时间，当Linux卡住5秒后仍未成功就会执行Linux恐慌异常的一些操作。rootwait该参数是告诉内核挂在文件系统之前需要先加载相关驱动，这样做的目的是防止因mmc驱动还未加载就开始挂载驱动而导致文件系统挂载失败，所以一般bootargs中都要加上这个参数。root=/dev/mmcblk0p2表示根文件系统的位置在mmc的0:2分区处，/dev是设备文件夹，内核在加载mmc中的时候就会在根文件系统中生成mmcblk0p2设备文件，该设备文件其实就是mmc的0:2分区，这样内核对文件系统的读写操作方式本质上就是读写/dev/mmcblk0p2该设备文件。earlyprintk参数是指在内核加载的过程中打印输出信息，这样内核在加载的时候终端就会输出相应的启动信息。rw表示文件系统的操作属性，此处rw表示可读可写。

设置bootargs参数时将光标移动到Enable boot, arguments处，点击键盘‘Y’按键使其参数开启，如图2.11然后将光标移动到下一个选项即Boot arguments (NEW)选项，然后按回车键对该环境变量进行编辑，将上



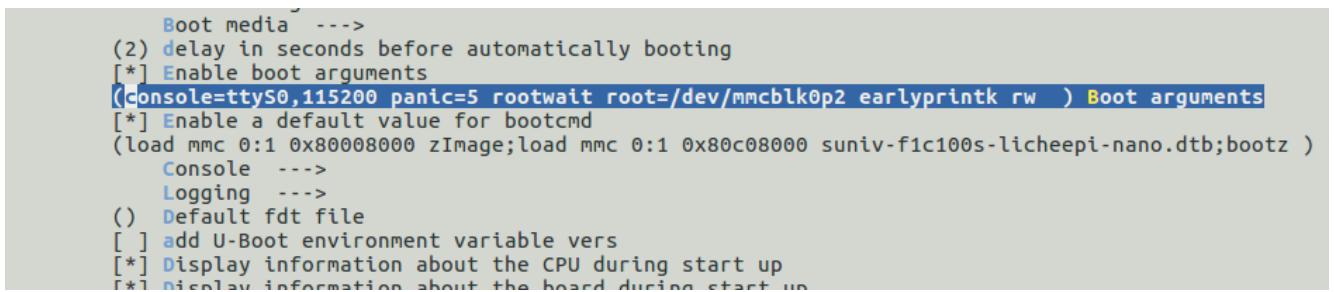
```

Boot timing --->
Boot media --->
(2) delay in seconds before automatically booting
[*] Enable boot arguments
() Boot arguments (NEW)
[ ] choose a default value for bootcmd
(load mmc 0:1 0x80008000 zImage;load mmc 0:1 0x80c08000 suniv-f1c100s-licheepi-nano.dtb;bootz ) bootcmd value
    Console --->
    Logging --->
() Default fdt file
[ ] add U-Boot environment variable vers
[*] Display information about the CPU during start up
[*] Display information about the board during start up
    Start-up hooks --->
    Security support ----

```

图 2.11

面的bootargs 字符串填写进入，然后按【Ok】键保存，如图2.12



```

Boot media --->
(2) delay in seconds before automatically booting
[*] Enable boot arguments
[*] Enable a default value for bootcmd
(load mmc 0:1 0x80008000 zImage;load mmc 0:1 0x80c08000 suniv-f1c100s-licheepi-nano.dtb;bootz )
    Console --->
    Logging --->
() Default fdt file
[ ] add U-Boot environment variable vers
[*] Display information about the CPU during start up
[*] Display information about the board during start up

```

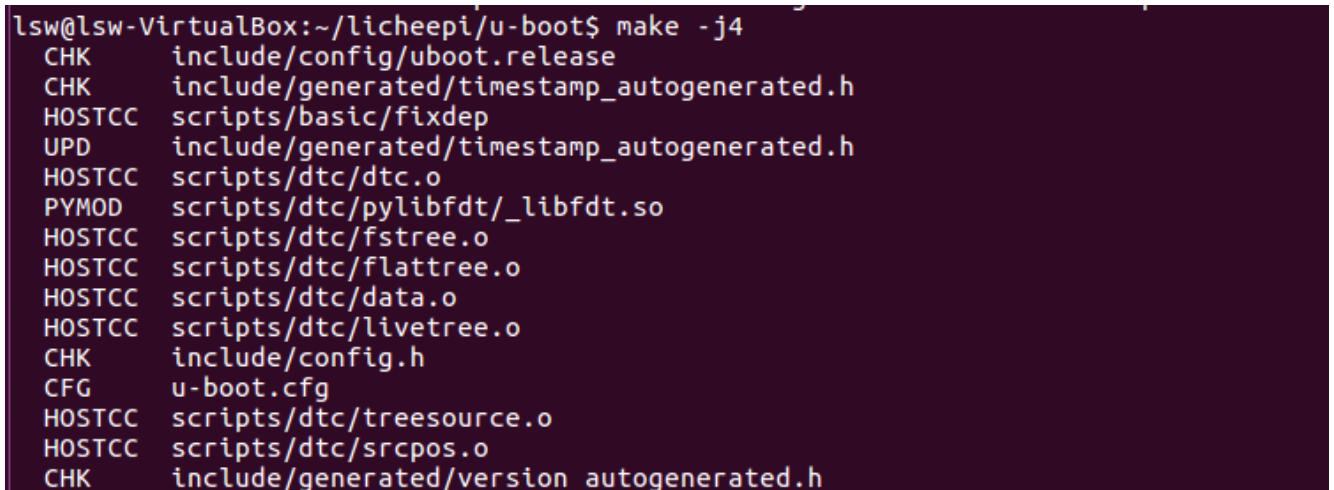
图 2.12

## 2.2.2 编译

先保存图形配置界面后推出界面，在终端执行make -j4即可对整个u-boot进行编译。

**make -j4**

如图2.13



```

lsw@lsw-VirtualBox:~/licheepi/u-boot$ make -j4
CHK include/config/uboot.release
CHK include/generated/timestamp autogenerated.h
HOSTCC scripts/basic/fixdep
UPD include/generated/timestamp autogenerated.h
HOSTCC scripts/dtc/dtc.o
PYMOD scripts/dtc/pylibfdt/_libfdt.so
HOSTCC scripts/dtc/fstree.o
HOSTCC scripts/dtc/flattree.o
HOSTCC scripts/dtc/data.o
HOSTCC scripts/dtc/livetree.o
CHK include/config.h
CFG u-boot.cfg
HOSTCC scripts/dtc/treesource.o
HOSTCC scripts/dtc/srcpos.o
CHK include/generated/version autogenerated.h

```

图 2.13

🔔 make -j4后面的-j4表示4个核心进行编译，若电脑的处理器是2核心，请使用make -j2进行编译。

编译完成后会在当前目录生成u-boot-sunxi-with-spl.bin烧录文件。如图2.14，该文件就是我们最终要烧录的二进制文件。在当前目录下会有一个隐藏的文件.config，该文件是u-boot编译后根据各个选项产生

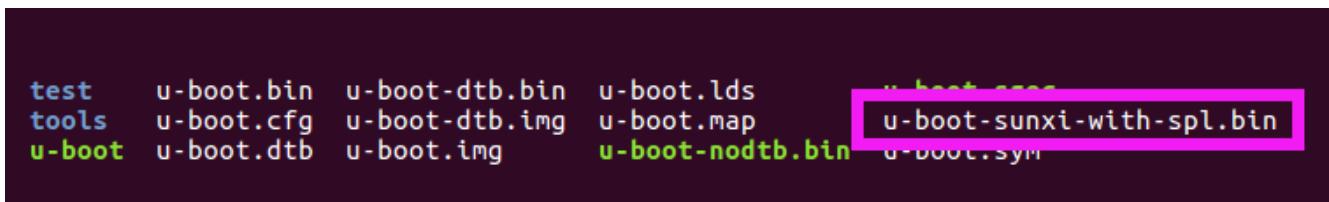


图 2.14

的配置文件，这个配置文件记录了所有配置选项的宏开关，编译的时候是根据最终的.config文件来进行编译的，当然编译前是需要有脚本解析.config文件然后进行相应的编译。

**笔记** 关于u-boot的配置过程这里简单说明下，其文件结构的和linux内核结构大致相同，对于当前而言，其配置方式有三种，分别是

1. make menuconfig
2. make xxx\_defconfig
3. .config

这三种方式都可以对内核进行配置，三者之间的关系比作去饭店吃饭，make menuconfig可以看作菜单，.config和make xxx\_defconfig是客人点的菜，但是其三个文件之间可能因为存在依赖关系，如果直接修改.config可能会导致配置失效。

只要将u-boot-sunxi-with-spl.bin烧录到tf卡的8k偏移处地址就可以了，烧录步骤如下：使用dd命令进行块搬移：

```
sudo dd if=u-boot-sunxi-with-spl.bin of=/dev/sdb bs=1024 seek=8
```

if指的是输入文件，of指的是输出文件，这里的输出文件为主机电脑的/dev/sdb文件，也就是TF卡，这个可以用gparted 软件查看，该软件可以直接用命令安装即可：sudo apt-get install gparted，安装好后打开该软件，如图2.15在右上角可以看到两个硬盘sda，其中一个是主机的本地硬盘，另一个是TF，即sdb这里可以看到TF卡的卷标名为sdb，因此这里的of=/dev/sdb 烧录到8k偏移地址处是指绝对地址，这个绝对地址指的是TF卡的物理地址。这里为何是8k处而不是其他地址是由F1C200S 内部的IROM中的一小段代码决定的。

烧录完毕后如图2.16 现在这个TF卡内的u-boot可以启动内核了。将TF卡插入到开发板上，然后通过USB连接到电脑端，打开串口工具，可以看到按下开发板上的复位按钮，可以看到此时串口终端有信息输出，如图2.17，由于没有烧录内核，因此加载内核失败，停止在了u-boot命令终端处。可以在终端输入pri命令打印环境变量的所有值，如图2.18所示，可以看到bootcmd和bootargs是正确的。

也许到这里读者可能会认为u-boot内容结束了，其实不然，u-boot还有很多需要分析，由于有些部分过于复杂，对初学者不友好，因此u-boot的进阶内容将放到后面详细讲解。

## 2.3 内核移植

内核移植相对与u-boot移植复杂些，对于F1C200S而言，Linux官方源码已经对licheepi nano进行了支持，因此本次移植采用licheepi nano的配置文件，下面以linux5.7版本内核来讲解kernel移植步骤。进入linux内核官网(<https://www.kernel.org/>)，点击<https://www.kernel.org/pub/> 进入下载界面，如图2.19，下载连接为 <https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.7.1.tar.gz>，可以使用下载工具进行下载，下

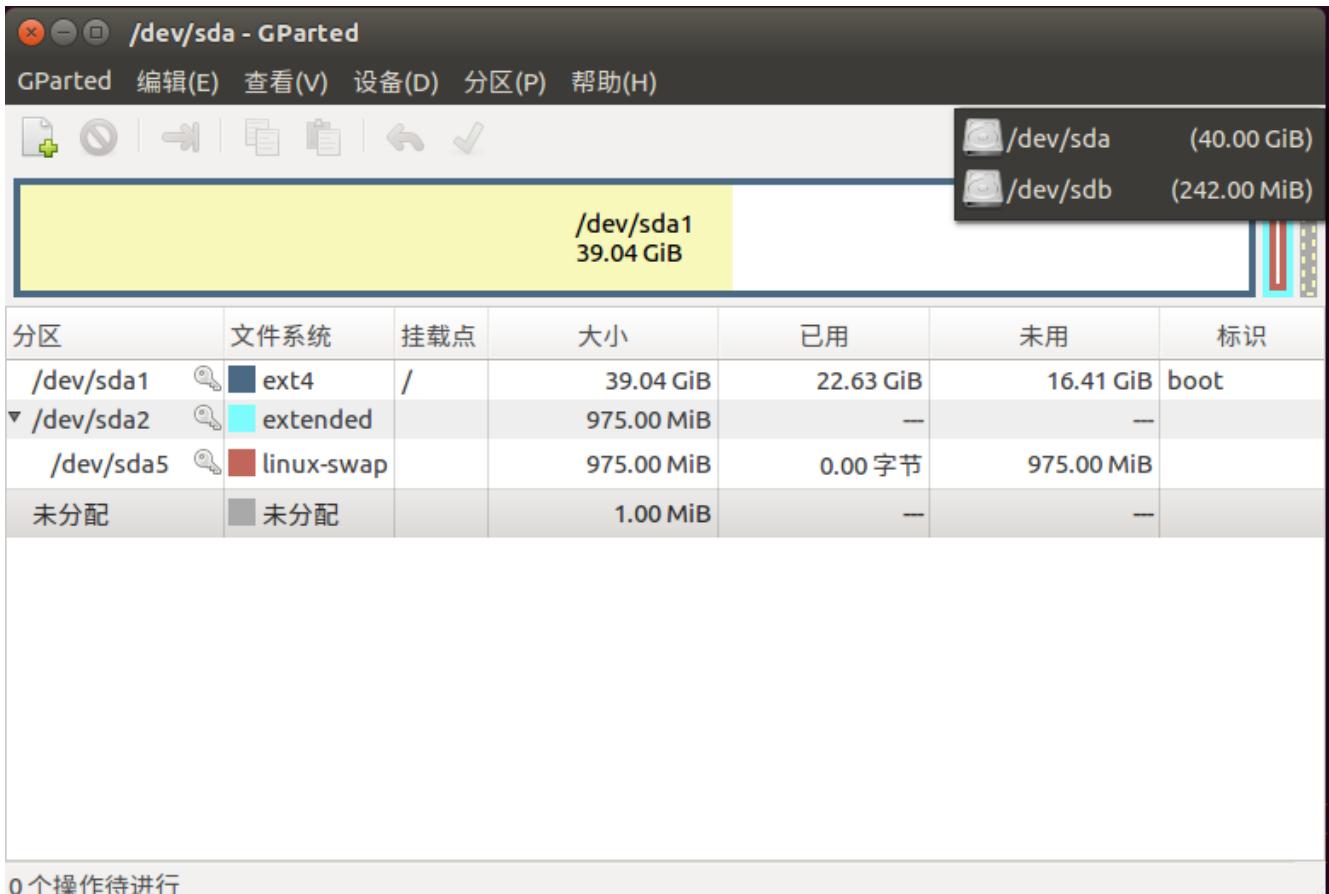


图 2.15

```
lsw@lsw-VirtualBox:~/licheepi/u-boot$ sudo dd if=u-boot-sunxi-with-spl.bin of=/dev/sdb bs=1024 seek=8
[sudo] lsw 的密码:
记录了 536+0 的读入
记录了 536+0 的写出
548864 bytes (549 kB, 536 KiB) copied, 2.95585 s, 186 kB/s
lsw@lsw-VirtualBox:~/licheepi/u-boot$
```

图 2.16

```
U-Boot SPL 2018.01-05679-g013ca45-dirty (Feb 16 2021 - 18:26:34)
DRAM: 64 MiB
Trying to boot from MMC1

U-Boot 2018.01-05679-g013ca45-dirty (Feb 16 2021 - 18:26:34 +0800) Allwinner Technology

CPU: Allwinner F Series (SUNIV)
Model: Lichee Pi Nano
DRAM: 64 MiB
MMC: SUNXI SD/MMC: 0
*** Warning - bad CRC, using default environment

In: serial@1c25000
Out: serial@1c25000
Err: serial@1c25000
Net: No ethernet found.
Hit any key to stop autoboot: 0
** Invalid partition 1 **
** Invalid partition 1 **
=>
```

图 2.17

```
=> pri
arch=arm
baudrate=115200
board=sunxi
board_name=sunxi
bootargs=console=ttyS0,115200 panic=5 rootwait root=/dev/mmcblk0p2 earlyprintk rw
bootcmd=load mmc 0:1 0x80008000 zImage;load mmc 0:1 0x80c08000 suniv-f1c100s-licheepi-nano.dtb;bootz 0x80008000 - 0x80c08000
bootdelay=2
cpu=arm926ejs
fdtcontroladdr=83e56da8
fel_booted=1
fileaddr=80c08000
filesize=1572
soc=sunxi
stderr=serial@1c25000
stdin=serial@1c25000
stdout=serial@1c25000
```

图 2.18

The screenshot shows the homepage of The Linux Kernel Archives. At the top, there's a navigation bar with links for About, Contact us, FAQ, Releases, Signatures, and Site news. To the right of the navigation is a small penguin icon. Below the navigation, there's a section for "Protocol" with options for HTTP, GIT, and RSYNC, each with its corresponding URL. A yellow button labeled "Latest Release" with the number "5.11" and a downward arrow is prominently displayed. To the right of this button is a list of kernel versions from 5.11 down to 5.7.1, each with a download link. The version "5.7.1" is highlighted with a red box.

| Version          | Date        | File Type |
|------------------|-------------|-----------|
| 5.6.7.tar.xz     | 23-Apr-2020 | 08:47     |
| 5.6.8.tar.gz     | 29-Apr-2020 | 14:42     |
| 5.6.8.tar.sign   | 29-Apr-2020 | 14:42     |
| 5.6.8.tar.xz     | 29-Apr-2020 | 14:42     |
| 5.6.9.tar.gz     | 02-May-2020 | 07:00     |
| 5.6.9.tar.signed | 02-May-2020 | 07:00     |
| 5.6.9.tar.xz     | 02-May-2020 | 07:00     |
| 5.6.10.tar.gz    | 30-Mar-2020 | 05:52     |
| 5.6.10.tar.sign  | 30-Mar-2020 | 05:52     |
| 5.6.10.tar.xz    | 30-Mar-2020 | 05:52     |
| 5.7.0.tar.gz     | 30-Mar-2020 | 05:52     |
| 5.7.0.tar.sign   | 30-Mar-2020 | 05:52     |
| 5.7.0.tar.xz     | 30-Mar-2020 | 05:52     |
| 5.7.1.tar.gz     | 07-Jun-2020 | 11:17     |
| 5.7.1.tar.sign   | 07-Jun-2020 | 11:17     |
| 5.7.1.tar.xz     | 07-Jun-2020 | 11:17     |
| 5.7.2.tar.gz     | 22-Jun-2020 | 07:44     |
| 5.7.2.tar.signed | 22-Jun-2020 | 07:44     |
| 5.7.2.tar.xz     | 22-Jun-2020 | 07:44     |
| 5.7.3.tar.gz     | 29-Jun-2020 | 08:27     |
| 5.7.3.tar.sign   | 29-Jun-2020 | 08:27     |
| 5.7.3.tar.xz     | 29-Jun-2020 | 08:27     |
| 5.7.4.tar.gz     | 31-Jun-2020 | 16:57     |
| 5.7.4.tar.sign   | 31-Jun-2020 | 16:57     |
| 5.7.4.tar.xz     | 31-Jun-2020 | 16:57     |

图 2.19

载后的源码包通过FTP传输到虚拟机。上传完毕后进入虚拟机解压源码，和u-boot步骤一样，在编译前必须对源码进行配置。进入该源码中的arch/arm/configs目录中，可以看到有很多开发板的配置文件，其中sunxi\_defconfig是全志的配置文件，但是该配置文件非常不全，需要额外配置大量的选项，一般选项多大上千个，这里先使用licheepi\_nano的配置文件。<https://github.com/LiShanwenGit/MelonPI-MINI/tree/master/software> 该目录下有一个linux-licheepi\_nano\_defconfig文件，这个文件是针对licheepi\_nano的配置文件。这个配置文件与Lite200完全兼容，后面会详细分析该文件以及一些配置，这里为了快速启动内核就不做过多阐述。下载该文件，然后将其放到arch/arm/configs/目录下，如图2.20，然后回到主目录，和u-boot一样，在编译时必须指定交叉编译器。打开主目录下的Makefile文件，然后找到CROSS\_COMPILE变量，将其修改为

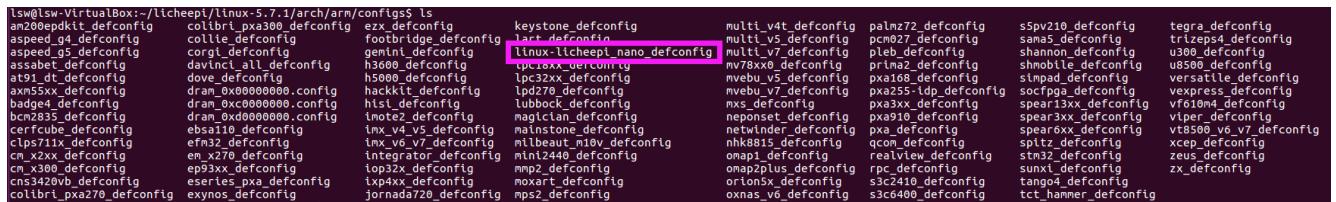


图 2.20

读者使用的交叉工具链前缀，同时指定架构即ARCH变量，这里使用arm-linux-gnueabi-为例，如图 保存

```
358 # Make CROSS_COMPILE=ld04-Linux-
359 # Alternatively CROSS_COMPILE can be set in the environment.
360 # Default value for CROSS_COMPILE is not to prefix executables
361 # Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
362 ARCH ?= arm
363 CROSS_COMPILE ?= arm-linux-gnueabi-
364
365 # Architecture as present in compile.h
366 UTS_MACHINE := $(ARCH)
367 SRCARCH := $(ARCH)
368
369 # Additional ARCH settings for x86
370 ifeq ($(ARCH),i386)
371     SRCARCH := x86
372 endif
373 ifeq ($(ARCH),x86_64)
374     SRCARCH := x86
```

图 2.21

退出，然后在终端处执行

**make menuconfig**

进入图形配置界面，如图2.22所示。可以看到其配置模式和u-boot近似相同，也是通过上下键左右来操作和[Y][N]键来选择是否编译进内核。这里简单的先让linux内核跑起来，因此使用默认配置，不做任何修改。选择[Save]然后退出，在终端下输入

**make -j4**

如图2.23所示。编译完毕后在就会生成zImage文件和dtb文件，zImage在arch/arm/boot目录下，dtb在arch/arm/boot/dts目录下。如图2.24。镜像编译完毕后就需要将其烧录到TF卡中，从上面的u-boot中bootcmd中可以看到需要将zImage 和dtb文件复制到TF卡的0:1分区中，这样u-boot在执行bootcmd中的load mmc命令时就可以找到zImage和dtb文件。下面来对TF卡进行分区。分区工具最常用的是gparted软件，该软件可以直接在终端中安装即可。

**sudo apt-get install gparted**

然后插入TF卡到电脑的USB上，打开该软件，可以看到此时有两个硬盘，一个是sda另一个是sdb，其中sdb就是TF卡。如图2.25所示，选中sdb，可以看到有一个未分配的空间，一般对于嵌入式系统而言需要将其分

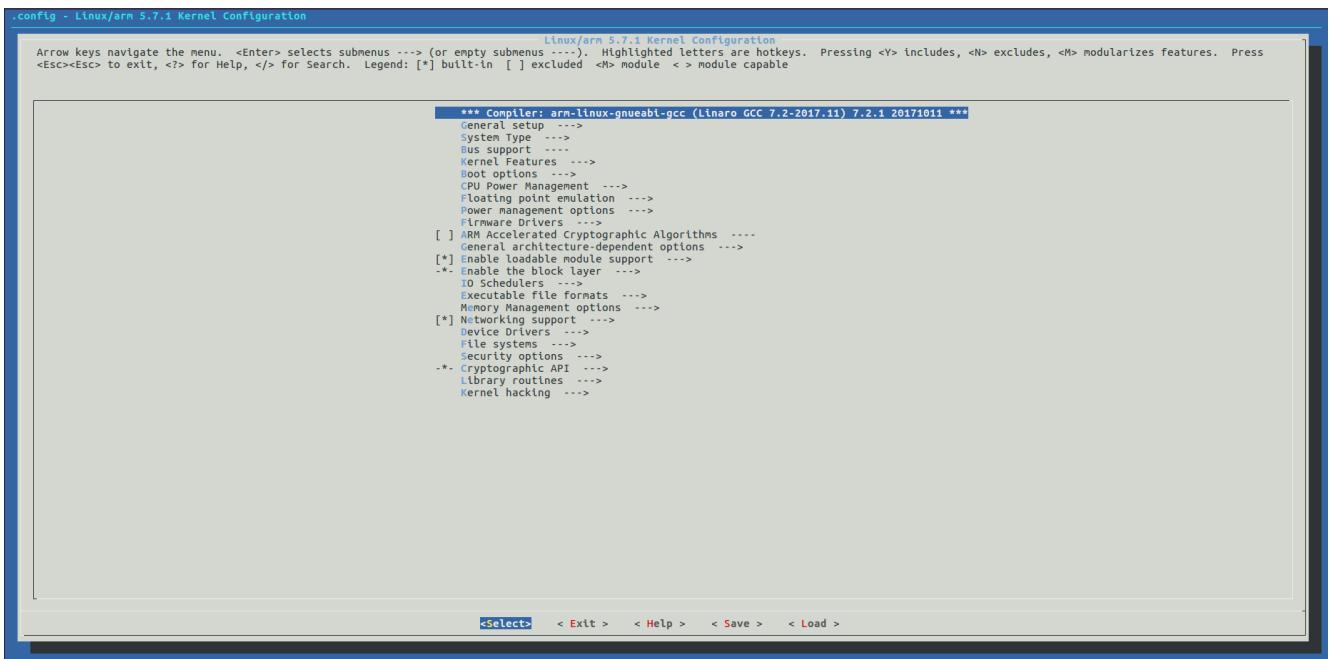


图 2.22

```
lsw@lsw-VirtualBox:~/licheepi/linux-5.7.1$ make -j4
SYSHDR arch/arm/include/generated/uapi/asm/unistd-common.h
SYSHDR arch/arm/include/generated/uapi/asm/unistd-oabi.h
SYSHDR arch/arm/include/generated/uapi/asm/unistd-eabi.h
WRAP arch/arm/include/generated/uapi/asm/kvm_para.h
WRAP arch/arm/include/generated/uapi/asm/bitsperlong.h
WRAP arch/arm/include/generated/uapi/asm/bpf_perf_event.h
WRAP arch/arm/include/generated/uapi/asm/errno.h
WRAP arch/arm/include/generated/uapi/asm/ioctl.h
WRAP arch/arm/include/generated/uapi/asm/ipcbuf.h
WRAP arch/arm/include/generated/uapi/asm/msgbuf.h
WRAP arch/arm/include/generated/uapi/asm/param.h
WRAP arch/arm/include/generated/uapi/asm/poll.h
WRAP arch/arm/include/generated/uapi/asm/resource.h
WRAP arch/arm/include/generated/uapi/asm/sembuf.h
WRAP arch/arm/include/generated/uapi/asm/shmbuf.h
WRAP arch/arm/include/generated/uapi/asm/siginfo.h
WRAP arch/arm/include/generated/uapi/asm/socket.h
WRAP arch/arm/include/generated/uapi/asm/sockios.h
WRAP arch/arm/include/generated/uapi/asm/termbits.h
```

图 2.23

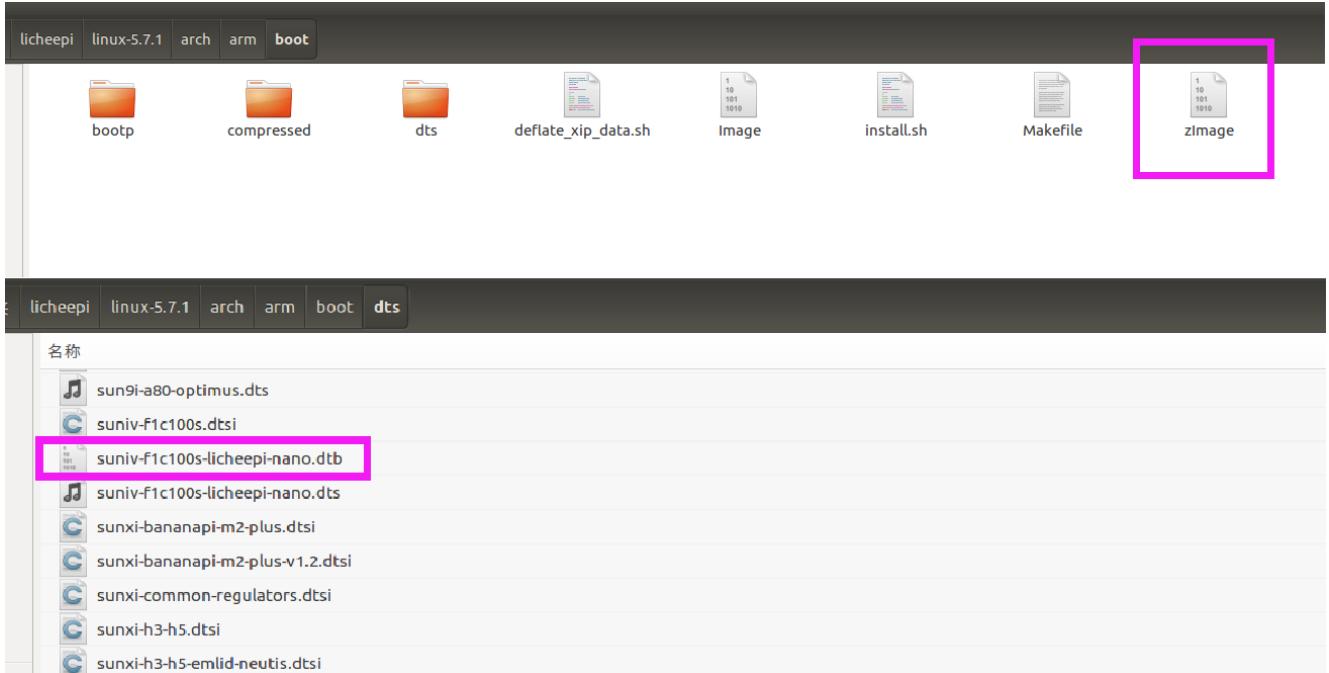


图 2.24

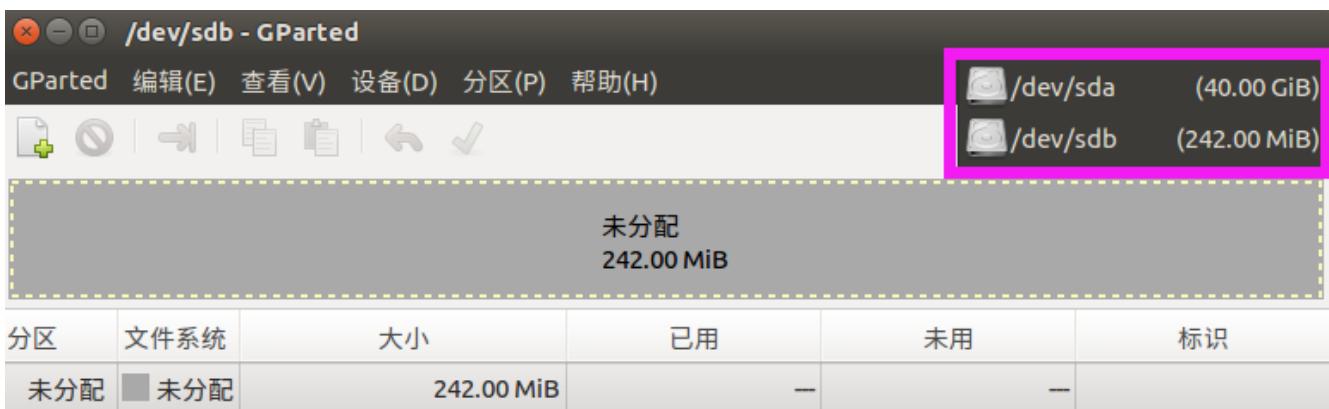


图 2.25

两个区，一个是存放zImage和dtb文件，另一个区存放根文件系统。对于第一个分区，一般格式为fat16格式，对于第二个区，一般为ext4格式。下面开始分区操作。

**A** 这里第一分区格式只能是fat16格式，原因在于在u-boot中对mmc的操作命令load只支持fat16格式，不支持ext2、3或者4格式，因此第一分区只能是fat16，对于第二分区，一般分区为ext4格式，原因在于一般Linux操作系统默认挂载的文件系统格式是ext类型的。

选中未分配空间并右击鼠标，点击[新建]，然后填写相关属性，如图2.26，然后点击[添加]，所示，以相同的方式新建第二分区，如图2.27所示，最后点击[对勾]完成分区操作，如图2.28，最终分区如



图 2.26

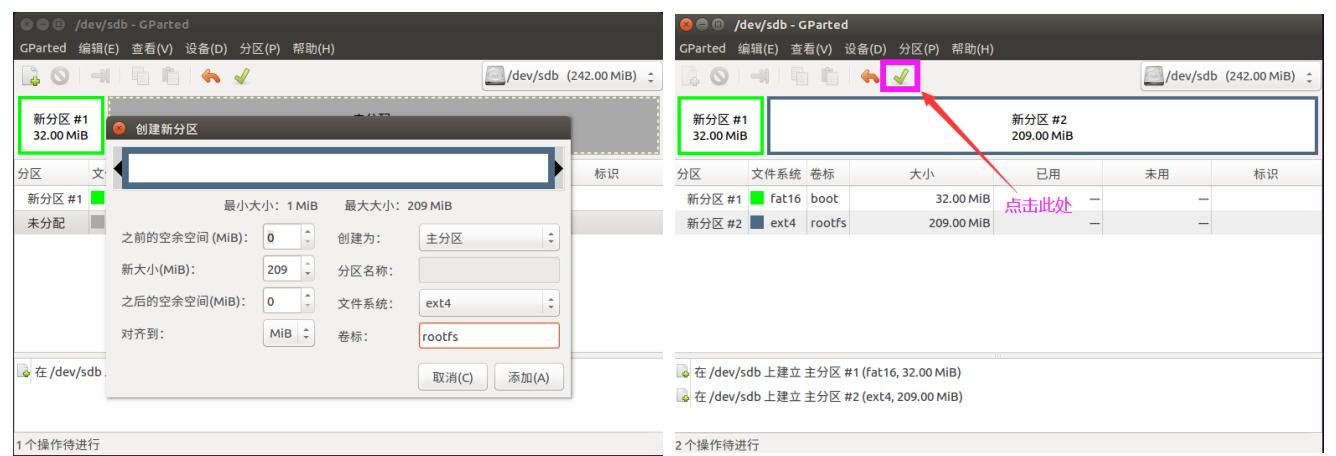


图 2.27

图 2.28

图2.29 分区完毕后剩下的就是将zImage和dtb文件复制到TF卡的BOOT分区中。此时复制可以直接通过图

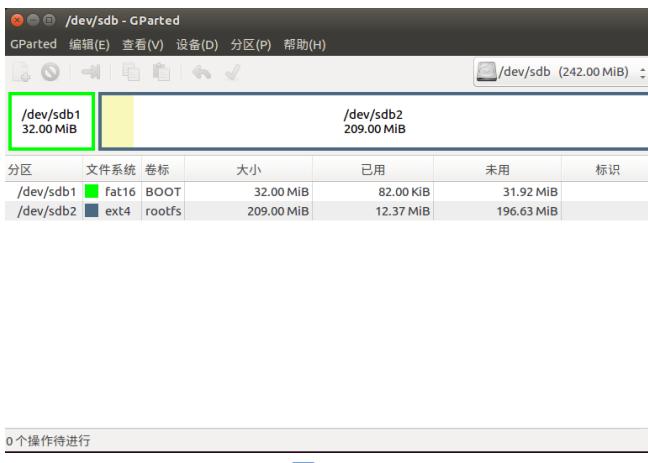


图 2.29

```
U-Boot 2018.01-05679-g013ca5-dirty (Feb 16 2021 - 10:26:34 +0800) Allwinner Technology
CPU: Allwinner F1 Series (SUN4V)
Model: Allwinner F1 Nano
DRAM: 64 MiB
NAND SD/MMC: 0
In: serial@2c9000
Out: serial@2c9000
Err: serial@2c9000
Net: none
mmc0: mmc0@2c9000
Hit any key to stop autoboot: 0
reading /stage
9096 bytes read in 347 ms (12.2 MiB/s)
reading suniv-f1c10h-l1cheep-nano.dtb
2496 bytes read in 25 ms (213.9 KiB/s)
TTF: f1c10h-l1cheep-nano.dtb@0x00000000
Booting using the fdt blob at 0x00080000
Loading Device Tree to 0x3e510800, end 0x3e5571 ... OK
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] DRAM: 64MB SDRAM @ 0x10000000
[2017/01/17 12:48:01 CST 2021] (sw@sw-VirtualBox) (gcc version 7.2.1 20170111 (Ubuntu 7.2-2017.11), GNU ld (Ubuntu Binutils-2017.11))
[    0.000000] CPU: ARM262x-S [41000000_rv32im_a15_m4f1], cr=000531f7
[    0.000000] CPU: 1000MHz, FPU: VFPv3, NEON, Function Cache
[    0.000000] OF: fdt: Machine model: Lichee F1 Nano
[    0.000000] U-Boot: 2018.01-05679-g013ca5-dirty
[    0.000000] Built-in list of aliases, memory grouping on. Total pages: 16256
[    0.000000] Kernel command line: console=ttyS0,115200 panic=1 root=/dev/mmcblk0p2 earlyprintk rw
[    0.000000] RAM: 64MB
[    0.000000] Indirect cache hash table entries: 4096 (order: 2, 16384 bytes, linear)
[    0.000000] Random number generator: stackoff, seed=0x12345678, heap free offset=0x10000000
[    0.000000] Mem: 64M total, 64M used, 0K edata, 160K rodata, 1024K init, 246K bss, 11372K reserved, 0K cma-reserved, 0K
[    0.000000] SLUB: Migrating, Order=3, MaxObjects=1, CPUs=1, Nodes=1
[    0.000000] NR_IRQS: 16, nr_irqs: 16, preallocated irqs: 16
[    0.000000] random: get_random_bytes called from start_kernel+0x400 with crng_init=0
[    0.000000] random: get_random_bytes called from start_kernel+0x400 with crng_init=0 every 804704007ms
[    0.000133] clocksource: timer: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 7961949 ns
```

图 2.30

形操作即可，也可以通过命令复制。将复制好内核的TF卡插到Lite200上接上USB和串口即可看到终端有信息输出，如图2.30

所示，可以看到内核已经成功启动，不过最终内核卡住了，其原因在于没有根文件系统，下节开始移植根文件系统。

**A** 上面仅仅是简单的启动了Linux内核，实际还有很多文件需要分析，但这里作为简单的了解大致的过程，故不做深入讲解，后面分析驱动的时候将会详细讲解内核的文件结构和设备树相关的驱动以及如何添加和修改驱动。

## 2.4 根文件系统移植

根文件系统是内核启动后挂载的第一个文件系统，如果没有根文件系统，内核将无法开启shell以及其他进程，下面来开始移植根文件系统。

**笔记** 实际上内核启动后会先挂载一个虚拟的文件系统，这个虚拟文件系统是在内存中运行的，其主要运行核心进程，虚拟文件系统挂载之后才挂载硬盘（TF卡或者emmc）上的根文件系统。

制作根文件系统的工具最有名的莫过于busyBox，该工具体积非常小，非常适合制作根文件系统。但是笔者尝试过使用busyBox制作，然而体积较大，这主要原因在于文件系统需要有动态库和静态库，而对于7.2版本的交叉编译器而言，其动态和静态链接库实在太大，因此本文将使用另一个非常强悍的工具—Buildroot，该工具集成了非常多的其他应用，制作过程相对简单，不会像busyBox那样出现硬件架构不兼容情况。

由于根文件系统制作比较简单，这里就完全从头开始。进入buildroot官网（<https://buildroot.org/downloads/>https://buildroot.org/downloads/https://buildroot.org/downloads/

下面以buildroot2018.2.11版本作为移植示例。将下载的源码包上传到虚拟机上，然后解压进入该源码目录中。进入源码目录后在终端输入

**make clean**

**A** 在开始编译之前必须执行**make clean**以清楚一些预设配置，即使是第一次编译也是一样。

然后执行

**make menuconfig**

此时会进入图形配置界面，如图2.31。进入第一个Target options选项，配置如图2.32 第一个选项为架构

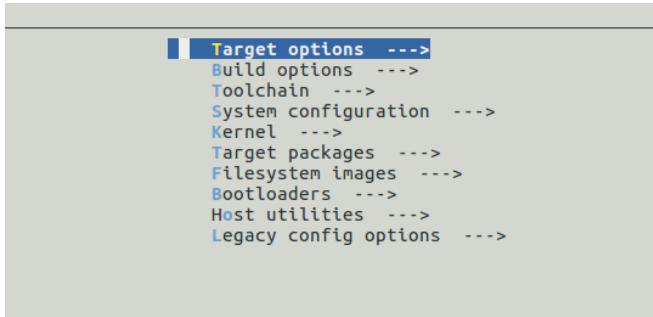


图 2.31

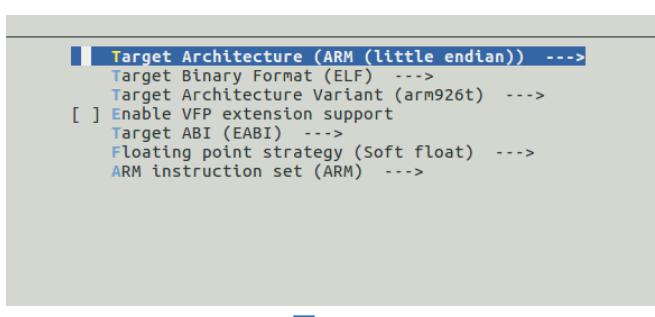


图 2.32

选择，这里选择ARM架构小端模式，第二个为输出的二进制文件格式，这里选择EFL格式，第三个为架构体系，这里选择arm926t，因为F1C100S的架构就是这个架构，第四个为矢量浮点处理器，这里不勾选，因为对于F1C100S而言，其内部没有浮点运算单元，只能进行软浮点运算，也就是模拟浮点预运算。第五个为应用程序二进制接口，这里选择EABI，原因是该格式支持软件浮点和硬件实现浮点功能混用。第六个为浮点运算规则，这里使用软件浮点，第七个选择指令集，这里选择ARM指令集，因为thumb主要针对Cortex M系列而言的，对于运行操作系统的A系列以及ARM9和ARM11而言，使用的都是32位的ARM指令集。

保存后，回到上一级配置界面，然后进入第二个Build options选项，配置如图2.33。保存后，回到上

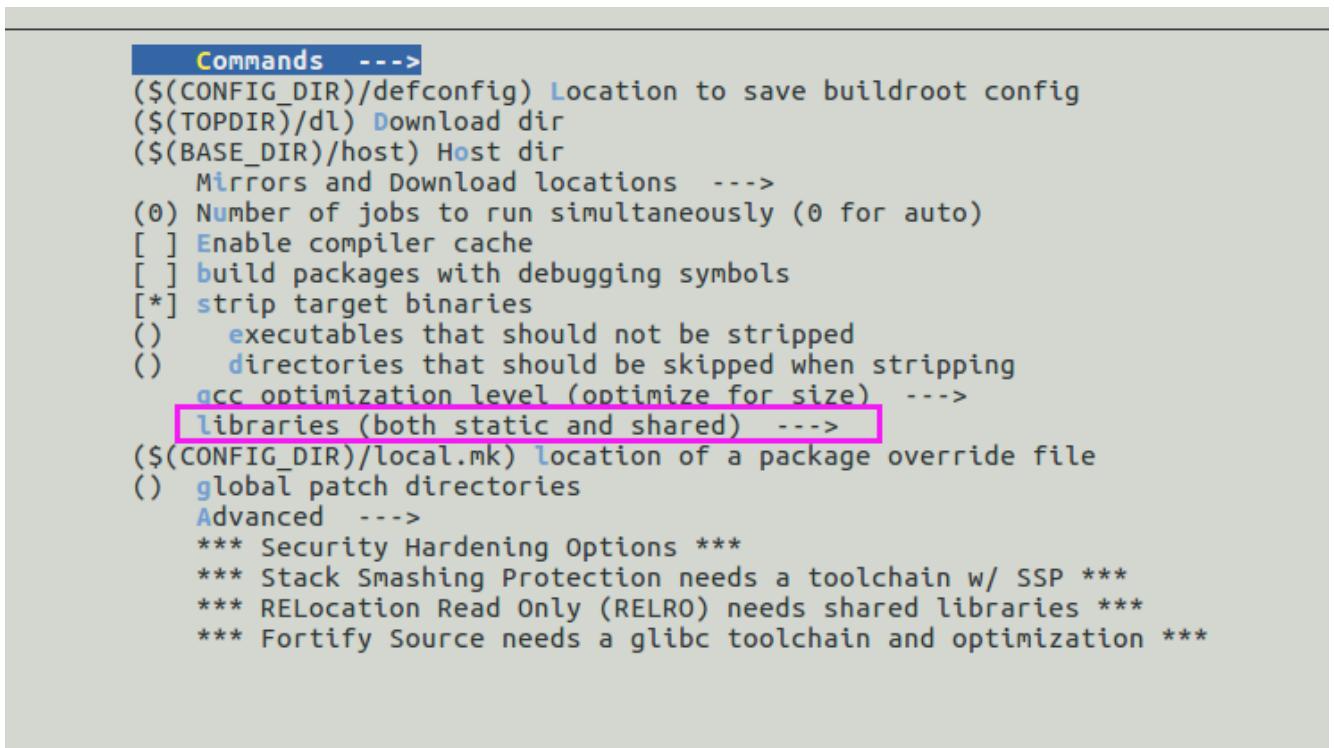


图 2.33

一级配置界面，然后进入第三个Toolchain选项，配置如图2.34。这里是选择交叉工具链，对于初学者而言，最好选择buildroot指定的默认交叉工具链，因为这里涉及到C库问题，如果使用自己安装的交叉工具链，编译很可能报错，因为使用的C库不匹配。黄色框中的选项尽可能勾选，因为后面移植QT5的时候需要用到C++相关库，如果这里没有勾选，QT5选项将无法勾选。

保存后回到上一级配置界面，然后进入第四个System configuration选项，配置如图2.35。第一个红色

```

Toolchain type (Buildroot toolchain) --->
  *** Toolchain Buildroot Options ***
  (buildroot) custom toolchain vendor name
    C library (uClibc-ng) --->
      *** Kernel Header Options ***
      Kernel Headers (Manually specified Linux version) --->
        () linux version (NEW)
        Custom kernel headers series (4.15.x) --->
          *** uClibc Options ***
          (package/uClibc/uClibc-ng.config) uClibc configuration file to use?
          () Additional uClibc configuration fragment files
          [*] Enable WCHAR support
          [ ] Enable toolchain locale/i18n support
            Thread library implementation (Native POSIX Threading (NPTL)) --->
              [*] Thread library debugging
              [ ] Enable stack protection support
              [*] Compile and install uClibc utilities
                *** Binutils Options ***
                Binutils Version (binutils 2.29.1) --->
              () Additional binutils options
                *** GCC Options ***
                GCC compiler Version (gcc 6.x) --->
              () Additional gcc options
                [*] Enable C++ support
                [*] Enable Fortran support
                [*] Enable compiler link-time-optimization support
                [*] Enable compiler OpenMP support
                [*] Enable graphite support
                *** Host GDB Options ***
                [ ] Build cross gdb for the host
                  *** Toolchain Generic Options ***
                [*] Enable MMU support
                () Target Optimizations
                () Target linker options
                [ ] Register toolchain within Eclipse Buildroot plug-in

```

图 2.34

```

| Root FS skeleton (default target skeleton) --->
  (buildroot) System hostname
  (Welcome to Buildroot) System banner
    Passwords encoding (md5) --->
    Init system (BusyBox) --->
    /dev management (Dynamic using devtmpfs only) --->
    (system/device_table.txt) Path to the permission tables
    [ ] support extended attributes in device tables
    [ ] Use svmlinks to /usr for /bin, /sbin and /lib
    [*] Enable root login with password
    () Root password
    /bin/sh (busybox' default shell) --->
    [*] Run a getty (login prompt) after boot --->
    [*] remount root filesystem read-write during boot
    () Network interface to configure through DHCP
    [*] Purge unwanted locales
    (C en_US) Locales to keep
    [ ] Enable Native Language Support (NLS) (NEW)
    [ ] Install timezone info
    () Path to the users tables
    () Root filesystem overlay directories
    () Custom scripts to run before creating filesystem images
    () Custom scripts to run inside the fakeroot environment
    () Custom scripts to run after creating filesystem images

```

图 2.35

框中表示启动根文件系统后输出的信息（横幅），这里默认，当然你也可以修改此值，比如改为Welcome to Lite200，第二个红色框中表示开启登录密码，可以看到默认密码为空，这里就默认了，读者也可以根据自己情况修改即可。

保存后回到上一级配置界面，可以看到后面还有部分选项没有配置，由于剩下的选项也可以不用配置，因此这里为了简便，直接保持推出，然后执行

**make**

**⚠** 这里最好不要用多核编译，一般来说，首次编译过程非常慢，因为要下载很多必要的文件和交叉工具链。

编译完毕后可以在output/images目录下有一个rootfs.tar，该文件就是最终生成的根文件系统镜像，现在只需要将该镜像解压到TF卡的第二分区即可。插入TF卡到电脑端，进入out/images目录，然后输入

```
sudo tar -xvf rootfs.tar -C /media/lsw/rootfs/
```

此时可以看到TF卡的rootfs分区中有文件系统了，现在将TF卡弹出，插入到Lite200上，连接好串口，打开串口助手或者其他串口终端软件，可以看到根文件系统成功挂载，同时进入shell交互，如图2.36。用

```
[ 1.205557] mmcblk0: p1 p2
[ 1.327033] random: fast init done
[ 1.498825] EXT4-fs (mmcblk0p2): recovery complete
[ 1.506330] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
[ 1.514656] VFS: Mounted root (ext4 filesystem) on device 179:2.
[ 1.521780] devtmpfs: mounted
[ 1.529265] Freeing unused kernel memory: 1024K
[ 1.534008] Run /sbin/init as init process
[ 1.696509] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
Starting logging: OK
Initializing random number generator... [ 2.039480] random: dd: uninitialized urandom read (512 bytes read)
done.
Starting network: OK
Welcome to Myboard
This login: ■
```

图 2.36

用户名默认为root，无密码，进入root账号后，在终端输出

**cd /**

**ls**

可以看到文件系统中的linux的根目录情况，到此根文件系统的移植完成。

## 第三章 设备树(DTS)

设备树是什么？又为何要引入设备树呢？这个需要追溯到2011年3月17日。这天Linus Torvalds对ARM Linux邮件列表宣称“this whole ARM thing is a fucking pain in the ass”，ARM Linux社区对此作出了回应，由此引入设备树。在未引入设备树之前，Linux源码中充斥这非常多的驱动代码，这些驱动代码虽然完整的适应很多处理器架构，但是由于处理器种类太多，而且每种处理器又分了非常多不同型号的处理器，这使得相应的驱动文件异常的多，Linus称其为垃圾代码。其后，Linux源码开始引入设备树，将不同型号的处理器驱动尽可能放在一起，为了方便管理，他们将所有的设备都描述成一棵树，Linux内核启动之后就会遍历这棵树，然后初始化相应的硬件。在设备树引入之前，Platform总线也是非常好的一个设备驱动框架，读者有兴趣可以深入了解，这种框架当前仍然还保留在，而且当前几乎大部分驱动都是Platform总线结合设备树一起管理驱动设备。

在Linux发展过程中总共经历了三种设备驱动管理框架，第一种是最原始的基于简单的注册，这种方式比较容易理解，需要什么设备驱动就将相应的驱动写在内核中，然后编译即可载入内核。这种方法缺陷很明显，当设备越来越多时，驱动文件将变得很大。最后PowerPC使用了一种基于Platform Bus的框架，这种框架改善的上面的情况，它将处理器与设备之间的交互虚拟成一根总线，相同类型的驱动使用同一根总线，比如IIC,IIS,SPI,USB等等，所有的设备都是这些总线上的子设备。第三种就是现在的基于设备树的驱动框架，这种驱动框架结合了Platform Bus的优点，同时引入了设备树概念，其核心思想是将处理器看作是一个数的主干，主干上有很多分支，分支上挂载着不同的设备。

需要说明的是设备树并不是驱动文件，它指示一个描述设备的表格，而真正的驱动代码会根据设备树选择的设备对相应的设备进行初始化，设备树上没有的设备不会初始化。这举个简单的例子：现在我们需要对两个数进行四则运算，但是我们暂时不知道具体是哪一种运算，那我们就可以这样写

```
1 int fun(int style,int x,int y)
2 {
3     switch(style)
4     {
5         case 0:return x+y;
6         case 1:return x-y;
7         case 2:return x*y;
8         case 3:return x/y;
9         default:return -1;
10    }
11 }
```

如果要进行四则运算，dts可以看作是选择其中一个运算方式。图3.1所示是dts的示意图，可以看到每个设备都是通过总线的方式描述的。图3.2，dts文件需要被dtc编译器编译为dtb的二进制文件，当内核获取dtb文件后会对文件进行解析，然后根据dtb文件的内容对相应的设备进行初始化以及注册。

**▲**一般而言，dts会分为两个文件，一个是dtsi文件，另一个是dts文件。dtsi文件类似于.h文件一样，dts文件一般包含dtsi文件，然后根据需要来配置设备节点。

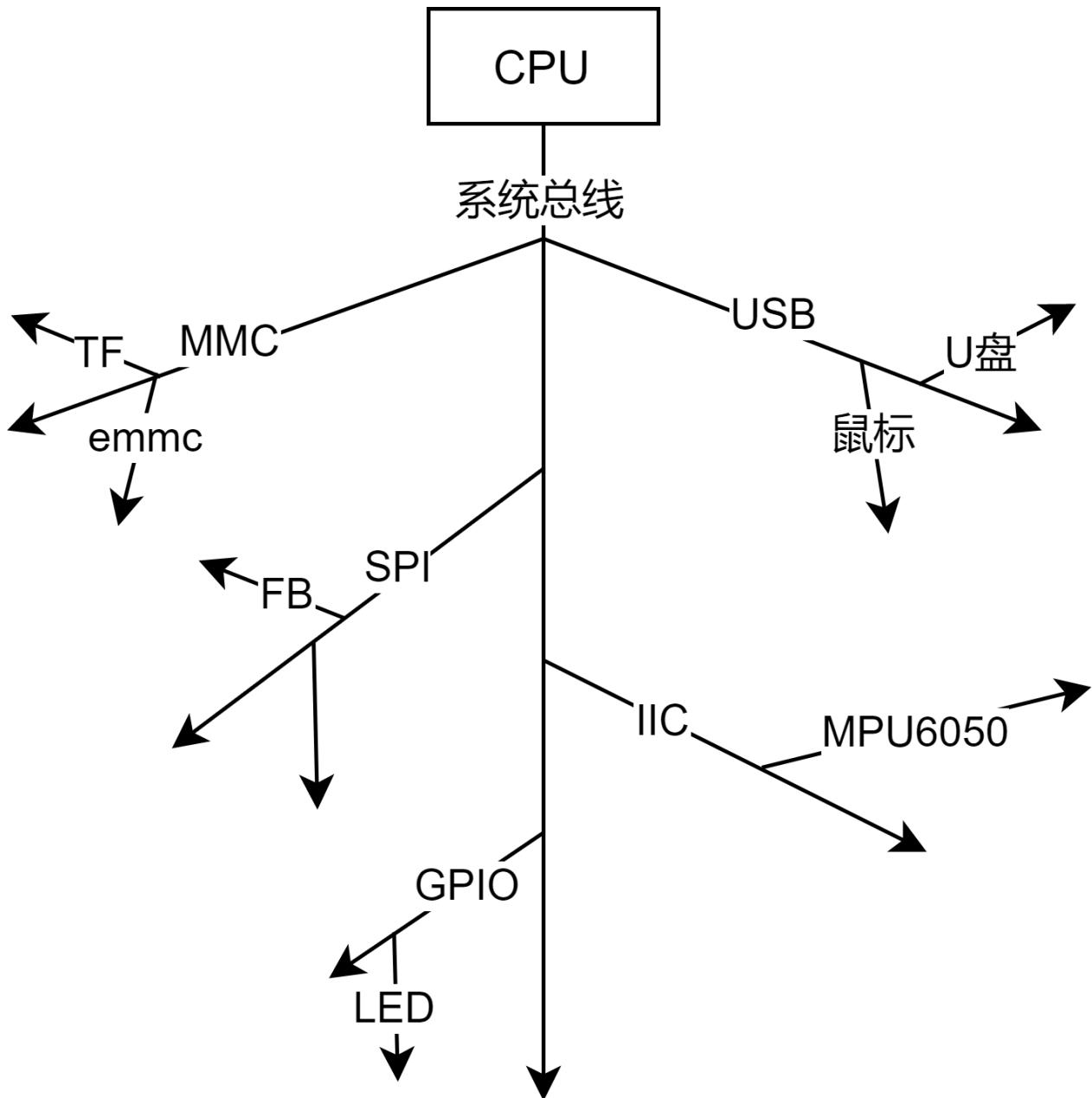


图 3.1

### 3.1 dts文件结构

先来看下suniv-f1c100s的dtsi文件，打开Linux5.7源码中的arch/arm/boot/dts/suniv-f1c100s.dtsi文件

```

1 #include <dt-bindings/clock/suniv-ccu-f1c100s.h>
2 #include <dt-bindings/reset/suniv-ccu-f1c100s.h>
3 /
4 #address-cells = <1>;
5 #size-cells = <1>;
6 interrupt-parent = <&intc>;
7 clocks {
8     osc24M: clk-24M {
  
```

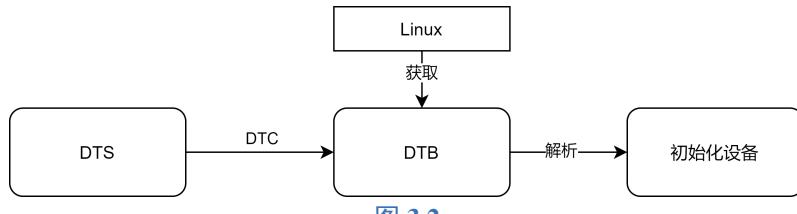


图 3.2

```

9      #clock-cells = <0>;
10     compatible = "fixed-clock";
11     clock-frequency = <24000000>;
12     clock-output-names = "osc24M";
13 };
14
15 osc32k: clk-32k {
16     #clock-cells = <0>;
17     compatible = "fixed-clock";
18     clock-frequency = <32768>;
19     clock-output-names = "osc32k";
20 };
21 };
22
23 cpus {
24     cpu {
25         compatible = "arm,arm926ej-s";
26         device_type = "cpu";
27     };
28 };
29
30 soc {
31     compatible = "simple-bus";
32     #address-cells = <1>;
33     #size-cells = <1>;
34     ranges;
35
36     sram-controller@1c00000 {
37         compatible = "allwinner,suniv-f1c100s-system-control",
38         "allwinner,sun4i-a10-system-control";
39         reg = <0x01c00000 0x30>;
40         #address-cells = <1>;
41         #size-cells = <1>;
42         ranges;

```

```

43
44     sram_d: sram@10000 {
45         compatible = "mmio-sram";
46         reg = <0x00010000 0x1000>;
47         #address-cells = <1>;
48         #size-cells = <1>;
49         ranges = <0 0x00010000 0x1000>;
50
51     otg_sram: sram-section@0 {
52         compatible = "allwinner,suniv-f1c100s-sram-d",
53         "allwinner,sun4i-a10-sram-d";
54         reg = <0x0000 0x1000>;
55         status = "disabled";
56     };
57 };
58 };
59
60 ccu: clock@01c20000 {
61     compatible = "allwinner,suniv-f1c100s-ccu";
62     reg = <0x01c20000 0x400>;
63     clocks = <&osc24M>, <&osc32k>;
64     clock-names = "hosc", "losc";
65     #clock-cells = <1>;
66     #reset-cells = <1>;
67 };
68 };
69 };

```

由于文件太大，上面的dtsi文件只放一部分，可以看到里面有很多关于设备的描述信息，下面将重点讲解其意义。1-2行是包含的头文件，该头文件中定义了时钟宏和复位相关的宏，这些宏可以在dtsi和dts文件中看到，例如osc24M等，这两个头文件在include/dt-bindings/clock和include/dt-bindings/reset目录里面。

一般默认规定，dtsi文件放一个系列芯片的公共部分，然后dts文件对其进行配置。F1C200S的dtsi文件中说明了几乎所有的硬件描述，dts文件中对各个硬件节点进行配置。这样对于不同开发板而言，只需要编写dts文件即可完成配置，当然，前提是dtsi文件中的硬件描述是完整的。

### 3.1.1 节点

所谓节点是针对总线而言的，将根节点作为一个树的根，然后在根节点上添加其他的节点。上述给的dtsi文件中/就是根节点，根节点里面会添加各个不同的节点，对于一个dts文件而言，有且只有一个根节点。从上面可以看到有些节点前面有：，例如clock节点：

```
1     ccu: clock@01c20000 {
```

```

2   compatible = "allwinner,suniv-f1c100s-ccu";
3   reg = <0x01c20000 0x400>;
4   clocks = <&osc24M>, <&osc32k>;
5   clock-names = "hosc", "losc";
6   #clock-cells = <1>;
7   #reset-cells = <1>;
8 };

```

该节点的节点名为clock，地址为0x1c20000，因此，节点的表示为：节点名&地址。

**▲** 这里的节点名@后面的地址并不是真实的地址，可以随便写一个，但是为了不发生重复，一般默认规定其值为该寄存器的地址。只有reg属性中的地址才是真实描述寄存器的地址。

这里的ccu是一个节点标号，所谓节点标号可以看作是该节点的一个别名，这样做的好处是当需要修改节点的属性时，只需要在dts文件中引用该标号，然后添加修改的属性即可，而不需要重复将该节点完整的写一遍。例如现在需要将ccu节点中的属性clocks修改为<&osc32M>, <&osc32.768>；，此时只需要在dts文件中这样写即可：

```

1 &ccu{
2   clocks = <&osc32M>, <&osc32.768k>;
3 };

```

 **笔记** 一般需要修改节点的status属性以及引脚属性。

### 3.1.2 属性

每个节点必须要有属性，所谓属性就是添加一些节点的必要信息，比如寄存器地址，时钟等，这些属性是为驱动程序提供具体的相关信息的，下面来简单讲解下一般接触到的节点属性。

#### 3.1.2.1 compatible

"compatible"表示“兼容”，对于某个LED，内核中可能有A、B、C三个驱动都支持它，那可以这样写：

```
led{compatible = "A", "B", "C";};
```

内核启动时，就会为这个LED按这样的优先顺序为它找到驱动程序：A、B、C。也可以从上面的clock节点可以看到其兼容属性为

```
compatible = "allwinner,suniv-f1c100s-ccu";
```

内核会通过of\_match\_table(也有通过of\_find\_compatible\_node)来匹配对应的节点，一旦匹配到就加载该驱动。因此当前大部分的驱动都需要使用compatible 节点来加载驱动程序，以降低不必要的代码量。有这么一个默认的规定，compatible后面一般格式为“厂家，驱动名称”，这只是一个约定俗成的规定，不一定非要这样写，只要驱动程序能够与dts文件中的节点属性对应起来就可以。对于有些节点可以兼容多个设备时，可以写在一起，例如：

```
compatible = "allwinner,suniv-f1c100s-ccu", "allwinner,suniv-f1c200s-ccu";
```

### 3.1.2.2 model

"model"字面意思是模型，主要来说明该设备的具体名称，例如上面我们可以添加一个model属性：

```

1 ccu: clock@01c20000 {
2     compatible = "allwinner,suniv-f1c100s-ccu";
3     model = "lite200";
4     reg = <0x01c20000 0x400>;
5     clocks = <&osc24M>, <&osc32k>;
6     clock-names = "hosc", "losc";
7     #clock-cells = <1>;
8     #reset-cells = <1>;
9 };

```

上面添加了一个model = "lite200";属性来说明此设备是lite200，一般而言，此驱动会被加载打印出来，不过并不是必须的，也可以没有此属性。

### 3.1.2.3 status

"status"可以看出这个属性是状态的意思，在dts节点中表示此节点是否有效，其属性有大多数情况下

表 3.1: status属性值

|          |                     |
|----------|---------------------|
| okay     | 表示设备可操作             |
| disabled | 表示设备不可操作            |
| fail     | 表示设备不可操作，同时检测到设备错误  |
| fail-sss | 表示设备不可操作，同时检测到sss错误 |

接触的只有okay和disabled这两个属性。

### 3.1.2.4 #address-cells #size-cells

address-cells表示该节点的地址要用多少个32位数来表示，size-cells表示该节点的大小要用多少个32位数来表示。这两个属性与后面的reg属性比较密切，因为reg属性需要用这两个属性来说明其寄存器所占用的空间大小。例如：

```

1 sram_d: sram@10000 {
2     compatible = "mmio-sram";
3     reg = <0x00010000 0x1000>;
4     #address-cells = <1>;
5     #size-cells = <1>;
6     ranges = <0 0x00010000 0x1000>;
7
8     otg_sram: sram-section@0 {
9         compatible = "allwinner,suniv-f1c100s-sram-d",
10        "allwinner,sun4i-a10-sram-d";

```

```

11     reg = <0x0000 0x1000>;
12     status = "disabled";
13 };
14 };

```

上面的

```
#address-cells = <1>;
#size-cells = <1>;
```

这两个表示reg的地址占一个32位数，即4个字节，大小占一个32位数，即4个字节。这样就可以清楚的知道reg = <0x00010000 0x1000>;的意思是寄存器的地址位0x00010000，大小为0x1000。

**▲ #address-cells和#size-cells属性不是从devicetree的祖先继承的。它们需要明确定义，如果未定义，对于设备树则默认按照地址cell为2个，长度cell为1个去解析reg的值。**

### 3.1.2.5 ranges

该属性一般为空，如果不为空，则表示一个地址映射表或者叫地址转换表，其一般

## 3.2 设备树与驱动关系

## 第四章 常用驱动移植

本章主要讲解常用的驱动移植。

**4.1 屏幕驱动移植**

**4.2 USB驱动移植**

**4.3 音频解码驱动移植**

**4.4 视频解码驱动移植**

## 第五章 常用应用移植

5.1 QT5移植

5.2 图片浏览器移植

5.3 NES模拟器移植

5.4 视频播放器mplayer移植

# 第六章 驱动开发

6.1 linux驱动框架

6.2 字符设备驱动

6.3 块设备驱动

6.4 网络设备驱动

## 第七章 深入理解C语言

7.1 C指针

7.2 C语言与汇编的关系

7.3 C语言的编译过程

7.4 C语言的运行过程