Linux 开发环境

用户指南

文档版本　11

发布日期　2018-12-03

商标声明

 、HISILICON、海思和其他海思商标均为海思技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

上海海思技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： http://www.hisilicon.com/cn/

客户服务邮箱： support@hisilicon.com

前言

概述

本文档介绍海思Linux SDK开发环境，包括Linux服务器搭建、SDK编译、应用程序的开发等。

本文档主要提供让客户更快地了解Linux开发环境指导。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3716M	V41X/V42X/V43X
Hi3716D	V1XX
Hi3798M	V1XX
Hi3796M	V1XX
Hi3798C	V2XX
Hi3798M	V2XX（H）
Hi3796M	V2XX
Hi3798M	V3XX（H）

读者对象

本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

作者信息

章节号	章节名称	作者信息
全文	全文	L00227819

修订记录

修订记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

修订日期	版本	修订说明
2014-06-09	00B01	第1次版本发布。
2014-10-30	01	新增支持Hi3796MV100芯片。
2015-04-25	02	新增支持Hi3798CV200芯片。
2015-12-16	03	刷新兼容Hi3798CV200方案。插入第2章Fastboot配置。
2016-07-18	04	全面刷新文档。
2016-11-04	05	新增支持Hi3798MV200芯片。
2017-04-05	06	新增支持Hi3796MV200芯片。
2017-08-31	07	新增支持Hi3798MV300芯片。
2018-03-09	08	修改4.1章节。添加“清除编译内核配置文件时产生的临时文件”的说明。
2018-05-14	09	修改3.5、3.6、5.6章节。
2018-10-17	10	新增4.4、5.8章节和第6章；修改3.2、3.6、4.1、4.2、5.7章节。
2018-12-03	11	2.4章节新增一个说明。

目 录

1 开发环境搭建

1.1 Linux SDK 开发环境简介

典型的Linux SDK开发环境通常包括：

- Linux服务器

  Linux服务器主要用于建立交叉编译环境。

- 工作台

工作台主要用于目标板烧录和调试，通过串口与目标板连接，开发人员可以在工作台中烧录目标板的镜像、调试程序。工作台通常需要安装终端工具，用于登录Linux服务器和目标板，查看目标板的打印输出信息。

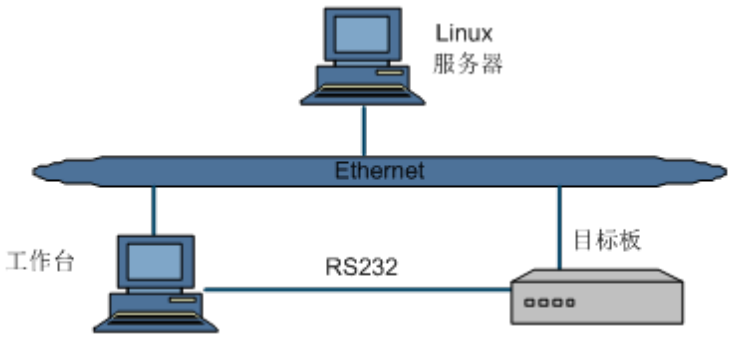
工作台与目标板需要处于同一网络环境中，以便调试文件的传输，而Linux服务器与目标板可以不处于一个网络，但需要通过工作台传输文件，当然，推荐三者处于同一个网络，方便调试。

工作台一般为Windows或Linux操作系统。

- 目标板

  本文的目标板以海思的DEMO板为例，DEMO板通常包含了网口和串口，用于开发者调试。DEMO板的镜像通过串口和网口烧录到DEMO板。如图1-1所示。

图 1-1 SDK 开发环境



说明

在Windows或Linux工作台运行的终端工具，通常有SecureCRT/Putty/miniCom等，这些软件需要从其官网下载。

1.2 搭建 Linux SDK 开发环境

推荐使用Ubuntu 14.04，Shell必须为bash。

在Ubuntu 14.04系统上安装编译SDK依赖的开源组件：

```
sudo apt-get install gcc make gettext zlib1g-dev libncurses5-dev bison
flex bc texinfo tree cmake
```

修改linux服务器默认sh为bash的方法：

```
rm -f /bin/sh
ln -s /bin/bash /bin/sh
```

2 编译和运行 SDK

2.1 安装 SDK

SDK版本命名规则如下：

- SDK基础版本开发包命名为：HiSTBLinuxV100R00XC0XSPC0X0.tar.gz，如HiSTBLinuxV100R005C00SPC010.tar.gz、HiSTBLinuxV100R005C00SPC020.tar.gz
- SDK累积补丁版本开发包命名为：HiSTBLinuxV100R00XC0XSPC0XX.tar.gz，如HiSTBLinuxV100R005C00SPC011.tar.gz、HiSTBLinuxV100R005C00SPC012.tar.gz
- SDK紧急补丁版本开发包命名为：HiSTBLinuxV100R00XC0XSPC0XXCP00XX.tar.gz，如HiSTBLinuxV100R005C00SPC010CP0001.tar.gz、HiSTBLinuxV100R005C00SPC010CP0002.tar.gz

注意

累积补丁和紧急补丁版本必须按顺序打到基础版本上，才可以正常编译运行。

例如：

SDK基础版本为HiSTBLinuxV100R005C00SPC010.tar.gz

SDK累积补丁版本为HiSTBLinuxV100R005C00SPC011.tar.gz、HiSTBLinuxV100R005C00SPC012.tar.gz

打补丁的方法：

```
tar -xf HiSTBLinuxV100R005C00SPC010.tar.gz
tar -xf HiSTBLinuxV100R005C00SPC011.tar.gz
tar -xf HiSTBLinuxV100R005C00SPC012.tar.gz
cd HiSTBLinuxV100R005C00SPC011
./patch.sh ../HiSTBLinuxV100R005C00SPC010
cd HiSTBLinuxV100R005C00SPC012
./patch.sh ../HiSTBLinuxV100R005C00SPC010
```

2.2 SDK 目录结构介绍

SDK的重点目录结构如表2-1所示。

表 2-1 SDK 的重点目录结构

目录	说明
configs	SDK参考配置文件目录
sample	SDK参考sample
scripts	SDK Kconfig文件的目录
source/boot	SDK Boot代码和Boot表格的目录
source/kernel	SDK使用的Linux Kernel
source/common	SDK的公共代码，包括内存管理、日志系统等
source/msp	SDK驱动相关的代码，包括图形、音视频编解码、音视频输出、外设等
source/component	SDK组件目录，包括音频解码库、wifi、hiplayer、ntfs等
source/rootfs	SDK文件系统目录，包括busybox、gdb等
source/tee	SDK安全OS及相关驱动、应用程序目录
third_party	SDK使用的第三方软件目录
tools/linux	SDK提供的Linux系统上使用的工具
tools/windows	SDK提供的Windows系统上使用的工具
out	SDK编译时自动创建的目录，存放编译时生成的镜像、内核驱动模块、文件系统、库文件、头文件等

2.3 编译 SDK

编译SDK时需要根据目标板类型选择对应的SDK配置文件，编译生成的文件放在：out/$CFG_HI_CHIP_TYPE/$CFG_HI_OUT_DIR/目录下，其中变量CFG_HI_CHIP_TYPE和CFG_HI_OUT_DIR在SDK配置文件中定义。

本文以HI3798CV2DMB单板的SDK配置文件hi3798cv2dmo_hi3798cv200_cfg.mak为例：

配置编译环境，指定交叉编译器和工具的路径，执行：

```
cd sdk
source ./env.sh
```

编译SDK方法一：

```
cp configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak cfg.mak
make build -j 2>&1 | tee bulid.log
```

编译SDK方法二：

```
make SDK_CFGFILE=configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak
build -j 2>&1 | tee bulid.log
```

编译成功后生成的文件在out/hi3798cv200/hi3798cv2dmo目录，镜像文件在out/hi3798cv200/hi3798cv2dmo/image目录，需根据目标板上的FLASH类型选择对应的镜像目录。例如：

- 目标板上只有eMMC Flash，烧写emmc_image目录下的镜像文件到目标板。
- 目标板上只有NAND Flash，烧写nand_image目录下的镜像文件到目标板。
- 目标板上有SPI Flash和eMMC Flash，烧写spi_emmc_image目录下的镜像文件到目标板，这种情况下通常是Boot烧写到SPI Flash，内核和文件系统烧写到eMMC Flash。

注意

SDK支持多线程编译，但在虚拟机或性能较差的服务器上编译SDK时，建议单线程编译或指定编译线程的个数，使用单线程编译：make build；使用两个线程编译：make build -j 2。

SDK多线程编译失败时打开build.log搜索unfinished、Error、error:、undefined、missing等关键字确认编译失败的原因。

2.4 运行 SDK

SDK编译成功后生成的镜像文件在out/hi3798cv200/hi3798cv2dmo/image目录，参考《HiTool工具平台 使用指南》将镜像烧写到目标板。

镜像烧写成功后，重启目标板进入Linux系统，提示如下信息表明已成功进入Linux系统：

```
Welcome to HiLinux.
#
```

运行SDK提供的sample，使用Linux NFS服务器挂载到目标板运行的方法：

```
# udhcpc
# mount -t nfs -o nolock -o tcp nfs_server_ip:/home/test /mnt
# cd /mnt/sdk
# cd out/hi3798cv200/hi3798cv2dmo/obj/sample/tsplay
# ./sample_tsplay /mnt/ts/test.ts
```

说明

编译对Flash存储空间不敏感的SDK配置文件（CFG_HI_ADVCA_SUPPORT=n）时，默认会将编译的sample拷贝到单板的/usr/local/sample目录，可以到该目录下执行对应sample。

2.5 SDK 配置文件

SDK参考配置文件存放在configs目录，按芯片区分。

SDK参考配置文件的命名方法：单板名称_芯片名称_特性_cfg.mak，如：hi3798cv2dmb_hi3798cv200_tee_cfg.mak。

```
查看和修改SDK配置文件的方法一：
cp configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak cfg.mak
make menuconfig
查看和修改SDK配置文件的方法二：
make SDK_CFGFILE=configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak
menuconfig
```

在menuconfig配置菜单中参考help信息修改配置选项。

3 Boot

3.1 Boot 简介

3.1.1 如何进入 Boot 命令行

- 在fastboot启动过程中，出现“Press Ctrl+C to stop autoboot”提示时，按Ctrl+C进入fastboot命令行模式。

  fastboot命令行提示符为：

  ```
  fastboot#
  ```

- 在miniboot启动过程中，出现“Press Ctrl+C quit autoboot 0”提示时，按Ctrl+C进入miniboot命令行模式。

  miniboot命令行提示符为：

  ```
  miniboot#
  ```

3.1.2 如何获取 fastboot 支持的命令

在miniboot中输入？可以查看miniboot支持的所有命令，如：

```
miniboot# ?
bootm       load image.
clearenv    clear one environment value.
md          memory display sub-system.
mm          memory sub-system.
mmc         MMC sub-system
```

输入command ?可以查看command命令的详细帮助信息，如：

```
miniboot# bootm ?
<address>   uImage start address.
```

3.1.3 如何获取 miniboot 支持的命令

```
在miniboot中输入？可以查看miniboot支持的所有命令，如：
miniboot# ?
bootm       load image.
clearenv    clear one environment value.
md          memory display sub-system.
mm          memory sub-system.
mmc         MMC sub-system
```

```
输入command ?可以查看command命令的详细帮助信息，如：
miniboot# bootm ?
<address>   uImage start address.
```

3.2 Boot 环境变量

Boot环境变量是指Boot运行时的环境变量和引导内核启动的参数配置，通常会在Flash划分一块区域来存储Boot环境变量，即bootargs分区。

fastboot启动时会打印出bootargs分区的位置和大小，如：

```
Boot Env on eMMC
    Env Offset:          0x00100000
    Env Size:            0x00010000
```

miniboot启动时会打印出bootargs分区的位置和大小，如：

```
Env: Load Env from eMMC, start:0x100000, size:0x10000
```

3.2.1 Boot 网络配置

```
配置目标板的MAC地址：
setenv ethaddr xx:xx:xx:xx:xx:xx
配置目标板的IP地址：
setenv ipaddr xxx.xxx.xxx.xxx
配置目标板的子网掩码：
setenv netmask 255.255.xxx.0
配置目标板的网关：
setenv gatewayip xxx.xxx.xxx.xxx
```

配置目标板tftp服务器的IP地址：

```
setenv serverip xxx.xxx.xxx.xxx
```

配置完成后使用ping命令测试目标板的网络环境是否正确。

说明

miniboot不支持网络，不需要配置。

3.2.2 bootargs 参数配置

bootargs是Boot传给内核的启动参数，其中包含了内存大小、串口、根文件系统和分区表等配置。

Boot启动过程中读取目标板的内存大小，选择对应的bootargs内存配置变量与booargs变量组合成bootargs传递给内核。

以下是bootargs的示例：

```
bootargs 1G=mem=1G mmz=ddr,0,0,48M vmalloc=500M
bootargs 2G=mem=2G mmz=ddr,0,0,48M vmalloc=500M
bootargs=console=ttyAMA0,115200 root=/dev/mmcblk0p10 rootfstype=ext4
rootwait mtdparts=hi_sfc:1M(boot),1M(bootargs)
blkdevparts=mmcblk0:4M(baseparam),4M(pqparam),           4M(logo),
4M(deviceinfo),4M(softwareinfo),4M(loaderdb),32M(loader),
16M(trustedcore),32M(kernel),384M(rootfs),-(others)
```

3.2.2.1 bootargs 内存配置变量

Boot支持不同DDR容量的内存配置，支持bootargs_512M、bootargs_1G、bootargs_2G、bootargs_768M、bootargs_1536M等，启动时读取目标板的DDR容量选择对应的内存配置。如目标板DDR容量为1G，则使用bootargs_1G。

bootargs内存配置变量有如下配置选项：

- mem —— 配置内核可用内存大小，一般配置为DDR的实际容量，部分芯片平台使用DDR高端内存做固定TEE使用的保留内存，例如，TEE使用65MB保留内存，则mem的值应该是DDR实际大小减去65MB，不同的芯片平台和版本可能调整TEE保留内存，请参考对应的SDK版本默认bootargs配置。
- mmz —— 从内核可用内存中划分出一段物理上连续的内存，提供给需要使用连续物理内存的模块使用。
- vmalloc —— 32位内核时需要指定vmalloc空间的大小。vmalloc空间是内核态的CPU虚拟地址空间，主要用于ioremap、vmalloc分配的内存、内核态映射的MMZ或是SMMU内存场景下使用的虚拟地址。MMZ和SMMU需要较大的虚拟地址空间，内核默认配置的256MB可能不够用，因此需要调整，实际需要的大小跟不同的业务场景有关系，建议以SDK默认的bootargs配置为准。

bootargs内存配置变量的配置方法可参考configs\芯片名称\prebuilts\bootargs.txt。

3.2.2.2 bootargs 变量

bootargs变量有如下配置选项，完整的bootargs配置参见**3.2.2 bootargs参数配置**节示例：

- console —— 配置内核的打印控制台，一般配置为ttyAMA0,115200，表示使用串口0作为控制台设备，波特率为115200。
- root —— 指定根文件系统所在的设备。
  - UBI根文件系统时配置为root=ubi0:ubifs ubi.mtd=rootfs；
  - mtdparts分区表中的分区从0开始编号，设备为/dev/mtdblockX，X为根文件系统的分区号；
  - blkdevparts分区表中的分区从1开始编号，设备为/dev/mmcblk0pX，X为根文件系统的分区号。
- rootfstype —— 指定根文件系统的类型，支持ext4、squashfs、cramfs、ubifs、yaffs2、jffs2。
- rootwait —— 表示等待根文件系统所在的设备就绪后再尝试挂载根文件系统，只有eMMC Flash增加此选项。
- Flash分区表 —— 根据目标板的Flash类型配置。
  - SPI Flash分区表格式：mtdparts=hi_sfc:size(name),size(name),...
  - NAND Flash分区表格式：mtdparts=hinand:size(name),size(name),...
  - 同时使用SPI Flash和NAND Flash的分区表格式：

    mtdparts=hi_sfc:size(name),size(name),...;hinand:size(name),size(name),...
  - eMMC Flash分区表格式：blkdevparts=mmcblk0:size(name),size(name),…

    其中：

    size(name)表示一个分区，size指定分区的大小，name指定分区的名字

bootargs变量的配置方法可参考configs\芯片名称\prebuilts\bootargs.txt。

3.2.3 bootcmd 变量

bootcmd变量用于引导内核，配置方法可参考configs\芯片名称\prebuilts\bootargs.txt。

3.3 Boot 常用命令

3.3.1 printenv 命令

打印出所有的Boot环境变量。

3.3.2 setenv 命令

设置或修改Boot环境变量。

setenv命令的格式：setenv 变量名 变量值

其中：

- 变量值中有空格时需要使用单引号将变量值包含起来；
- 变量值中有多个命令时使用分号分隔；
  例：
  setenv bootcmd 'mmc read 0 0x1FFFFC0 0x33000 0x10000;bootm 0x1FFFFC0'

3.3.3 saveenv 命令

将Boot环境变量保存到bootargs分区。

3.3.4 reset 命令

重启Boot。

3.3.5 tftp 命令

fastboot支持tftp客户端，可以从tftp服务端下载文件到指定地址的DDR区域，或者将指定DDR地址的内容上传到服务端。

tftp命令的用法：

```
tftp [loadAddress] [bootfilename] <upload_size>
```

其中：

- loadAddress：DDR地址
- bootfilename：要下载或上传的文件名
- upload_size：可选项
  - 如果不指定，即表示下载文件
  - 如果指定了，即表示上传指定大小的DDR内容到服务端
    upload_size的单位是Byte。

说明

miniboot不支持网络，无法使用tftp命令。

3.3.6 Flash 分区烧录命令

为了便于说明在fastboot命令行烧录Flash分区，以表3-1Flash分区表为例：

表 3-1 Flash 分区表

分区名称	Flash偏移	长度	大小
Boot	0x00000000	0x00080000	512KB
bootargs	0x00080000	0x00100000	512KB
kernel	0x00100000	0x00400000	3MB
rootfs	0x00500000	0x06000000	91MB

说明

偏移是相对Flash首地址的偏移。Flash可以为SPI Flash/Nand/SPI Nand/eMMC，用户需要根据目标板的配置合理安排分区。

通常0x1000000以下的DDR空间预留给Boot使用，因此，我们选择0x1000000以后的空间用于镜像下载。

3.3.6.1 nand 命令

NAND Flash和SPI NAND使用nand命令烧录。

```
fastboot# help nand
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write[.yaffs] - addr off|partition size
    read/write 'size' bytes starting at offset 'off'
    to/from memory address 'addr', skipping bad blocks.
nand erase [clean] [off size] - erase 'size' bytes from
    offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
```

其中：

- nand read和nand write：分别是读nand分区和写Nand分区命令
- addr：要读和写的DDR缓存地址
- off：nand分区的起始位置
- size：要读或写的大小，off和size都是以Bytes为单位。
- nand write.yaffs：将yaffs文件系统写入Nand分区的命令，与nand write的区别是nand write.yaffs会将yaffs OOB数据写入到Nand的OOB区域。
- nand erase：擦除Nand分区的命令，可以指定偏移off和大小size擦除，如果不输入从参数，则整片擦除，但是此命令会跳过坏块。
- nand dump：可以打印指定偏移off处的Nand页面数据。

- nand bad：命令可以显示当前Nand器件中的坏块。
- nand scrub：命令会擦除整片Nand，包括坏块。

以下是用nand命令烧录表3-1 Flash分区的示例：

步骤1 烧录Boot：

将fastboot-burn.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000  fastboot-burn.bin
```

擦除flash上0x0~0x80000空间：

```
nand erase 0x0 0x80000
```

把DDR 0x1000000~0x180000的内容(fastboot-burn.bin)写入flash 0x0~0x80000的空间：

```
nand write 0x1000000 0x0 0x80000
```

步骤2 烧录bootargs：

将bootargs.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000  bootargs.bin
```

擦除flash上0x80000~0x100000空间：

```
nand erase 0x80000 0x80000
```

把DDR 0x1000000~0x180000的内容(bootargs.bin)写入flash 0x80000~0x100000的空间：

```
nand write 0x1000000 0x80000 0x80000
```

步骤3 烧录Kernel（Linux内核）：

将hi_kernel.bin下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 hi_kernel.bin
```

擦除flash上0x100000~0x500000空间：

```
nand erase 0x100000 0x400000
```

把DDR 0x1000000~0x1400000的内容(hi_kernel.bin)写入flash 0x100000~0x500000的空间：

```
nand write 0x1000000 0x100000 0x400000
```

步骤4 烧录rootfs(使用yaffs)：

将rootfs.yaffs文件下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 rootfs.yaffs
```

擦除flash上0x500000~0x6500000空间：

```
nand erase 0x500000 0x6000000
```

把DDR 0x1000000~0x7000000的内容(rootfs.yaffs)写入flash 0x500000~(0x500000+$(filesize))的空间：

```
nand write.yaffs 0x1000000 0x500000 $(filesize)
```

此处用的是write.yaffs命令，$(filesize)是tftp命令设置的rootfs.yaffs文件的大小。

说明

NAND Flash上也可以使用cramfs/squashfs/UBI文件系统，如果要烧录这三种文件系统，应该使用nand write命令。

----结束

3.3.6.2 sf 命令

SPI Flash使用sf命令烧录。

```
fastboot# help sf
sf probe [bus:]cs [hz] [mode]   - init flash device on given SPI bus
                                  and chip select
sf read addr offset len         - read `len' bytes starting at
                                  `offset' to memory at `addr'
sf write addr offset len        - write `len' bytes from memory
                                  at `addr' to flash at `offset'
sf erase offset len             - erase `len' bytes from `offset'
```

sf命令与nand命令类似，不同点在于SPI Flash烧录前需要执行一次sf probe 0，0表示SPI Flash设备号，通常海思只支持一片SPI Flash，因此设备号为0。

以下是使用sf命令烧录表3-1 Flash分区的示例：

步骤1 探测一下是否存在SPI Flash

```
sf probe 0
```

每次重启后，都需要执行此步骤才能继续后续的操作。

步骤2 烧录Boot：

将fastboot-burn.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000 fastboot-burn.bin
```

擦除flash上0x0~0x80000空间：

```
sf erase 0x0 0x80000
```

把DDR 0x1000000~0x180000的内容(fastboot-burn.bin)写入flash 0x0~0x80000的空间：

```
sf write 0x1000000 0x0 0x80000
```

步骤3 烧录bootargs：

将bootargs.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000  bootargs.bin
```

擦除flash上0x80000~0x100000空间：

```
sf erase 0x80000 0x80000
```

把DDR 0x1000000~0x180000的内容(bootargs.bin)写入flash 0x80000~0x100000的空间：

```
sf write 0x1000000 0x80000 0x80000
```

步骤4 烧录Kernel（Linux内核）：

将hi_kernel.bin下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 hi_kernel.bin
```

擦除flash上0x100000~0x500000空间：

```
sf erase 0x100000 0x400000
```

把DDR 0x1000000~0x1400000的内容(hi_kernel.bin)写入flash 0x100000~0x500000的空间：

```
sf write 0x1000000 0x100000 0x400000
```

步骤5 烧录rootfs(使用jffs2)：

将rootfs.jffs2文件下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 rootfs.jffs2
```

擦除flash上0x500000~0x6500000空间：

```
nand erase 0x500000 0x6000000
```

把DDR 0x1000000~0x7000000的内容(rootfs.jffs2)写入flash 0x500000~(0x500000+$(filesize))的空间：

```
sf write 0x1000000 0x500000 $(filesize)
```

$(filesize)是tftp命令设置的rootfs.jffs2文件的大小。

说明

SPI Flash上可以使用jffs2/cramfs/squashfs，cramfs和squashfs也是使用sf write命令烧录。

----结束

3.3.6.3 mmc 命令

eMMC Flash使用mmc命令烧录。

```
fastboot# help mmc
mmc read <device num> addr blk# cnt
mmc write <device num> addr blk# cnt
mmc erase <device num> blk# cnt
mmc write.ext4sp <device num> addr blk# cnt
mmc bootread <device num> addr blk# cnt
mmc bootwrite <device num> addr blk# cnt
mmc rescan <device num>
mmc list - lists available devices
mmc reg <device num>
```

说明

eMMC Flash读写操作的单位是块，块大小为512Bytes，与SPI Flash和Nand Flash不同，eMMC在写入之前不需要擦除。

其中：

- mmc read和mmc write：读写eMMC分区的命令。
- <device num>：eMMC设备号，通常海思芯片只支持一片eMMC器件，因此，设备号为0。
- addr：DDR缓存的地址。
- blk#和cnt：需要读或写的起始块号和块的数量，例如需要在1M~1.5M之间写入512K的数据，则blk#是0x800 (即1024*1024/512)，cnt是0x400（即512*1024/512）。
- mmc erase：eMMC擦除命令，虽然eMMC可以不用擦除就写，但是eMMC协议还是提供了擦除的命令，用于可靠地将器件上的数据清除，但依赖于eMMC器件的实现。

- mmc write.ext4sp：烧写ext4文件系统的命令。
- mmc bootread和mmc bootwrite：用于读写eMMC Boot分区。
- mmc rescan：可以用于重新扫描eMMC总线上的器件，通常在eMMC初始化失败时，用于尝试再初始化eMMC，便于定位问题。
- mmc list：用于查询可用的eMMC器件。
- mmc reg：用于查询eMMC器件内部的OCR/CID/CSD/RCA/EXT_CSD等寄存器。

注意

mmc erase命名能否将器件的数据彻底清除，依赖于器件本身，不同eMMC厂家对eMMC擦除命令的处理方式不同，因此需要与eMMC厂家确认。

以下是用mmc命令烧录表3-1 Flash分区的示例：

步骤1 烧录Boot：

将fastboot-burn.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000 fastboot-burn.bin
```

把DDR 0x1000000~0x180000的内容(fastboot-burn.bin)写入flash 0x0~0x80000的空间：

```
mmc write 0x1000000 0x0 0x400
```

步骤2 烧录bootargs：

将bootargs.bin下载到0x1000000开始的DDR缓存：

```
tftp  0x1000000  bootargs.bin
```

把DDR 0x1000000~0x180000的内容(bootargs.bin)写入flash 0x80000~0x100000的空间：

```
mmc write 0x1000000 0x400 0x400
```

步骤3 烧录Kernel（Linux内核）：

将hi_kernel.bin下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 hi_kernel.bin
```

把DDR 0x1000000~0x1400000的内容(hi_kernel.bin)写入flash 0x100000~0x500000的空间：

```
mmc write 0x1000000 0x800 0x2000
```

步骤4 烧录rootfs(使用ext4)：

将rootfs.ext4文件下载到0x1000000开始的DDR缓存：

```
tftp 0x1000000 rootfs.ext4
```

把DDR 0x1000000~0x7000000的内容(rootfs.ext4)写入flash 0x500000~(0x500000+0x6500000)的空间：

```
mmc write.ext4sp 0x1000000 0x2800 0x80000
```

注意

mmc write.ext4sp最后一个参数是根据分区大小计算出来的eMMC块数。

说明

eMMC Flash上可以使用ext4/cramfs/squashfs，cramfs和squashfs使用mmc write命令烧录。

----结束

3.4 支持多单板类型的 Boot 配置

多单板类型的Boot指在编译Boot时将指定的多个Boot表格同时编译到Boot镜像中，芯片启动时根据目标板的ADC电压选择使用其中的一个Boot表格启动，达到同一个Boot镜像支持多种类型的单板。

下面以Hi3798CV200平台为例，介绍多单板类型的Boot配置方法。

Hi3798CV200最多支持6种不同类型的单板，不同类型的单板通过ADC电压来区分。Hi3798CV200默认发布的单板类型与ADC电压的对应关系如表3-2所示。

表 3-2 Hi3798CV200 默认单板类型与 ADC 电压的对应关系

类型	Demo板名称	ADC电压值
0	Hi3798CV2DMB	3.3V
1	Hi3798CV2DMC	2.475V
2	Hi3798CV2DMD	1.925V
3	Hi3798CV2DME	1.375V
4	Hi3798CV2DMF	0.825V
5	TBD	0V

参考2.5 SDK配置文件，进入SDK配置的menuconfig菜单，在Board->Boot Regfile Config List中配置，如果在Board菜单下面找不到Boot Regfile Config List，就表示当前芯片不支持此功能或者已配置为高安芯片。

⚠注意

- 单板类型编号与ADC电压对应，不能随意更改；
- 高安芯片不支持此功能，因为高安芯片不允许在启动过程中有任何外部因素影响芯片的启动流程。

3.5 Miniboot usb 调试

3.5.1 miniboot usb 命令

- 扫描U盘命令。
  命令的格式：usb start
- 读取文件命令。
  命令的格式：fat read 文件名 ddr地址 读取长度
  例：

```
fat read secondboot.bin 0x1000000 0x100000
```

3.5.2 miniboot usb 配置项

选择进入miniboot usb配置项：

```
Location:
-->Boot System
-->[*]Usb Support
```

3.5.3 miniboot usb 调试步骤

miniboot usb调试步骤如下：

步骤1 配置miniboot usb配置项：

以hi3716mv410 nagra为例。

1. 首先需要拷贝配置文件到sdk根目录：
   ```
   cp configs/hi3716mv410/hi3716m41dma_hi3716mv410_nagra_debug_cfg.mak
   cfg.mak
   ```
2. 选择USB配置选项，根据“**3.5.1 miniboot usb命令**miniboot usb命令
   miniboot usb配置项”配置：
   ```
   Make menuconfig
   ```
3. 编译boot：
   ```
   Make hiboot
   make advca_programmer
   ```

步骤2 烧录boot：

使用hitool工具将boot烧录到单板，重启单板，按ctrl + c按键使单板停留在命令行。

步骤3 插入u盘，读取镜像：

1. 扫描u盘：
   ```
   usb start
   ```
2. 读取二级boot到ddr 0x1000000位置，启动二级boot：
   ```
   fat read secondboot.bin 0x1000000 0x100000
   go 0x1000000
   ```
3. 按ctrl + c 按键停留在二级boot的命令行
4. 读取文件系统镜像到指定的initmrd基地址，地址printenv查看：
   fat read *.squashfs addr size
5. 读取内核镜像到ddr, 启动内核：
   ```
   fat read hi_kernel.bin 0x1000000 0x800000
   bootm 0x1000000
   ```

步骤4 启动调试：

可以将步骤3中的命令配置到bootcmd环境变量中，自动运行，方便调试：

```
printenv
setenv bootcmd 'fat read *.squashfs  addr  0x100000;fat read
hi_kernel.bin 0x1000000 0x100000;bootm 0x1000000'
saveenv
```

⚠注意

高安版本只有debug配置文件支持miniboot usb调试，release配置文件不支持。

----结束

3.6 Fastboot 支持 USB 转网口

3.6.1 开启配置项

- 开启SDK下USB和NET配置项。

```
cp configs/hi3716mv430/hi3716m430dma_cfg.mak cfg.mak
make menuconfig

Location:
|-> Boot System
|  ->  Usb Support[*]

Location:
  |-> Board
  |  -> Ethernet Config
  |    -> Ethernet Support[*]
```

- 开启USB转网口配置项

  以Hi3716MV430芯片为例：

```
Source/boot/fastboot/include/configs/hi3716mv430.h
#undef CONFIG_USB_HOST_ETHER
```

  改为：

```
#define CONFIG_USB_HOST_ETHER
```

3.6.2 支持的 USB 转网口芯片型号

可以支持采用以下芯片的USB转网口：

- Asix 系列芯片
- Mcs7830芯片
- Smsc95xx系列芯片

4 Linux 内核

4.1 配置 Linux 内核

Linux内核配置文件的命名方式为：芯片名称_特性_defconfig，如：hi3798cv200_defconfig、hi3798cv200_loader_defconfig。

- 32位Linux内核配置文件目录：source/kernel/linux-4.4.y/arch/arm/configs
- 64位Linux内核配置文件目录：source/kernel/linux-4.4.y/arch/arm64/configs

说明

默认的SDK包没有包含内核源码包，请从内核社区下载内核源码！

以下说明修改Linux内核配置文件的方法，以Hi3798CV200平台32位Linux内核配置为例：

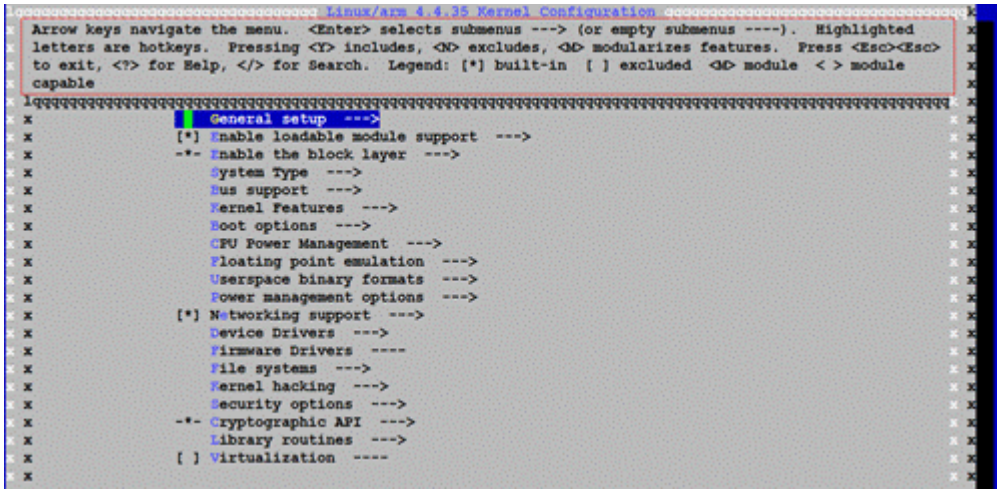
步骤1 进入内核源码目录：cd source/kernel/linux-4.4.y

步骤2 准备内核配置：make ARCH=arm hi3798cv200_defconfig

步骤3 启动内核的menuconfig：make ARCH=arm menuconfig

步骤4 进入内核的menuconfig界面，如图4-1所示。

图 4-1 Linux 内核 menuconfig 界面



menuconfig界面可以用上下键翻页，也可以用J和K键翻页，上方有menuconfig的使用帮助，如果要搜索某个配置项的位置，可以敲‘/’键，然后在弹出的对话框中输入需要搜索的关键字，menuconfig会提示所有匹配的选项。

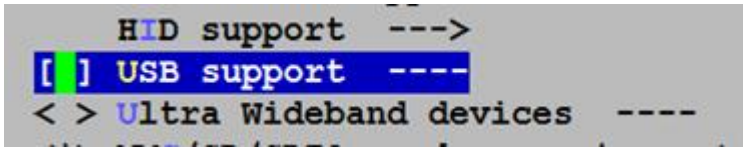
以关闭USB为例，敲”/”键后，输入“USB”，如图4-2所示。

图 4-2 在 Linux menuconfig 搜索 USB

图3 在Linux menuconfig搜索USB的结果显示了搜索的结果，其中Location显示了USB support选项的位置，按Enter键退出后，根据Location的提示，用上下左右找到对应的选项，敲入N就可以关闭USB support，如图4-4所示。

图 4-3 在 Linux menuconfig 搜索 USB 的结果

图 4-4 关闭 USB support 后



修改完配置项后，按Esc键退到主界面，再连续敲两次Esc键，会提示是否保存的界面，选择yes，然后退出menuconfig。。

步骤5 新修改的Linux内核配置保存在source/kernel/linux-4.4.y/.config文件，需要再将.config文件复制到Linux内核配置文件目录，以免丢失，执行：

```
cp .config arch/arm/configs/hi3798cv200_defconfig
```

步骤6 修改Linux内核配置文件后，需要清除编译内核配置文件时产生的临时文件，否则会导致后续编译失败。清除临时文件需执行：

```
make ARCH=arm distclean
```

----结束

说明

如果需要修改64位Linux内核配置文件，只需将上述命令中的arm替换为arm64即可。用menuconfig修改内核配置的优点是内核会自动开关关联的选项，修改完后的配置文件就是最终配置状态。通常不建议直接用文本编辑工具手动修改内核配置文件，如果习惯于手动修改，在改完后，需要操作一遍步骤 1、步骤 2和步骤 5才能生成最终的配置，不过，即使不做这些操作，内核Kconfig机制也会在编译内核时确保关联选项被打开或是关闭，只是比较最终生成的.config文件和内核源码目录的内核配置时，会发现一些差异。

注意

已提供默认的Linux内核配置文件，若无特殊情况不要修改。如果修改了内核配置文件，需要重新编译NTFS模块，否则可能导致不可预知的问题，详见NTFS模块编译方法。

仅在芯片支持64位时提供默认的64位Linux内核配置文件。

4.2 常用内核配置

常用内核配置的配置项说明如表4-1所示。

表 4-1 常用内核配置项说明

编号	配置项	说明
1	CONFIG_PRINTK CONFIG_TTY CONFIG_SERIAL_AMBA_PL011 CONFIG_SERIAL_AMBA_PL011_CONSOLE	串口打印配置项，如果需要打开串口打印，需要使能这几项配置。
2	CONFIG_NETDEVICES CONFIG_ETHERNET	如果仅仅需要关闭网络设备驱动，但需要保留协议栈的socket通信功能，可以仅关闭这些内核配置。
3	CONFIG_NET	如果要关闭网络协议栈，完全禁掉网络通讯功能，则除第2个配置外，还需要关闭此项配置。
4	CONFIG_MMC	如果要关闭eMMC或是SD卡驱动，需要关闭此项配置。
5	CONFIG_ATA	如果需要关闭SATA设备驱动，需要关闭此项配置。

4.3 编译 Linux 内核

在SDK配置文件中配置好Linux内核配置文件后，可以单独编译Linux内核。

单独编译Linux内核的命令：

```
cp configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak cfg.mak
make linux -j 2>&1 | tee linux.log
```

编译成功会生成Linux内核镜像文件和Linux内核驱动：

- Linux内核镜像文件为out/hi3798cv200/hi3798cv2dmo/image/hi_kernel.bin
- Linux内核驱动放在out/hi3798cv200/hi3798cv2dmo/kmod目录

4.4 Linux 内核支持 USB 转网口

4.4.1 开启内核配置项

- 开启USB配置项。

```
Symbol: USB_SUPPORT
[=y]


| Type  :
boolean



  | Prompt: USB
support


  |
Location:



  |     -> Device Drivers
```

- 开启USB转网口配置项

```
Symbol: USB_USBNET
[=y]



  | Type  :
tristate



  | Prompt: Multi-purpose USB Networking
Framework


  |
Location:



  |     -> Device
Drivers
```

```
  |        -> Network device support (NETDEVICES
[=y])


  |          -> USB Network Adapters (USB_NET_DRIVERS [=y])
```

4.4.2 支持的 usb 转网口芯片型号

内核下支持的USB转网口芯片型号如下，根据需要选择配置项。

图 4-5 USB 转网口芯片型号配置项

```
--- USB Network Adapters
< >   USB CATC NetMate-based Ethernet device support (NEW)
< >   USB KLSI KL5USB101-based ethernet device support (NEW)
< >   USB Pegasus/Pegasus-II based ethernet device support (NEW)
< >   USB RTL8150 based ethernet device support (NEW)
< >   Realtek RTL8152/RTL8153 Based USB Ethernet Adapters
< >   Microchip LAN78XX Based USB Ethernet Adapters (NEW)
<*>   Multi-purpose USB Networking Framework
<*>     ASIX AX88xxx Based USB 2.0 Ethernet Adapters
<*>     ASIX AX88179/178A USB 3.0/2.0 to Gigabit Ethernet (NEW)
-*-     CDC Ethernet support (smart devices such as cable modems)
< >     CDC EEM support (NEW)
<*>     CDC NCM support (NEW)
< >     Huawei NCM embedded AT channel support (NEW)
< >     CDC MBIM support (NEW)
< >     Davicom DM96xx based USB 10/100 ethernet devices (NEW)
< >     CoreChip-sz SR9700 based USB 1.1 10/100 ethernet devices (NEW)
< >     CoreChip-sz SR9800 based USB 2.0 10/100 ethernet devices
<*>     SMSC LAN75XX based USB 2.0 gigabit ethernet devices
< >     SMSC LAN95XX based USB 2.0 10/100 ethernet devices (NEW)
<*>     GeneSys GL620USB-A based cables
<*>     NetChip 1080 based cables (Laplink, ...)
<*>     Prolific PL-2301/2302/25A1 based cables
<*>     MosChip MCS7830 based Ethernet adapters
<*>     Host for RNDIS and ActiveSync devices
<*>     Simple USB Network Links (CDC Ethernet subset)
[*]       ALi M5632 based 'USB 2.0 Data Link' cables
[*]       AnchorChips 2720 based cables (Xircom PGUNET, ...)
[*]       eTEK based host-to-host cables (Advance, Belkin, ...)
[*]       Embedded ARM Linux links (iPaq, ...) (NEW)
[*]       Epson 2888 based firmware (DEVELOPMENT)
[*]       KT Technology KC2190 based cables (InstaNet)
<*>     Sharp Zaurus (stock ROMs) and compatible (NEW)
< >     Conexant CX82310 USB ethernet port (NEW)
< >     Samsung Kalmia based LTE USB modem (NEW)
< >     QMI WWAN driver for Qualcomm MSM based 3G and LTE modems (NEW)
< >   Option USB High Speed Mobile Devices (NEW)
< >   Intellon PLC based usb adapter
< >   Apple iPhone USB Ethernet driver (NEW)
<*>   USB-to-WWAN Driver for Sierra Wireless modems (NEW)
< >   LG VL600 modem dongle (NEW)
< >   QingHeng CH9200 USB ethernet support (NEW)
```

5 Linux 文件系统

5.1 文件系统简介

Linux目录结构的最顶层是一个被称为“/”的根目录。系统加载Linux内核之后，会挂载一个设备到根目录上。存在这个设备中的文件系统被称为根文件系统，即通常说的rootfs。所有的系统命令、系统配置以及其他文件系统的挂载点都位于这个根文件系统中。

根文件系统通常存放于内存和Flash中。根文件系统中存放了嵌入式系统使用的所有应用程序、库以及其他需要用到的服务。

嵌入式系统中常用文件系统包括有：cramfs、squashfs、JFFS2、yaffs2、UBIFS、ext4。它们的特点如下：

- cramfs和JFFS2具有好的空间特性，生成的镜像小，占用的Flash空间小，很适合小型嵌入式产品应用；
- cramfs和squashfs为只读压缩文件系统，squashfs相比于cramfs，能提供更大的压缩比，支持更大的镜像和文件；
- JFFS2为可读写的压缩文件系统，其挂载时间与Flash容量有关，越大的Flash，挂载时间越长，而且每次加载都需要将Flash上的所有节点(JFFS2的存储单位)加载到内存，因此内存消耗较大。由于JFFS2是按字节访问Flash，因此，只能用于SPI NOR Flash，不能用于NAND Flash；
- yaffs2文件系统是NAND Flash上使用较为成熟的文件系统，并且只能用于NAND Flash，相比于JFFS2，其内存消耗较小，但挂载时间仍然与Flash容量相关，越大的Flash，挂载时间越长；
- UBIFS一种用在大容量Flash上的可读写文件系统，对Flash的容量依赖较小，相比于yaffs2和JFFS2，其挂载时间和内存消耗都小很多，可以很好的适应GB以上大小的大容量Flash；
- ext4是Linux系统下的日志文件系统，适用于eMMC这类块设备存储器件。

5.2 cramfs

cramfs是针对Linux内核2.4之后的版本所设计的一种新型文件系统，使用简单，加载容易，速度快。

cramfs的优缺点如下：

- 优点：

  将文件数据以压缩形式存储，在需要运行时进行解压缩，能节省Flash存储空间。

- 缺点：

  由于它存储的文件是压缩的格式，所以文件系统不能直接在Flash上运行。同时，文件系统运行时需要解压数据并拷贝至内存中，在一定程度上降低读取效率。另外cramfs文件系统是只读的。

5.3 squashfs

squashfs文件系统也是一种压缩的只读文件系统。与cramfs相比，能提供更大的压缩比，支持更大的镜像和文件。

5.4 JFFS2

JFFS2是RedHat的David Woodhouse在JFFS基础上改进的文件系统，是用于微型嵌入式设备的原始闪存芯片的实际文件系统。JFFS2文件系统是日志结构化的可读写的文件系统。

JFFS2的优缺点如下：

- 优点：

  使用了压缩的文件格式。最重要的特性是可读写操作。

- 缺点：

  JFFS2文件系统挂载时需要扫描整个JFFS2文件系统，因此当JFFS2文件系统分区增大时，挂载时间也会相应的变长。使用JFFS2格式可能带来少量的Flash空间的浪费，这主要是由于日志文件的过度开销和用于回收系统的无用存储单元，浪费的空间大小大致是若干个数据段。

  JFFS2的另一缺点是当文件系统已满或接近满时，JFFS2运行速度会迅速降低。这是因为垃圾收集的问题。

加载JFFS2文件系统时的步骤如下：

步骤1 扫描整个芯片，对日志节点进行校验，并且将日志节点全部装入内存缓存。

步骤2 对所有日志节点进行整理，抽取有效的节点并整理出文件目录信息。

步骤3 找出文件系统中无效节点并且将它们删除。

步骤4 最后整理内存中的信息，将加载到缓存中的无效节点释放。

----结束

由此可以看出，虽然这样能有效地提高系统的可靠性，但是在一定程度上降低了系统的速度。尤其对于较大的闪存芯片，加载过程会更慢。

5.5 yaffs2

yaffs2是专门为NAND、SPI NAND设计的嵌入式文件系统。它是日志结构的文件系统，提供了损耗平衡和掉电保护，可以有效地避免意外掉电对文件系统一致性和完整性的影响。

yaffs2的优缺点如下：

- 优点：
  - 专门针对NAND、SPI NAND，软件结构得到优化，速度快。
  - 使用硬件的spare area区域存储文件组织信息，启动时只需扫描组织信息，启动比较快。
  - 采用多策略垃圾回收算法，能够提高垃圾回收的效率和公平性，达到损耗平衡的目的。
- 缺点：

  没有采用压缩的文件格式。如果用同等大小的文件夹制作根文件系统，yaffs2镜像文件要比JFFS2镜像文件大。

5.6 UBIFS

UBIFS (Unsorted Block Image File System)是用于固态硬盘存储设备上的JFFS2后继文件系统之一。是专门为了解决MTD（Memory Technology Device）设备所遇到瓶颈的文件系统。

由于NAND Flash容量的暴涨，YAFFS等皆无法再控制NAND Flash的空间。UBIFS在设计与性能上比YAFFS2、JFFS2更能适用于MLC NAND Flash。

UBIFS的优缺点如下：

- 优点：
  - UBIFS支持write-back，其写入的数据会被cache缓存，直到有必要写入时才写到Flash，大大地降低分散小区块数量及I/O使用率。
  - UBIFS文件系统目录存储在Flash上，UBIFS mount时不需要scan整个Flash的数据来重新创建文件目录。
  - 支持on-the-flight压缩文件数据，而且可选择性压缩部分文件。另外UBIFS使用日志（journal），可减少对flash index的更新频率。
- 缺点：

  与JFFS2一样，UBIFS建构于MTD device之上，而且与一般的block device不兼容。

挂载 UBIFS文件系统时的步骤如下：

步骤1 烧录UBIFS镜像到对应的NAND分区。

步骤2 启动系统，绑定UBI到对应的MTD分区。

说明

17代表的是第17个分区，需要与对应的分区对齐。

```
# ubiattach /dev/ubi_ctrl -m 17
ubi0: attaching mtd17
ubi0: scanning is finished
ubi0: attached mtd17 (name "data", size 32 MiB)
ubi0: PEB size: 131072 bytes (128 KiB), LEB size: 126976 bytes
ubi0: min./max. I/O unit sizes: 2048/2048, sub-page size 2048
ubi0: VID header offset: 2048 (aligned 2048), data offset: 4096
ubi0: good PEBs: 256, bad PEBs: 0, corrupted PEBs: 0
ubi0: user volume: 1, internal volumes: 1, max. volumes count: 128
ubi0: max/mean erase counter: 3/1, WL threshold: 128, image sequence
number: 1382560713
ubi0: available PEBs: 0, total reserved PEBs: 256, PEBs reserved for bad
PEB handling: 20
ubi0: background thread "ubi_bgt0d" started, PID 1546
```

步骤3 挂载UBIFS文件系统。

```
# mount -t ubifs /dev/ubi0_0 /mnt/
UBIFS (ubi0:0): background thread "ubifs_bgt0_0" started, PID 1549
UBIFS (ubi0:0): UBIFS: mounted UBI device 0, volume 0, name "ubifs"
UBIFS (ubi0:0): LEB size: 126976 bytes (124 KiB), min./max. I/O unit
sizes: 2048 bytes/2048 bytes
UBIFS (ubi0:0): FS size: 28188672 bytes (26 MiB, 222 LEBs), journal size
1396736 bytes (1 MiB, 11 LEBs)
UBIFS (ubi0:0): reserved for root: 1331420 bytes (1300 KiB)
UBIFS (ubi0:0): media format: w4/r0 (latest is w4/r0), UUID A2AB1D24-
BC63-469F-8BC4-E2067A9484B5, small LPT model
```

步骤4 查看分区信息。

```
# df -h
Filesystem                Size      Used Available Use% Mounted on
/dev/root                 8.6M      8.6M         0 100% /
dev                     115.7M     12.0K    115.7M   0% /dev
tmp                     115.7M         0    115.7M   0% /tmp
ubi0_0                   24.2M     28.0K     22.9M   0% /mnt
```

----结束

5.7 initmrd

initmrd是内存文件系统，可以支持挂载多个内存文件系统。

5.7.1 initmrd 参数

```
initmrd=index, addr, size
```

参数意义如下：

- addr：文件系统在DDR中的加载位置。
- size：DDR中预留给文件系统的空间（该值必须大于文件系统Image 的大小）。
- index：用于支持多个内存文件系统挂载，例如：
  - index =1表示第一个内存文件系统分区信息。
  - index =x 表示第x个内存文件系统分区信息。
  - index最小值为1，SDK默认最大支持挂载16个内存文件系统。

addr与size的值推荐使用Bootargs中的默认值，如果需要修改请调整内存排布。

5.7.2 initmrd 文件系统加载步骤

步骤1 制作文件系统镜像，烧录到对应的分区。

步骤2 在bootargs中增加分区initmrd (**initmrd=index, addr, size**)的配置.

说明

该示例挂载了两个内存文件系统，不包括启动的内存文件系统。

```
bootargs=mem=512M mmz=ddr,0,0,256M console=ttyAMA0,115200
initrd=0x4000000,0x2000000 initmrd=1,0x6000000,0x40000
initmrd=2,0x6400000,0x400000 root=/dev/ram rootfstype=squashfs
mtdparts=hinand:1M(boot),1M(secondboot),1M(bootargs),1M(baseparam),
1M(pqparam),1M(logo),1M(deviceinfo),1M(softwareinfo),1M(loaderdb),
1M(loaderdbbak),8M(loader),8M(loaderbak),8M(trustedcore),8M(kernel),
20M(rootfs), 4M(app),4M(calib),32M(data),-(others)
```

步骤3 在bootcmd中把分区内容读到对应的内存配置区域。

```
bootcmd=nand read 0x4000000 0x2A00000 0x1400000;nand read 0x6000000
0x3E00000 0x400000;nand read 0x6400000 0x4200000 0x400000;nand read
0x1FFFFC0 0x2200000 0x800000;bootm 0x1FFFFC0
```

步骤4 启动系统，手动挂载RAM文件系统分区。预先创建挂载节点/mnt/app和/mnt/calib。

```
mount /dev/ram1 /mnt/app
mount /dev/ram2 /mnt/calib
```

----结束

5.8 构建 rootfs 文件系统

5.8.1 配置

5.8.1.1 基本配置

SDK menuconfig的Linux(REE) System -> Filesystem包含了rootfs的基本配置，其中包含：

- Busybox Config：rootfs基于busybox构建，此处可以配置需要使用的busybox配置文件，关于busybox的配置，见第**6 Busybox**章。
- [ ]C++ Runtime Libarary Support：是否需要包含C++库，对于小容量Flash的产品，可以考虑裁剪掉C++库，使用纯C代码实现应用，以节省Flash空间。
- [ ]Enable Strip：是否strip二进制程序和动态库，strip可以节省Flash空间，但会导致不能用gdb调试程序。
- [] Install Share Libs to Rootfs：是否安装动态库，如果只需要静态可执行程序，不需要动态库，可以关闭此项，以节省Flash空间。

5.8.1.2 权限配置

如果需要定制各个目录和文件的权限，例如，在高安配置中，希望使用多用户，不同的home目录配置不同的用户和组权限，则需要使用到AttrRule文件来配置，AttrRule文件存放在以下目录，以txt文件的格式存放：

```
source/rootfs/scripts/attr_rule
```

例如，非高安的SDK配置使了rulelist_noca.txt文件。

如果需要查询当前SDK配置对应使用哪个AttrRule文件，可以用以下命令查询：

```
make fs AT=
```

-T后面的文件就是当前使用的AttrRule文件，以Hi3798CV200为例：

```
cp configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak cfg.mak
make fs AT=
```

打印：

```
-T /home/sdk/Code/source/rootfs/scripts/attr_rule/rulelist_noca.txt
```

AttrRule的每一行是一个规则，每条规则定义了某个目录或文件的权限配置，规则的格式为：

```
<文件权限>  <所有者ID:组ID>  <文件能力>  <”目录名或文件名”>
```

- <文件权限>：八进制格式的文件权限描述，遵循标准的linux权限描述格式，以0开头，后面跟三位数字，分别表示文件所有者的权限、组内用户的权限和其它用户的权限，例如，0550表示文件的所有者和组内用户对该文件有可读可执行权限，但不可写，其它所有用户都不可访问此文件。
- <所有者ID:组ID>：代表文件或目录的所有者ID和所属的组ID，以冒号分隔
- <文件能力>：文件的能力项，即Linux capabilities，可以通过此配置给文件赋予一些只有root用户才具备的权限，详细请查看capabilities的man手册（man capabilities）。
- <”目录名或文件名”>：指定此规则适用的文件或目录，如果指定目录，则此目录下所有文件都会配置此规则指定的权限。

  示例：
- 0550 1002:1002 "sys_admin" "/home/lxc1/home/lxc_mount"，表示/home/lxc1/home/lxc_mount文件可以被ID为1002的用户读取和执行，但不能修改，也可以被属于ID 1002用户组的用户读取和执行，但不能修改，其它用户不能访问此文件，同时该文件具有sys_admin能力，可以执行挂载U盘等系统操作，详见man capabilities。
- 0550 0:1000 "" "/bin",表示/bin目录下所有文件都配置成只有root用户和属于1000组的用户可以读和执行，但不能写，其它用户都不能访问。

  AttrRule文件在编译文件系统时生效，制作文件系统的工具会解析此文件，并将每个规则应用到对应的目录或文件。修改完AttrRule文件后，重编文件系统就可以，单独编译文件系统的方法见**5.8.2 编译**节。

5.8.2 编译

SDK提供了默认的参考rootfs包，用于构建Linux的根文件系统，放在：

- source/rootfs/scripts/rootfs.tar.bz2，用于构建大系统的rootfs
- source/rootfs/scripts/rootfs_loader.tar.bz2，用于构建AppLoader的rootfs

  编译rootfs时会将此包解压并拷贝到out目录下的rootbox目录，作为基础的rootfs目录结构，然后追加busybox、udev、C/C++库等组件，构成一个完整的rootfs。

  单独编译rootfs的方法，执行：

```
make -j rootbox
make fs
```

  如果是高安的配置，需要再执行：

```
make signature
```

6 Busybox

6.1 简介

Busybox是一个在Linux上可执行二进制程序，提供基本的Linux命令集的，包括ls、cat、find、grep等，所有命令都是以链接的形式链接到/bin/busybox，busybox程序会根据执行的命令名字来区分需要执行功能，同时Busybox还包含了一个shell。由于其体积小，且支持裁剪和交叉编译，适用于嵌入式系统。

SDK的rootfs文件系统基于Busybox制作。

- Busybox的源码位置：
  `third_party/open_source/busybox-X_XX_X.tar.bz2`
- Busybox配置文件的位置：
  `source/rootfs/busybox/busybox-X_XX_X.config`

说明

X_XX_X代表Busybox的版本号

6.2 配置 Busybox

以下以默认的glibc.config配置和busybox-1_26_2版本为例说明如何修改Busybox配置。

步骤1 进入存放busybox源码的目录：cd third_party/open_source

步骤2 解压busybox源码包：tar -xf busybox-1_26_2.tar.bz2

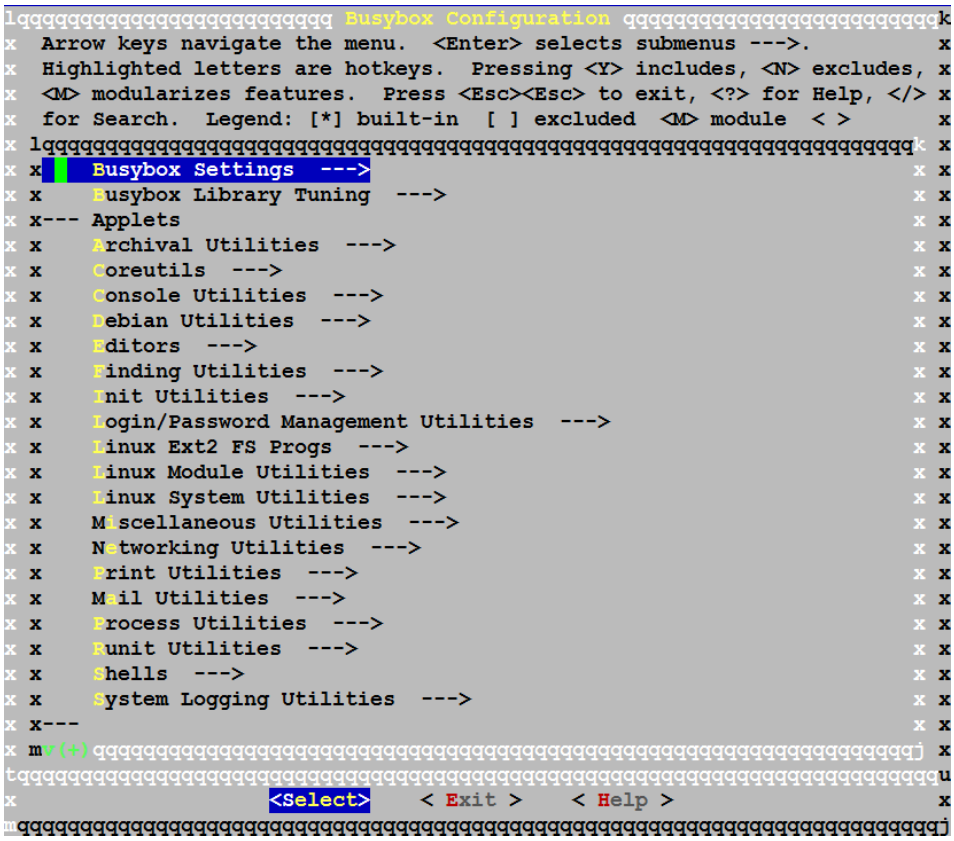
步骤3 回到SDK根目录: cd -

步骤4 拷贝busybox配置到busybox源码目录：cp source/rootfs/busybox/busybox-1_26_2.config/glibc.config third_party/open_source/busybox-1_26_2/.config，注意，需要重命名为.config

步骤5 进入busybox源码目录：cd third_party/open_source/busybox-1_26_2

步骤6 输入make menuconfig，进入busybox配置界面，如图6-1所示

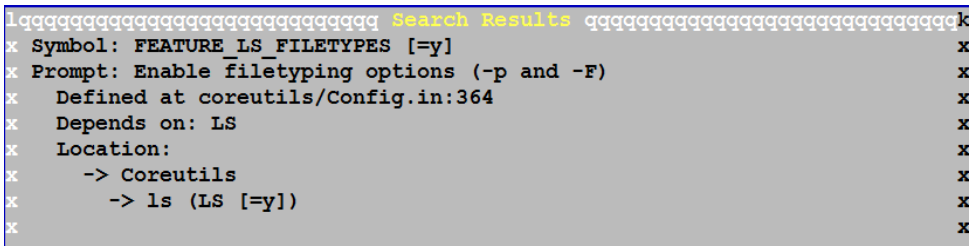
图 6-1 Busybox 配置界面



可以用上下键翻页，也可以用J和K键翻页，上方有menuconfig的使用帮助，如果要搜索某个配置项的位置，可以敲‘/’键，然后在弹出的对话框中输入需要搜索的关键字，menuconfig会提示所有匹配的选项。

以关闭ls命令为例，敲”/”键后，输入“LS”，如图6-2所示。

图 6-2 搜索 ls 命令的配置



其中Location显示了ls命令选项的位置，即位于Coreutils->ls，按Enter键退出后，根据Location的提示，用上下左右找到对应的选项，敲入N就可以关闭，如图6-3所示。

图 6-3 关闭 ls 命令

需改完配置项后，按Esc键退到主界面，再连续敲两次Esc键，会提示是否保存的界面，选择yes，然后退出menuconfig。

步骤7 修改后的Busybox配置保存在third_party/open_source/busybox-1_26_2/.config，需要拷贝回Busybox配置的存放路径，以免丢失：

```
cp third_party/open_source/busybox-1_26_2/.config source/rootfs/busybox/
busybox-1_26_2.config/glibc.config
```

----结束

6.3 编译 Busybox

Busybox是随rootfs一起编译，编译后在rootbox目录下生成bin/busybox文件，在SDK menuconfig的Linux(REE) System-> Filesystem-> Busybox Config位置可以配置当前用的Busybox配置文件，配置好后，需重新编译文件系统：

```
cp configs/hi3798cv200/hi3798cv2dmo_hi3798cv200_cfg.mak cfg.mak
make rootbox -j 2>&1 | tee rootbox.log
make fs
```

如果是高安的SDK配置，还需要签名：

```
make signature
```

编译后会生成rootfs镜像：

out/hi3798cv200/hi3798cv2dmo/image/emmc_image/rootfs.squashfs

注意，此处以squashfs为例，其它文件系统格式，生成的镜像有不同的扩展名，如果是高安的SDK配置，通常扩展名还会有.sig，详见各高安开发指南。

7 应用程序开发简介

7.1 编译选项

7.1.1 使用 VFP

使用下面的编译选项，可以生成硬浮点指令，通过FPU（Floating Point Unit）模块完成浮点运算。

```
-mfloat-abi=softfp -mfpu=vfpv3-d16
```

示例：

```
CFLAGS+=-march=armv7-a -mcpu=cortex-a9 -mfloat-abi=softfp -mfpu=vfpv3-d16
```

说明

推荐使用VFP（Vector Floating-point），兼容性好。64位工具链aarch64-histbv100-linux-默认打开了浮点指令，不能指定VFP选项。

7.1.2 使用 NEON

使用下面的编译选项，可以生成NEON指令，通过NEON模块完成浮点运算。

```
-mfloat-abi=softfp -mfpu=neon
```

示例：

```
CFLAGS+=-march=armv7-a -mcpu=cortex-a9 -mfloat-abi=softfp -mfpu=neon
```

说明

只有在CPU带NEON模块，且内核支持NEON的情况下，才可以使用NEON编译选项，否则会报指令异常错误。

- 如何查看内核是否支持NEON

  在单板上使用命令：

  ```
  cat /proc/cpuinfo
  ```

  如果其中的“Features”行含有“neon”，则表示支持NEON。

  ```
  Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls
  ```

- 如何查看镜像是否使用了NEON

  在linux服务器上使用命令：

  ```
  arm-histbv310-linux-readelf -A a.out
  ```

如果其中含有“Tag_Advanced_SIMD_arch: NEONv1”，则表示使用了NEON。

```
Tag_FP_arch: VFPv3
Tag_Advanced_SIMD_arch: NEONv1
Tag_ABI_PCS_wchar_t: 4
```

说明

64位工具链aarch64-histbv100-linux-默认打开了浮点指令，不能指定NEON选项。

7.2 使用 gdbserver 调试应用程序

在很多情况下，需要对应用程序进行调试。在Linux下调试程序，常用的工具是gdb。由于嵌入式单板的资源有限，一般不直接在目标机上运行gdb进行调试，而是采取gdb+gdbserver的方式。gdbserver在目标机中运行，gdb则在宿主机上运行。根文件系统中已经包含gdbserver。使用gdbserver调试应用程序的步骤如下：

步骤1 启动Linux并登陆进入shell。

如要进行gdb调试，首先要启动gdb server。方法是先进入需要调试的程序所在目录，如：被调试的程序文件名是**hello**，则输入命令：

```
gdbserver :2000 hello &
```

上述命令表示在目标机的2000端口开启了一个调试进程，**hello**就是要调试的程序。

步骤2 在Linux服务器上启动gdb程序，因为目标机为ARM，所以启动arm-xxxx-gdb(以发布版本真实名称为准)。

步骤3 启动arm-xxx-gdb后，在命令提示符状态下输入命令，与目标机进行连接。

```
(gdb) target remote 192.168.0.5:2000  /*192.168.0.5为单板IP*/
```

注意

端口号和目标机开启的端口号要一致。

步骤4 连接成功后，会输出提示信息，如下所示：

```
remote debugging using 10.70.153.100:2000
0x40000a70 in ?? ()
```

步骤5 进行符号文件加载：

```
(gdb) add-symbol-file hello 40000a70
```

或者使用：

```
(gdb) file hello
```

步骤6 输入各种gdb命令如list、run、next、step、break 即可进行程序调试。

----结束