



# CV180X & CV181X SDK 编译使用手册

Version: 1.0

Release date: 2022-10-28

©2022 北京晶视智能科技有限公司  
本文件所含信息归北京晶视智能科技有限公司所有。  
未经授权，严禁全部或部分复制或披露该等信息。

# 目录

<b>1</b>	<b>声明</b>	<b>2</b>
<b>2</b>	<b>建构 CVITEK 软件编译环境</b>	<b>3</b>
2.1	Linux 服务器	3
2.1.1	于 VirtualBoxVM 安装 Ubuntu	3
2.1.2	Ubuntu 开机设定	5
2.1.3	安装 SSH Server	8
2.1.4	安装 Samba Server	9
2.2	建构编译环境	9
2.2.1	使用 Docker 编译	10
2.3	配置 github 账号	12
2.4	获取 SDK 的方式	12
2.5	编译	13
2.5.1	环境变量说明	13
2.5.2	编译整个软件包	13
2.5.2.1	透过 defconfig 设定	13
2.5.2.2	透过 Menuconfig 设定	14
2.5.3	编译完整 SDK 文档	15
2.5.4	编译部份 SDK 文档	16
2.5.4.1	单独编译 Uboot	16
2.5.4.2	单独编译 kernel	17
2.5.4.3	单独编译 middleware	18
2.6	磁盘分区	19
2.6.1	磁盘分区修改	19
2.7	内存映射	20
2.7.1	内存映射修改	22
<b>3</b>	<b>烧录说明</b>	<b>23</b>
3.1	使用前准备	23
3.2	操作过程	23
3.3	操作实例	23
3.4	打包烧录器烧录镜像	24
3.5	注意事项	25
<b>4</b>	<b>EVB 接口说明</b>	<b>26</b>
<b>5</b>	<b>根文件系统 (rootfs)</b>	<b>28</b>
5.1	根文件系统简介	28
5.2	Rootfs	29
5.2.1	Pre-build rootfs 架构	29
5.2.2	编译来自 buildroot 的 rootfs	31

5.2.3	将 rootfs 包装成可烧录映像档 . . . . .	34
5.2.4	Linux kernel 自动加载 rootfs . . . . .	35
<b>6</b>	<b>使用 NFS 加速开发</b>	<b>36</b>
6.1	Ubuntu Server 端设置说明: . . . . .	36
6.2	EVB 板端 mount 说明: . . . . .	36
6.3	注意事项: . . . . .	37

**修订记录**

Revision	Date	Description
0.0.0.1	2020/03/08	Initial.
0.0.0.2	2022/01/28	Update Document Chapter
0.0.0.3	2022/02/17	Update root file system related information
1.0	2022/10/28	Revise for CV180X/CV181X processor.



# 1 声明



## 法律声明

本数据手册包含北京晶视智能科技有限公司（下称“晶视智能”）的保密信息。未经授权，禁止使用或披露本数据手册中包含的信息。如您未经授权披露全部或部分保密信息，导致晶视智能遭受任何损失或损害，您应对因之产生的损失/损害承担责任。

本文件内信息如有更改，恕不另行通知。晶视智能不对使用或依赖本文件所含信息承担任何责任。本数据手册和本文件所含的所有信息均按“原样”提供，无任何明示、暗示、法定或其他形式的保证。晶视智能特别声明未做任何适销性、非侵权性和特定用途适用性的默示保证，亦对本数据手册所使用、包含或提供的任何第三方的软件不提供任何保证；用户同意仅向该第三方寻求与此相关的任何保证索赔。此外，晶视智能亦不对任何其根据用户规格或符合特定标准或公开讨论而制作的可交付成果承担责任。

## 联系我们

**地址** 北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

深圳市宝安区福海街道展城社区会展湾云岸广场 T10 栋

**电话** +86-10-57590723 +86-10-57590724

**邮编** 100094（北京）518100（深圳）

**官方网站** <https://www.sophgo.com/>

**技术论坛** <https://developer.sophgo.com/forum/index.html>

# 2 建构 CVITEK 软件编译环境

---

## 2.1 Linux 服务器

开发者可选择使用：

- Ubuntu OS 计算机
- Windows OS 计算机 + Virtualbox VM (上面运行 Ubuntu)

两种方式，都请安装成 Ubuntu 20.04 LTS 版本。

Virtualbox VM 下载网址: <https://www.virtualbox.org/wiki/Downloads>

Ubuntu 20.04 LTS 下载网址: <https://releases.ubuntu.com/focal/ubuntu-20.04.6-desktop-amd64.iso>

### 2.1.1 于 VirtualBoxVM 安装 Ubuntu

- 建立新的 VM，并加以命名

? X


← 建立虛擬機器

### 名稱和作業系統

請為新的虛擬機器選擇描述性名稱和目的地資料夾，並選取要在其上安裝的作業系統類型。您選擇的名稱將在整個 VirtualBox 中使用，以標識這部電腦。

名稱:

機器資料夾:

類型(T):  

版本(V):

專家模式(E)

下一個(N)

取消

- 规划 8GB 记忆体供 VM 使用。

? X

← 建立虛擬機器

### 記憶體大小

選取配置到虛擬機器的記憶體量 (RAM)，單位 MB。

建議的記憶體大小為 **1024MB**。

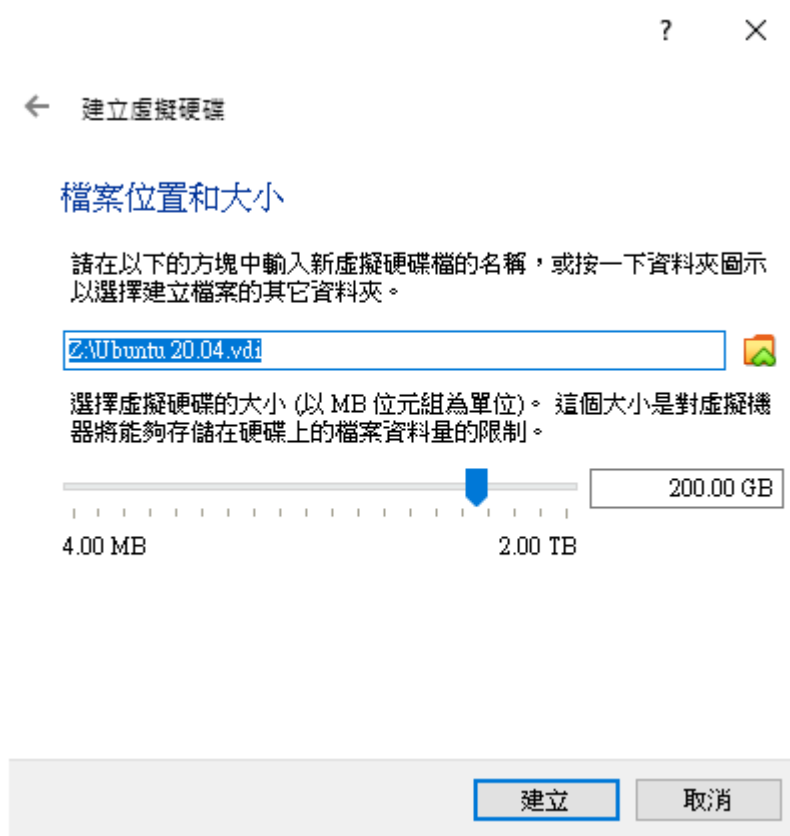
MB

4 MB 16384 MB

下一個(N)

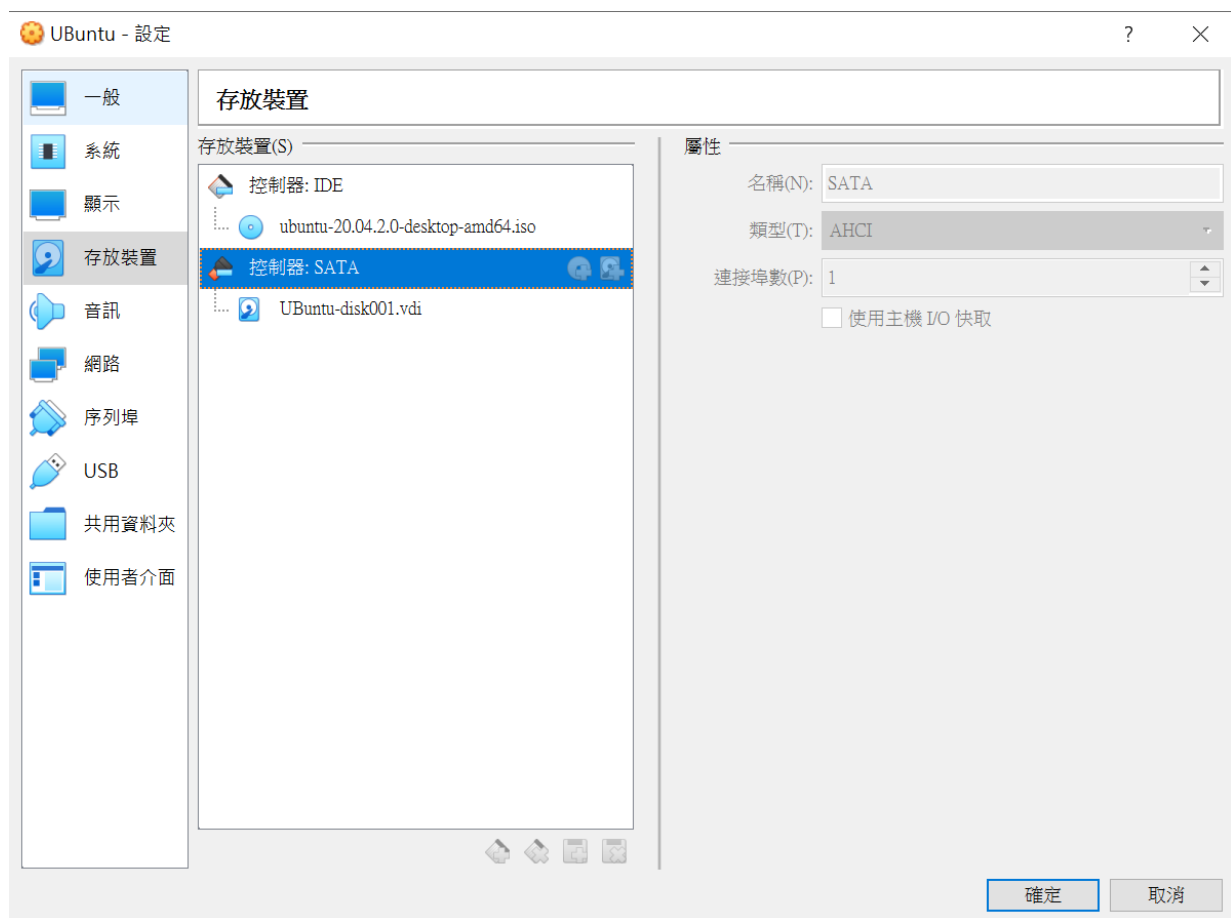
取消

- 预留 200GB 硬盘空间，供后续存放 SDK 用。



## 2.1.2 Ubuntu 开机设定

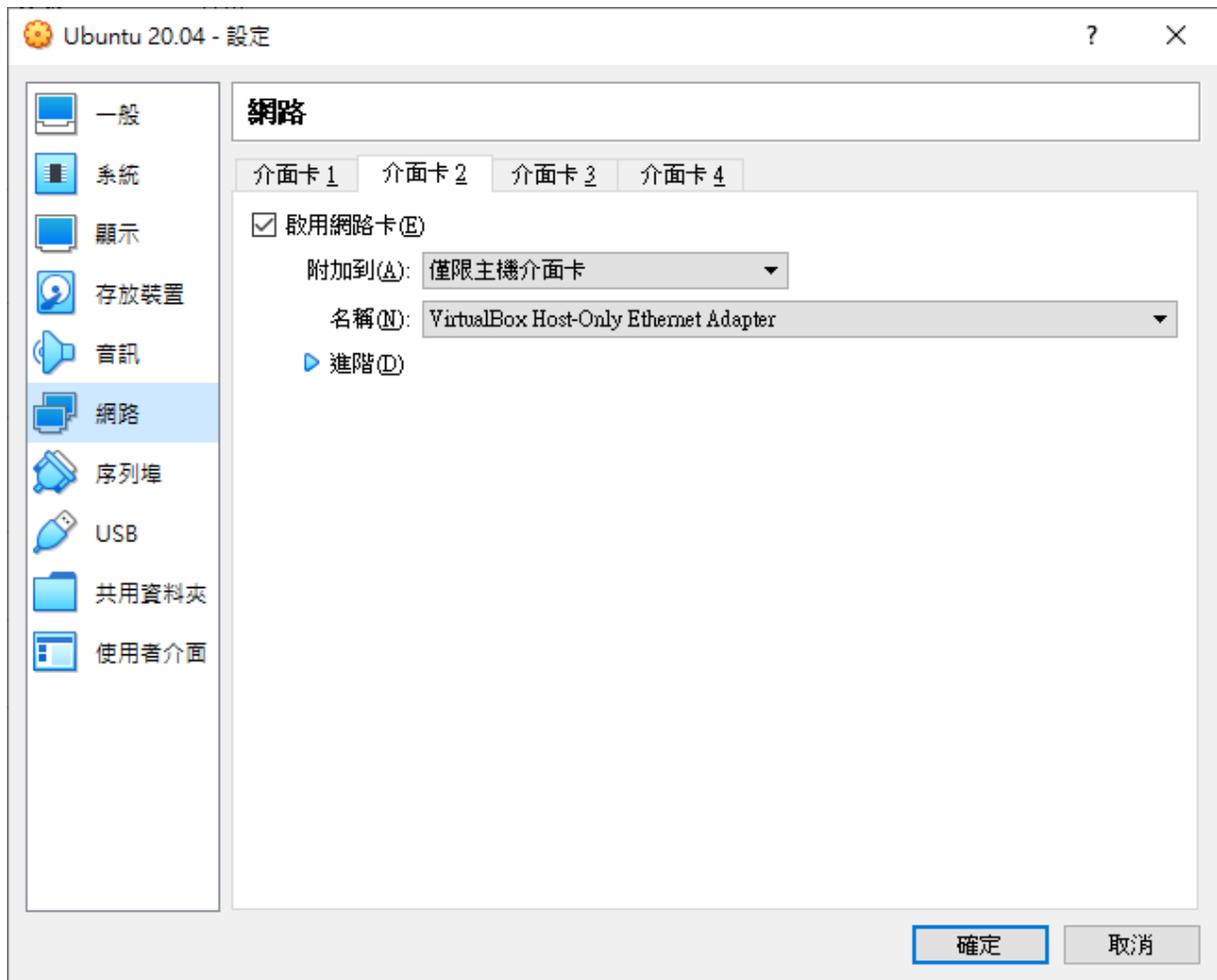
- 第一次开机需要挂载安装光盘 ISO 档案



· 开始安装



- 设定 VirtualBox Host-only Ethernet Adapter 以便 Host 与 VirtualBox 沟通（终端服务以及档案分享）



### 2.1.3 安裝 SSH Server

SSH Server 安裝

```
sudo apt-get install ssh
sudo apt-get install openssh-server
```

安裝後可以修改一些 ssh 的設定, 如 port、密碼認證、root 登入等

```
vim /etc/ssh/sshd_config

Port 22

PasswordAuthentication yes

PermitRootLogin yes -> 是否開放 root 登入
```

修改後要重啟 SSH

```
sudo /etc/init.d/ssh restart
```

## 2.1.4 安装 Samba Server

Ubuntu VB 需要安装 Samba 套件，方便后续 Host PC 与其做档案分享。

安装 Samba 前，先用 ifconfig 获取 IP 地址，第一次安装会发现没有 net-tool 支持，需要安装 net-tool

```
sudo apt install net-tools
sudo apt-get install samba samba-common
```

建立账号的 samba 密码

```
sudo smbpasswd -a cvitek
```

修改/etc/samba/smb.conf，增加以下内容

```
[cvitek]
path = /home/cvitek
writable = yes
browseable = yes
valid users = cvitek
```

启动 samba server

```
sudo service smbd restart
```

WINDOW PC 端连接 Samba server (<Server IP>)

▼ 網路位置 (1)



参考2.2. 安装 CVITEK Build Environment 即可进行编译。

## 2.2 建构编译环境

在编译 SDK 之前，Ubuntu 需要安装以下套件：

```
sudo apt-get update
sudo apt-get install -y build-essential
sudo apt-get install -y ninja-build
sudo apt-get install -y automake
sudo apt-get install -y autoconf
sudo apt-get install -y libtool
sudo apt-get install -y wget
sudo apt-get install -y curl
sudo apt-get install -y git
sudo apt-get install -y gcc
sudo apt-get install -y libssl-dev
sudo apt-get install -y bc
sudo apt-get install -y slib
```

(下页继续)



(续上页)

```
sudo apt-get install -y squashfs-tools
sudo apt-get install -y android-sdk-libparse-utils
sudo apt-get install -y android-sdk-ext4-utils
sudo apt-get install -y jq
sudo apt-get install -y cmake
sudo apt-get install -y python3-distutils
sudo apt-get install -y tcsh
sudo apt-get install -y scons
sudo apt-get install -y parallel
sudo apt-get install -y ssh-client
sudo apt-get install -y tree
sudo apt-get install -y python3-dev
sudo apt-get install -y python3-pip
sudo apt-get install -y device-tree-compiler
sudo apt-get install -y libssl-dev
sudo apt-get install -y ssh
sudo apt-get install -y cpio
sudo apt-get install -y squashfs-tools
sudo apt-get install -y fakeroot
sudo apt-get install -y libncurses5
sudo apt-get install -y flex
sudo apt-get install -y bison
sudo apt-get install -y pkg-config
sudo pip3 install -U yoctoools
```

**注意：** 注意检查 python 命令是否存在，如果不存在，需要创建软连接到 python3 命令。

```
sudo ln -s /usr/bin/python3 /usr/bin/python
```

## 2.2.1 使用 Docker 编译

可以将 SDK 映射到 docker 容器中，在容器内运行编译命令，如果你的编译环境准备有问题，可以使用此方式。

1. 准备 Dockerfile，将下面的内容保存到文件 cvitek-linux-Dockerfile

```
# 基础镜像使用 ubuntu:20.04
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive
ENV TZ=Asia/Shanghai

# 安装依赖
RUN apt-get update \
    && apt-get install -y \
    pkg-config

RUN DEBIAN_FRONTEND=noninteractive apt-get install -y \
    build-essential \
    ninja-build \
```

(下页继续)

(续上页)

```

automake \
autoconf \
libtool \
wget \
curl \
git \
gcc \
libssl-dev \
bc \
slib \
squashfs-tools \
android-sdk-libparse-utils \
android-sdk-ext4-utils \
jq \
cmake \
python3-distutils \
tclsh \
scons \
parallel \
ssh-client \
tree \
python3-dev \
python3-pip \
device-tree-compiler \
libssl-dev \
ssh \
cpio \
squashfs-tools \
fakeroot \
libncurses5 \
flex \
bison \
rsync \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

# 按照 yoctools, 编译 alios 需要
RUN ln -s /usr/bin/python3 /usr/bin/python
RUN pip3 install yoctools -U

# 设置工作目录
WORKDIR /cvitek-sdk

# 设置挂载点
VOLUME ["/cvitek-sdk"]

CMD ["/bin/bash"]

```

## 2. 编译生成 docker 镜像

```
docker build -t cvitek-linux -f cvitek-linux-Dockerfile .
```

## 3. 获取 SDK 到文件夹 /data/xxx/SDK/ 后（见下面的描述），启动容器：

```
docker run -it --name cvitek-linux \  
-v /data/xxx/SDK:/cvitek-sdk \  
cvitek-linux
```

## 2.3 配置 github 账号

在 github 建立个人账号，并配置好 ssh key，下载代码需要用到个人 github 账号

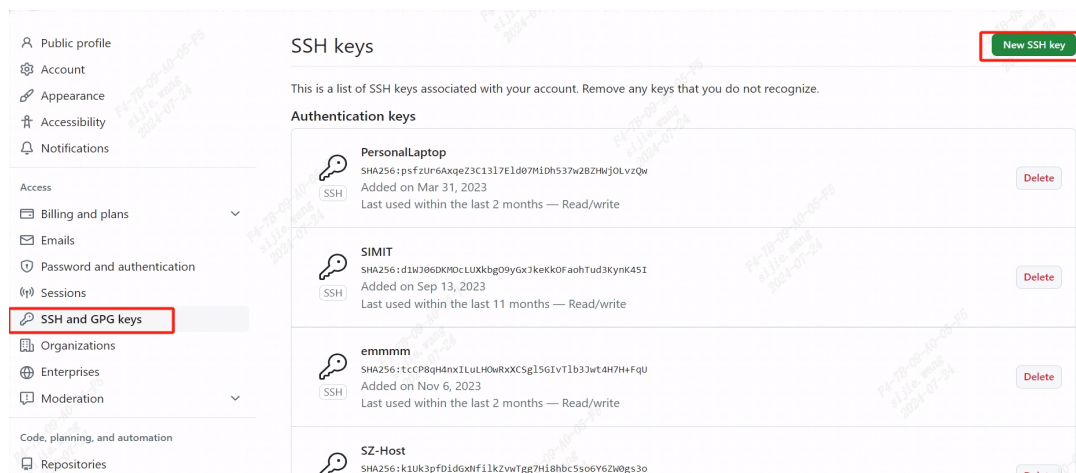
### 1. 设置账号邮箱

```
git config --global user.name "your_name"  
git config --global user.email "your_email@example.com"
```

### 2. 配置密钥

```
ssh-keygen -t ed25519 -C "your_email@example.com"  
cat ~/.ssh/id_ed25519.pub
```

### 3. 将公钥添加到 github



### 4. 验证 ssh 是否配置成功

```
ssh -T git@github.com
```

## 2.4 获取 SDK 的方式

### 1. 使用 git 从 github 拉取最新 SDK 源码，见：<https://github.com/sophgo/sophpi>

```
git clone -b sg200x-evb git@github.com:sophgo/sophpi.git  
# 获取 v4.1.0 SDK  
./sophpi/scripts/repo_clone.sh --gitclone sophpi/scripts/subtree.xml
```

## 2.5 编译

### 2.5.1 环境变量说明

编译前置动作最主要是为了设置两个环境变量：\$CHIP, \$BOARD,

**\$CHIP** 变量是需要根据用户的 SOC 来做设置。

**\$BOARD** 变数是针对每张 EVB, 有不同的驱动, 必须要正确设置。

例如:

\$BOARD=wevb\_0008a\_spinor: 是 SPINOR + DDR2 1333 64 MB 硬件组合

\$BOARD=wevb\_0009a\_spinand: 是 SPINAND + DDR3 1866 128MB 硬件组合

注: wevb\_0008a / wevb\_0009a 可以直接看 EVB 上雷射型号得知。

### 2.5.2 编译整个软件包

设定环境变量前需要先透过下列命令初始化环境, 系统会列出目前 SDK 支持的 IC 以及 EVB 版本号.

```
$ source build/cvsetup.sh

-----
Usage:
(1) menuconfig - Use menu to configure your board.
    ex: $ menuconfig
(2) defconfig $CHIP_ARCH - List EVB boards($BOARD) by CHIP_ARCH.
    ** cv183x ** -> ['cv1829', 'cv1832', 'cv1835', 'cv1838', 'cv9520', 'cv7581']
    ** cv182x ** -> ['cv1820', 'cv1821', 'cv1822', 'cv1823', 'cv1825', 'cv1826', 'cv7327', 'cv7357']
    ** cv181x ** -> ['cv181x', 'cv1823a', 'cv1821a', 'cv1820a', 'cv1811h', 'cv1811c', 'cv1810c', 'cv1812h']
    ** cv180x ** -> ['cv180x', 'cv1800b', 'cv1800c', 'cv1801b', 'cv1801c']
    ex: $ defconfig cv183x    (3) defconfig $BOARD - Choose EVB board settings.
    ex: $ defconfig cv1835_wevb_0002a
    ex: $ defconfig cv1826_wevb_0005a_spinand
    ex: $ defconfig cv180x_fpga_c906
-----
```

初始化之后, 可以下列两种方式进行编译组态设定

#### 2.5.2.1 透过 defconfig 设定

选取 IC : 以 cv180x 为例, 系统会打印出 cv180x 内建支持的 EVB (\$CHIP\_\$BOARD) 板。

```
$ defconfig cv180x
* cv180x * the available cvitek EVB boards
cv180x - cv180x_fpga [FPGA]
        cv180x_palladium [PALLADIUM]
cv1800b - cv1800b_wdmb_0008a_spinor [C906B + SPINOR 8MB + QFN SIP 64MB]
        cv1800b_wevb_0008a_spinor [C906B + SPINOR 16MB + QFN SIP 64MB]
```

(下页继续)

(续上页)

```

cv1800c - cv1800c_wevb_0009a_spinor [C906B + SPINOR 16MB + QFN SIP 64MB]
cv1801b - cv1801b_wevb_0008a_spinor [C906B + SPINOR 16MB + QFN SIP 128MB]
cv1801c - cv1801c_wdmb_0009a_spinor [C906B + SPINOR 16MB + QFN SIP 128MB]
          cv1801c_wevb_0009a_spinand [C906B + SPINAND 256MB + QFN SIP 128MB]
          cv1801c_wevb_0009a_spinor [C906B + SPINOR 16MB + QFN SIP 128MB]
          cv1812h_wevb_0007a_spinor [C906B + SPINOR 16MB + BGA SIP 256MB]

```

选取 EVB 版号为 cv1801c\_wevb\_0009a\_spinor，此时系统会列出自动设定好的环境变数。(后续亦可用 cvi\_print\_env 来打印目前使用的环境变数)

```

$ defconfig cv1801c_wevb_0009a_spinor

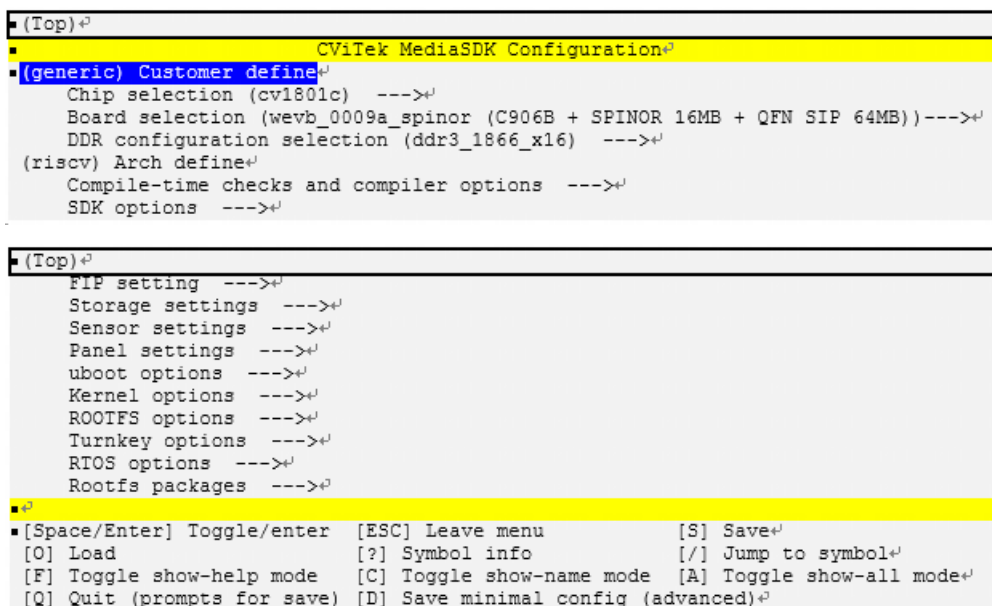
===== Environment Variables =====

PROJECT: cv1801c_wevb_0009a_spinor, DDR_CFG=ddr3_1866_x16
CHIP_ARCH: CV180X, DEBUG=0
SDK_VERSION: musl_riscv64, RPC=0
ATF options: ATF_KEY_SEL=default, BL32=1
Linux source folder: linux_5.10, Uboot source folder: u-boot-2021.10
CROSS_COMPILE_PREFIX: riscv64-unknown-linux-musl-
ENABLE_BOOTLOGO: 0
Flash layout xml: build/boards/cv180x/cv1801c_wevb_0009a_spinor/partition/partition_spinor.xml
Sensor tuning bin: gcore_gc4653
Output path: install/soc_cv1801c_wevb_0009a_spinor

```

### 2.5.2.2 透过 Menuconfig 设定

初始化之后，键入 menuconfig 进入以下页面，即可选择各种 SDK 内部设定，包含 CHIP，EVB 板号等等。



```

■ (Top)
■ CViTek MediaSDK Configuration
■ (generic) Customer define
  Chip selection (cv1801c) --->
  Board selection (wevb_0009a_spinor (C906B + SPINOR 16MB + QFN SIP 64MB)) --->
  DDR configuration selection (ddr3_1866_x16) --->
  (riscv) Arch define
  Compile-time checks and compiler options --->
  SDK options --->

■ (Top)
  FIP setting --->
  Storage settings --->
  Sensor settings --->
  Panel settings --->
  uboot options --->
  Kernel options --->
  ROOTFS options --->
  Turnkey options --->
  RTOS options --->
  Rootfs packages --->

■
■ [Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
  [O] Load [?] Symbol info [/] Jump to symbol
  [F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
  [Q] Quit (prompts for save) [D] Save minimal config (advanced)

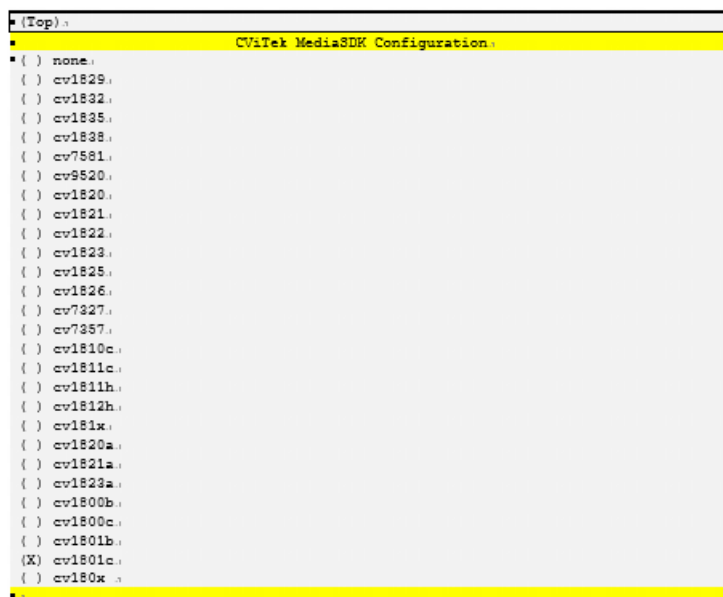
```

配置过程可以透过 [Enter] [Space] [ESC] 等进行设置/返回的动作

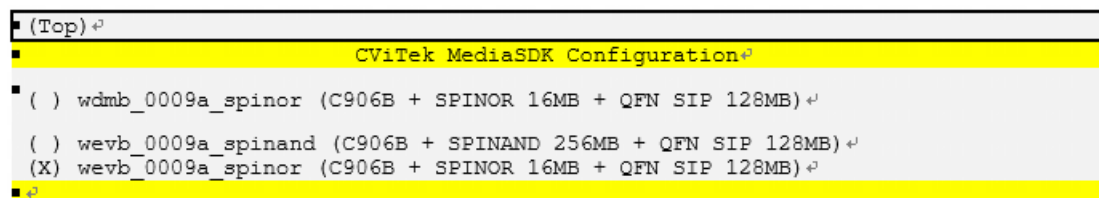
配置完毕后，按下 [S] 储存配置文件，接着按下 [Q] 离开图形化接口

(或者按下 ESC, 会自动弹出是否需要储存的图形)

选取 IC (以 cv1801c 为例)



EVB 版号会列出相对应的选择, 选取 EVB 的同时也会决定编译出的 image 适配的 DDR 以及 Flash Size。(以这个例子选取的 EVB 上所带的 DDR 为 DDR3, Flash 为 16MB 的 SPINOR Flash)



最后退出选择储存设定 (设定会储存于 ./build/.config), 即可完成 SDK 编译组态的选定。

最后退出选择储存设定 (设定会储存于 ./build/.config), 即可完成 SDK 编译组态的选定。

### 2.5.3 编译完整 SDK 文档

执行编译, 会得到可用于烧录的 images。

```

cvitek@cvitek-VirtualBox:~/working_dir$ build_all
. Run build_uboot () function
...
/work/install/cv1801c_wevb_0009a_spinor/upgrade.zip done!
  
```

编译出的档案会放置于 ./install/soc\_<EVB Name>/ 之下 fip.bin

## 2.5.4 编译部份 SDK 文档

### 2.5.4.1 单独编译 Uboot

每个 EVB 板会在特定位置定义进入 U-Boot 之前, EVB 需要采取的初始化动作或是定义特定 PINMUX。以 cv1801c\_wevb\_0009a\_spinor 这张板子为例, 会定义在:

build/boards/mars/\$CHIP\_\$BOARD/u-boot/cvi\_board\_init.c

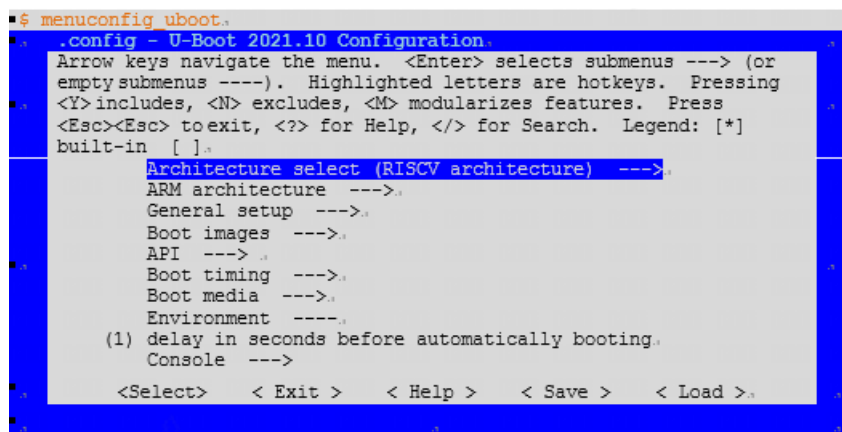
```
int cvi_board_init(void)
{
#ifdef CV180X_QFN_88_PIN
    PINMUX_CONFIG(PAD_MIPI_TXP1, IIC2_SCL);
    PINMUX_CONFIG(PAD_MIPI_TXM1, IIC2_SDA);
    PINMUX_CONFIG(PAD_MIPI_TXP0, XGPIOC_13);
    PINMUX_CONFIG(PAD_MIPI_TXM0, CAM_MCLK1);
#elif defined(CV180X_QFN_88_PIN_38)
    return 0;
}
```

其对应的 u-boot 组态, 定义在:

./build/boards/mars/\$CHIP\_\$BOARD/u-boot/\$CHIP\_\$BOARD\_defconfig

```
Partial cvitek_cv1801c_wevb_0009a_spinor_defconfig
CONFIG_RISCV=y
CONFIG_SYS_MALLOC_F_LEN=0x2000
CONFIG_NR_DRAM_BANKS=1
CONFIG_DEFAULT_DEVICE_TREE="cv180x_asic"
CONFIG_IDENT_STRING=" cvitek_cv180x"
...
```

以图形化接口修改 Uboot Config



退出后会把设定储存在:

./u-boot/build/"\$CHIP"\_"\$BOARD"/.config

执行编译

```
$ build_uboot
```

完成后会生成 fip.bin

Makefile 中编译 U-Boot 的片段。

```
u-boot-build: ${UBOOT_PATH}/${UBOOT_OUTPUT_FOLDER} ${UBOOT_CVIPART_DEP}
${UBOOT_OUTPUT_CONFIG_PATH}
$(call print_target)
${Q}rm -f ${UBOOT_CVI_BOARD_INIT_PATH}
${Q}ln -s ${BUILD_PATH}/boards/${PROJECT_FULLNAME}/u-boot/cvi_board_init.c
${UBOOT_CVI_BOARD_INIT_PATH}
${Q}${MAKE} -j${NPROC} -C ${UBOOT_PATH} olddefconfig
${Q}${MAKE} -j${NPROC} -C ${UBOOT_PATH} all
$(call uboot_compress_action)
```

#### 2.5.4.2 单独编译 kernel

修改 kernel (ex: \*.dts, 内核), 重新编译 Linux kernel image。

每张 EVB 都有对应的 dts 档案来定义其 device tree, 以 cv1801c\_wevb\_0009a\_spinor 为例, 其 DTS 档案定义在:

```
./build/boards/cv180x/"$CHIP"_"$BOARD"/dts_riscv/"$CHIP"_"$BOARD".dts
```

```
#!/dts-v1/;
#include "cv180x_base_riscv.dtsi"
#include "cv180x_asic_qfn.dtsi"
#include "cv180x_asic_spinor.dtsi"
#include "cv180x_default_memmap.dtsi"
/ {
};
```

其相对应的 linux 组态, 定义在:

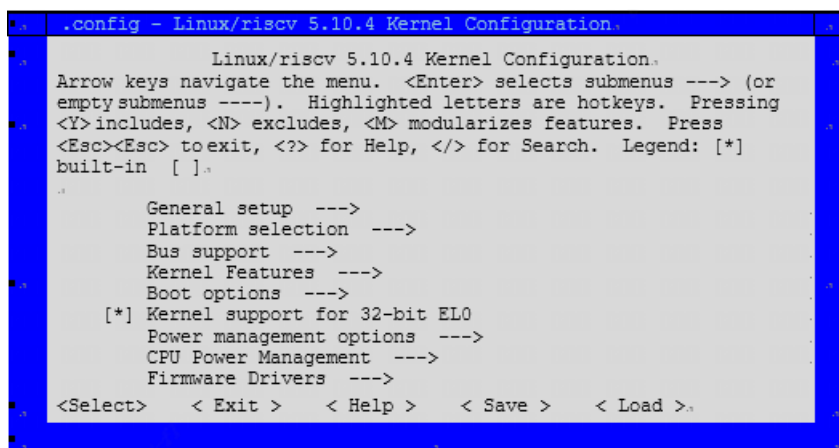
```
./build/boards/"$CHIP"_"$BOARD"/linux/"$CHIP"_"$BOARD"_defconfig
```

```
Partial cv1801c_wevb_0009a_spinor
CONFIG_SYSVIPC=y
CONFIG_POSIX_MQUEUE=y
CONFIG_NO_HZ_IDLE=y
CONFIG_HIGH_RES_TIMERS=y
CONFIG_PREEMPT=y
CONFIG_IKCONFIG=y
CONFIG_IKCONFIG_PROC=y
CONFIG_LOG_BUF_SHIFT=15
CONFIG_BLK_DEV_INITRD=y
...
```

以图形化接口修改 Kernel Config

```
$ menuconfig_kernel
```





退出后会把设定储存在:

`./linux/build/"$CHIP"_"$BOARD"/.config`

```
$ build_kernel
```

完成后会生成 `boot.spinor`

Makefile 中编译 Kernel 的片段。

```
kernel-build: ${KERNEL_OUTPUT_CONFIG_PATH}
    ${call print_target}
    ${Q}echo LOCALVERSION=${LOCALVERSION}
    ${Q}${(MAKE)} -j${NPROC} -C ${KERNEL_PATH} O=${KERNEL_PATH}/${KERNEL_
    ↪OUTPUT_FOLDER}
olddefconfig
    ${Q}${(MAKE)} -j${NPROC} -C ${KERNEL_PATH}/${KERNEL_OUTPUT_FOLDER}Image_
    ↪modules
    ${Q}${(MAKE)} -j${NPROC} -C ${KERNEL_PATH}/${KERNEL_OUTPUT_FOLDER}modules_
    ↪install
headers_installINSTALL_HDR_PATH=${KERNEL_PATH}/${KERNEL_OUTPUT_FOLDER}/
    ↪${(ARCH)}/usr
    ${Q}ln -sf ${KERNEL_PATH}/${KERNEL_OUTPUT_FOLDER}/${(ARCH)}/usr/include
    ${KERNEL_PATH}/${KERNEL_OUTPUT_FOLDER}/usr/include
```

#### 2.5.4.3 单独编译 middleware

修改 `middleware (cvi_test / sample_dsi)`, 重新编译 `middleware` 及 `system`

生成的 `Install/PROJECT_NAME/system.*` 包含最新的 `middleware`

```
$ build_middleware; pack_rootfs
```

编译 Middleare 的 shell script 片段

```
pushd "$MW_PATH"/component/isp
make all
popd

pushd "$MW_PATH"/sample
make all
```

build\_middleware 会针对 Sensor driver (位于 middleware/component/isp/ 下) 以及 sample application (位于 middleware/sample/下) 重新编译, 末了 pack\_rootfs 会将变更后的 driver 以及 application 包装成可烧录映像档。

## 2.6 磁盘分区

Mars SDK 会生成以下的 image file, 每个文件代表不同的分区, 分列如下:

- FIP : Bootloader/U-Boot 分区
  - CV180X/ CV181X C906 采用 FSBL+OPENSBI+UBOOT 架构, 最后打包后也复用 (FIP) 档名, 方便后续使用。
- 2nd(双系统) : Yun on Processor (YOC) 所在分区
- BOOT : Partition for Linux Kernel 分区
- MISC : Boot Logo 分区
- ROOTFS : root file system 分区
- SYSTEM: CVITEK libraries 所在分区
- DATA : 使用这资料分区

注: 2nd 分区为双系统特有分区, 仅在双系统环境下存在

### 2.6.1 磁盘分区修改

相同的开发板可能会上不同的 Flash, SDK 会以不同的板号作区隔。比如说 cv1811c\_wdmb\_0006a\_spinand 与 cv1811c\_wdmb\_0006a\_spinor 分别代表在开发板上的 Flash 各为 SPINAND 及 SPINOR, 分区档案分别置于

./build/boards/<CHIP>/<EVB\_Name>/partition/partition\_<physical\_partition>.xml

Note: physical\_partition 支持 SPINAND/SPINOR。

例如 cv1811c\_wdmb\_0006a\_spinor 的分区档案, 陈列如下:

单系统环境下:

```
build/boards/cv181x/cv1811c_wdmb_0006a_spinor/partition/partition_spinor.xml
<physical_partition type="spinor">
  <partition label="fip" size_in_kb="800" readonly="false" file="fip.bin"/>
  <partition label="BOOT" size_in_kb="2600" readonly="false" file="boot.spinor"/>
  <partition label="ENV" size_in_kb="64" file="" />
  <partition label="ROOTFS" size_in_kb="4000" readonly="false" file="rootfs.spinor" />
  <partition label="DATA" size_in_kb="512" readonly="false" file="data.spinor" mountpoint="/
  ↪mnt/data" type="jffs2" />
</physical_partition>
```

双系统环境下:

```
build/boards/cv181x/cv1811c_wdmb_0006a_spinor/partition/partition_spinor.xml
<physical_partition type="spinor">
  <partition label="fip" size_in_kb="512" readonly="false" file="fip.bin"/>
  <partition label="2nd" size_in_kb="3072" readonly="false" file="yoc.bin"/>
  <partition label="BOOT" size_in_kb="5120" readonly="false" file="boot.spinor"/>
  <partition label="MISC" size_in_kb="128" readonly="false" file="logo.jpg"/>
  <partition label="PARAM" size_in_kb="64" file="" />
  <partition label="PARAM_BAK" size_in_kb="64" file="" />
  <partition label="ENV" size_in_kb="64" file="" />
  <partition label="ENV_BAK" size_in_kb="64" file="" />
  <partition label="ROOTFS" size_in_kb="3392" readonly="false" file="rootfs.spinor" />
  <partition label="DATA" size_in_kb="1024" readonly="false" file="data.spinor" mountpoint="/
  ↪mnt/data" type="jffs2" />
</physical_partition>
```

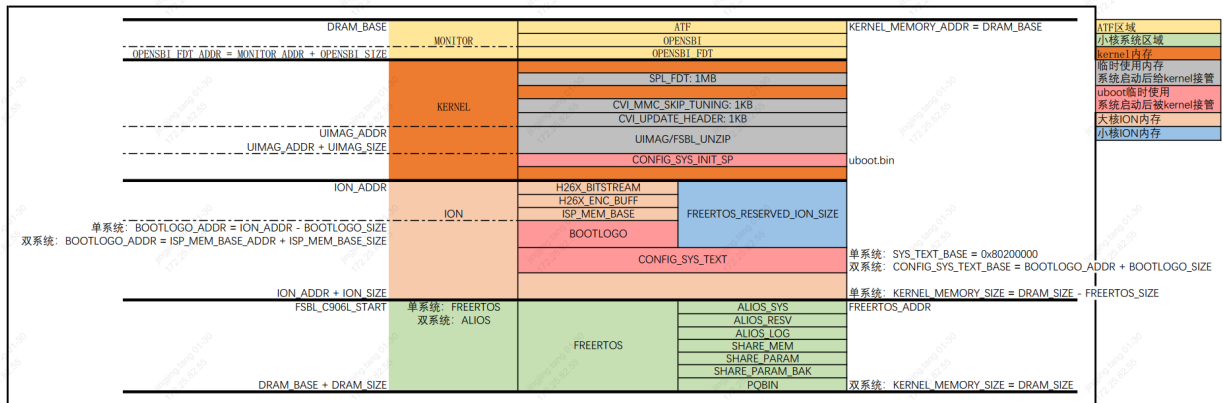
- physical\_partiti type: flash 种类。
- partition label: 分区名称。
- size\_in\_kb: 分区大小 (以 KB 为单位)。
- file: 所指向的 image file 名称。
- type: (在 partition label 栏位中) 文件系统格式。
- mountpoint: 分区挂载路径。

注: 2nd 分区为双系统特有分区, 仅在双系统环境下存在

## 2.7 内存映射

内存的基本分配如下图所示。

- MONITOR: ATF 固件使用内存, 包括 bl31 和 bl32。
- KERNEL\_MEMORY\_ADDR: kernel 管理的内存的起始地址。如果使用单系统, 系统启动后, KERNEL 管理 DRAM\_SIZE - FREERTOS\_SIZE 大小的内存空间。如果使用双系统, 系统启动后, KERNEL 管理 DRAM\_SIZE 大小的内存空间。
- UIMAG/FSBL\_UNZIP: 在 bootload 阶段, kernel 区域内存都可以被任意使用, bl2 会使用里面的一段区域进行镜像的解压缩, uboot 系统会运行在这一段, 但是 kernel 运行起来后, 这一段将被 kernel 接管。
- ION: 大核 ION 驱动使用的内存, 从 FREERTOS\_ADDR 往前分配 ION\_SIZE 大小。
- FREERTOS\_RESERVED\_ION\_SIZE: 使用 FREERTOS 时, 小核保留的 ION 内存。
- FREERTOS\_ADDR: 小核使用内存。当使用双系统 alios 时, 在内存尾部保留一块共享内存, 大小核通信使用, ALIOS\_LOG + SHARE\_MEM + SHARE\_PARAM\_SHARE\_PARAM\_BAK + PQBIN。
- ALIOS\_RESV: 使用 alios 时, 多媒体会使用这部分内存。



具体的分配地址和大小定义在 memmap.py 文件中。在编译时，解析 memmap.py 并生成 cvi\_board\_memmap.\* 文件，提供给 linux、u-boot、fsbl 等编译使用。

```
$ cat build/boards/cv181x/cv1811h_wevb_0007a_spinor/memmap.py
SIZE_1M = 0x100000
SIZE_1K = 1024

# Only attributes in class MemoryMap are generated to .h
class MemoryMap:
    # No prefix "CVIMMAP_" for the items in _no_prefix[]
    _no_prefix = [
        "CONFIG_SYS_TEXT_BASE" # u-boot's CONFIG_SYS_TEXT_BASE is used without CPP.
    ]

    DRAM_BASE = 0x80000000
    DRAM_SIZE = 128 * SIZE_1M
    .....
```

cvi\_board\_memmap.txt 文件会统计 fsbl/u-boot/kernel 使用到的内存情况

```
$ cat build/output/cv1811h_wevb_0007a_spinor/cvi_board_memmap.txt
FBSL stage:
| Name          | MONITOR   | OPENSBI_FDT | FSBL_UNZIP | UIMAG      | FREERTOS   |
| Start Address | 0x80000000 | 0x80080000  | 0x81800000 | 0x81800000 | 0x87e00000 |
| End Address   | 0x80000000 | 0x80080000  | 0x82800000 | 0x82800000 | 0x88000000 |
| Size          | 0x0        | 0x0         | 0x1000000  | 0x1000000  | 0x200000   |
| Size(M/K/B)   | 0M         | 0M          | 16M        | 16M        | 2M         |

uboot stage:
| Name          | MONITOR   | OPENSBI_FDT | CVI_UPDATE_HEADER | UIMAG      | |
| Start Address | 0x80000000 | 0x80080000  | 0x817ffc00        | 0x81800000 | 0x82800000 |
| End Address   | 0x80000000 | 0x80080000  | 0x81800000        | 0x82800000 | 0x82800000 |
| Size          | 0x0        | 0x0         | 0x400            | 0x1000000  | 0x0        |
| Size(M/K/B)   | 0M         | 0M          | 1K               | 16M        | 0M         |
```

(下页继续)

(续上页)

kernel stage:

Name	KERNEL_MEMORY	MONITOR	OPENSBI_FDT	FRAMEBUFFER	H26X_	
→ BITSTREAM	ION	H26X_ENC_BUFF	ISP_MEM_BASE	FREERTOS		
Start Address	0x80000000	0x80000000	0x80080000	0x8313e000	0x83300000	0x83300000
→ 0x83500000	0x83500000	0x87e00000				
End Address	0x87e00000	0x80000000	0x80080000	0x83300000	0x83500000	0x87e00000
→ 0x83500000	0x84900000	0x88000000				
Size	0x7e00000	0x0	0x0	0x1c2000	0x200000	0x4b00000
→ 0x1400000	0x200000					
Size(M/K/B)	126M	0M	0M	1800K	2M	75M
→ 20M	2M					
-----						
→ -----						

PS: When the kernel is booted, it will overwrite the FSBL and uboot memory space

## 2.7.1 内存映射修改

通过修改 memmap.py 文件中的参数，可以修改内存映射。

- 相同“DRAM\_SIZE”的内存分配基本相同，按需可以修改 ION\_SIZE、FREERTOS\_SIZE 等参数。
- 不同“DRAM\_SIZE”的内存分配，对应按需修改 DRAM\_SIZE、FREERTOS\_SIZE、ION\_SIZE 及其 ADDR 等参数。
- 参数修改要计算好内存的起始地址，当心出现踩内存。

# 3 烧录说明

## 3.1 使用前准备

- 2.4.2.3 前述章节产生的烧录档案。
- FAT32 格式的 Micro SD 卡。

## 3.2 操作过程

- 将烧录档案（如下表）放到 SD 卡中。
- 将 SD 卡插入 CVITEK EVB 的 SD 卡槽中。
- 将平台重开机。

## 3.3 操作实例

使用前确认文件

以 SPINAND 为例	以 SPINOR 为例	以 EMMC 为例
1.8M yoc.bin(双系统) 535K fip.bin 2.6M fw_payload_uboot.bin 4.0M ramboot.itb 2.6M boot.spinand 3M rootfs.spinand 1.9M cfg.spinand 1.9M system.spinand	1.8M yoc.bin(双系统) 508K fip.bin 182K fip_spl.bin 2.49M fw_payload_uboot.bin 4.3M ramboot.itb 3.7M boot.spinor 3.18M rootfs.spinor 292B data.spinor	1.8M yoc.bin(双系统) 486K fip.bin 2.5M fw_payload_uboot.bin 5.3M ramboot.itb 2.8M boot.emmc 6.7M rootfs.emmc 9.5M cfg.emmc 4.7M system.emmc

注：上表中 yoc.bin 文件为双系统独有文件，仅在双系统环境下存在

插入 SD 卡，并将 CV180X/CV181X 平台接上电源后开机，自动启动烧录程序，DL flag 表示主 IC 侦测到目前 SD Card 中存在可以烧录的档案。

```
In: serial
Out: serial
Err: serial
Net:
Warning: ethernet@4070000 (eth0) using random MAC address - 3a:6d:3f:aa:9e:d6
eth0: ethernet@4070000
Hit any key to stop autoboot: 0
## Resetting to default environment
Start SD downloading.....
```

平台烧录完成时，可于 UART 端口看到以下信息，将平台断电，拔出 SD 卡，再重开机，即完成烧录过程。（log 会针对每个分区，显示读取档案，写入 Flash 所在）

```
## Resetting to default environment
Start SD downloading..
497664 bytes read in 25 ms (19 MiB/s)
spinor id = 1C 71 18
SF: Detected EN25QX128A with page size 256 Bytes, erase size 64 KiB, total 16 MiB
.....
Header Version:1
2996612 bytes read in 137 ms (20.9 MiB/s)
device 0 offset 0x100000, size 0x2db944
0 bytes written, 2996548 bytes skipped in 0.125s, speed 23972384 B/s
sf update speed 22.196 MB/s
64 bytes read in 3 ms (20.5 KiB/s)
Header Version:1
.....
Saving Environment to SPIFlash... Erasing SPI flash...Writing to SPI flash...done
Valid environment: 1
OK
cv180x_c906#
```

## 3.4 打包烧录器烧录镜像

pack\_prog\_img 是构建系统中用于打包镜像的通用函数，该镜像只支持烧录器烧录，支持多种存储类型（nand、nor、emmc）。

用法：

```
pack_prog_img [nandid]

# 其中 nandid = 0x{DID/MID} 为可选参数，仅在 storage 类型为 nand_
→时需要，具体值可查找datasheet或在板端dmesg中查找。
# 例如： nand: device found, Manufacturer ID: 0x0b, Chip ID: 0x35 ; 则nandid = 0x350b
```

输出目录：

{SDK}/install/{board}/burnimages/

不同存储介质的输出文件如下：

- **NAND**: 输出文件与 {SDK}/install/{board}/rawimages/ 目录下的文件相同（仅 fip\_spl.bin 特殊处理,fip.bin 未拷贝），同时包含分区表信息文件 partition\_info.txt

- **NOR** : 仅输出 Data.bin 文件
- **eMMC**: 输出 Data.bin 和 Boot.bin (包含 fip.bin) 两个文件

在其他 shell 脚本中调用示例:

```
source build/envsetup_soc.sh
defconfig cv1811c_wevb_0006a_spinand
pack_prog_img 0x95c8
```

## 3.5 注意事项

请确认 SD Card 被正确格式化为 FAT32 格式。



# 4 EVB 接口说明

下方图片为 CV180x EVB

A.UART Debug Port

B.Sensor 0 EVB 插槽

C.Sensor 1 EVB 插槽

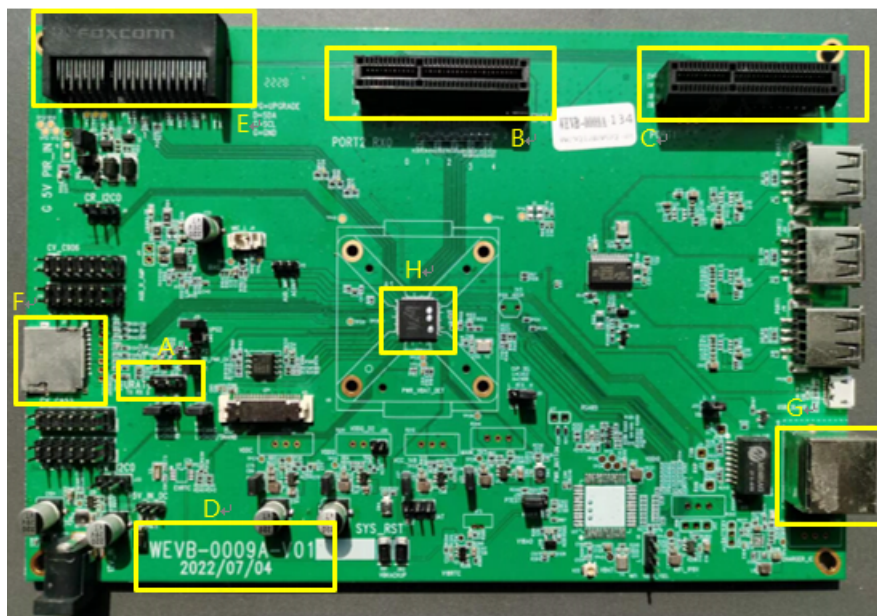
D.EVB 版号

E.Panel EVB 插槽

F.SD 卡插槽

G.Ethernet 0 连接口

H. 主 IC CV180x



下方图片为 CV181x EVB

I.UART Debug Port

J.Sensor 0 EVB 插槽

K.Sensor 1 EVB 插槽

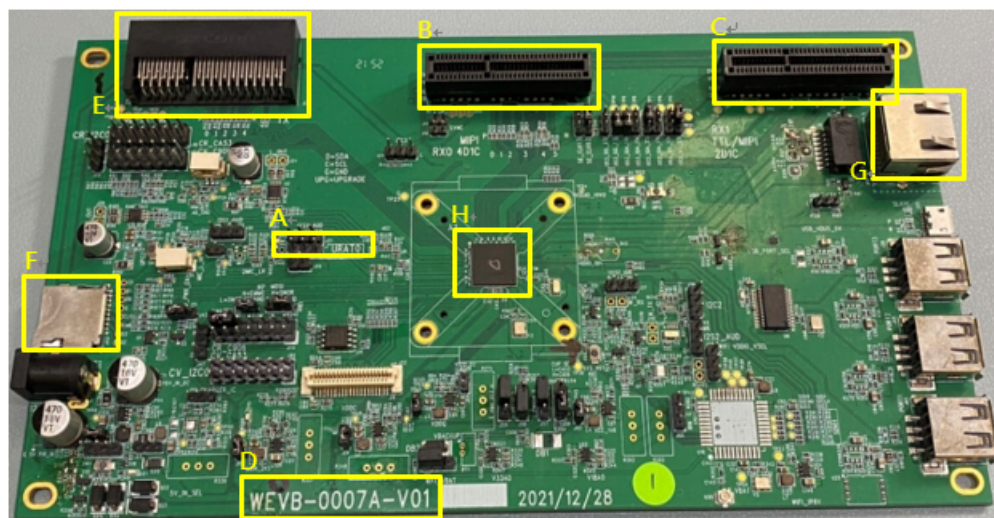
L.EVB 版号

M.Panel EVB 插槽

N.SD 卡插槽

O.Ethernet 0 连接口

P. 主 IC CV181x



# 5 根文件系统 (rootfs)

## 5.1 根文件系统简介

内核是 Linux 操作系统的核心，文件系统是用户和操作系统沟通的主要工具。所以要使用 Linux 时，要先了解文件系统原理。

根文件系统结构是以“/”为“根 (root)”起始的树状目录结构，当内核程序映像 (uImage) 启动会挂载一个设备 (ex:eMMC) 在根目录上，根文件系统通常存放在内部存储器 (DRAM) 或非挥发内存 (FLASH) 中，或是透过网络存取的文件系统 (NFS)。所有应用程序和函式库都会按照分类放入文件系统中，以下列出根文件系统目录结构图。

```
/ 根目录
├── bin 可执行文件
├── dev 设备文件
├── etc 系统配置文件(ex: 启动文件)
├── home 用户目录
├── init 开机执行script
├── kdump 内核除错目录
├── lib 函式库包含glibc, shared library和内核模块
├── mnt 临时文件系统的挂载点
├── proc 内核和行程信息的虚拟文件系统
├── sbin 系统管理的可执行文件
├── sys 系统设备和文件层次结构，提供内核数据数据
├── usr 此目录下包含用户自定义应用程序和文文件
└── var 存放系统日志和服务程序文件
```

## 5.2 Rootfs

本章节是描述文件系统之组成方式，详细路径于 ramdisk/rootfs/

### 5.2.1 Pre-build rootfs 架构

文件系统之结构目录主要拆了三个类型，且逐层迭加于 Rootfs，将于下方分别描述：

- **Basic rootfs:**

现阶段本公司提供了基于以下四种 Arch 产生之 pre-build rootfs 档案

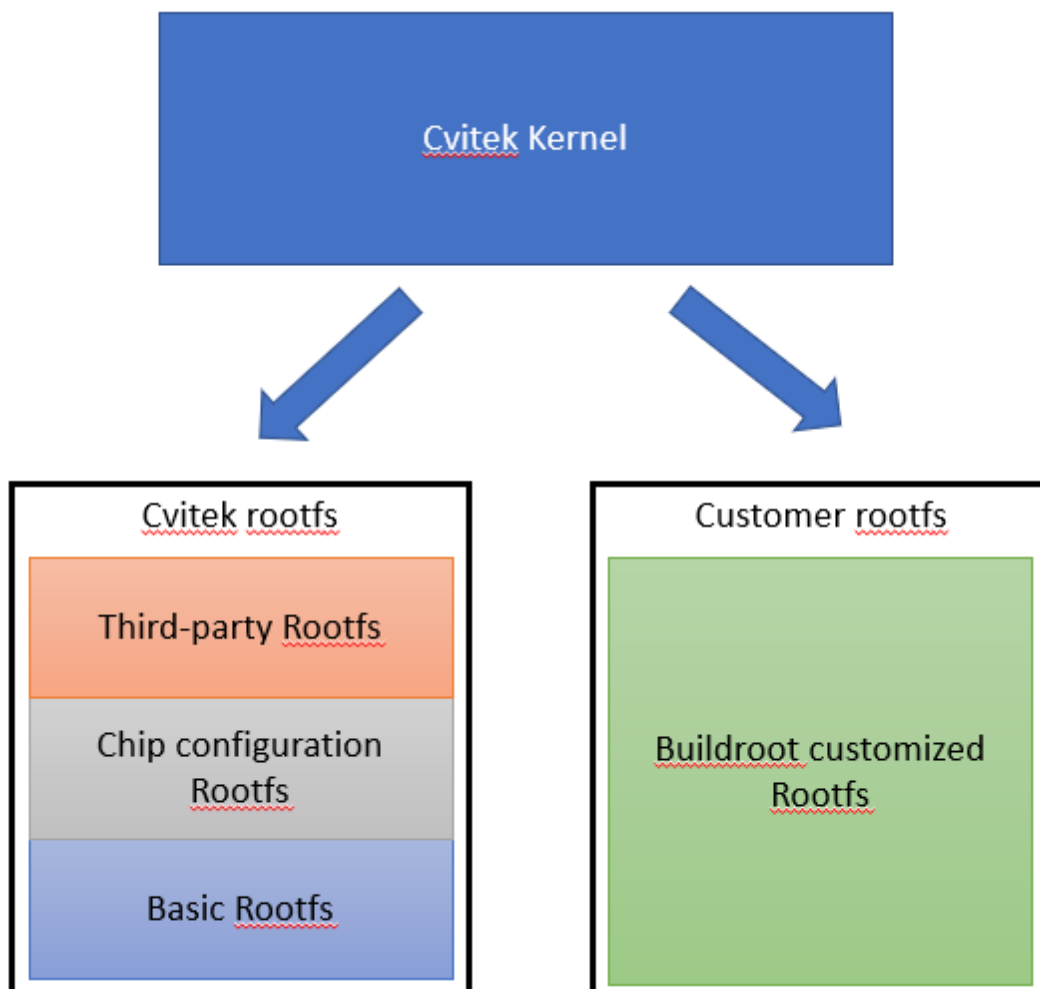
Arch	Libc	Pre-build ramdisk path
Arm	glibc	ramdisk/rootfs/common_arm/
Arm	uclibc	ramdisk/rootfs/common_uclibc/
Aarch64	glibc	ramdisk/rootfs/common_arm64/
Riscv64	glibc	ramdisk/rootfs/common_riscv64/
Riscv64	musl	ramdisk/rootfs/common_musl64/

- **Processor configuration rootfs:**

本公司将所有 Processorset 相依之开机设置均放置于 ramdisk/rootfs/overlay/\$CHIP

- **Third-party rootfs:**

本公司将所有第三方软件编译出来之 library、utility、related file 均放置于 ramdisk/rootfs/public/



可以透过选单的方式决定需要那些 Third-party software 要放置进 Rootfs

```
$ menuconfig
```

```

■ (Top) ↵
■ CViTek MediaSDK Configuration ↵
■ Chip selection (cv1801c) --->↵
  Board selection (wevb_0009a_spinor (C906B + SPINOR 16MB + QFN SIP 64MB)--->↵
  DDR configuration selection (ddr3_1866_x16) --->↵
  (arm64) Arch define ↵
  Compile-time checks and compiler options --->↵
  SDK options --->↵
  FIP setting --->↵
  Storage settings --->↵
  Sensor settings --->↵
  panel settings --->↵
  Kernel options --->↵
  ROOTFS options --->↵
  Turnkey options --->↵
  RTOS options --->↵
  Rootfs packages --->↵
■ ↵
■ [Space/Enter] Toggle/enter [ESC] Leave menu [S] Save ↵
  [O] Load [?] Symbol info [/] Jump to symbol ↵
  [F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode ↵
  [Q] Quit (prompts for save) [D] Save minimal config (advanced) ↵
  
```

```

■ (Top) → Rootfs packages
■ CViTek MediaSDK Configuration
■ [ ] Target adbd
[ ] Target package of AP6201BM fw files
[ ] Target package bluetooth
[ ] Target package cvitracer
[ ] Target package dropbear
[ ] Target package e2fsprogs
[*] Target gdbserver
[ ] Target package htop
[ ] Target package libbtrace
[ ] Target package libcrypto
[ ] Target package libcurl
[ ] Target package libevent
[ ] Target package libiperf
[ ] Target package libiw
[ ] Target package libopenssl
[ ] Target package libprotobuf
[ ] Target package libz
■ ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
■ [Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
[O] Load [?] Symbol info [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)

```

## 5.2.2 编译来自 buildroot 的 rootfs

此章节是示范如何从 buildroot 产生 rootfs 并且于 EVB 上面运行的例子，若采用上一章节描述的 pre-build rootfs，可忽略此章节。

1. 拉取 buildroot 仓库

<https://github.com/sophgo/buildroot-2021.05>

2. 配置 buildroot

```
$ cd buildroot-2021.05
```

```
$ make menuconfig
```

```

Buildroot -gd2c753ad24-dirty Configuration
Arrow keys navigate the menu. <Enter> selects submenus --- (or empty submenus ---). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

Target options --->
Build options --->
Toolchain --->
System configuration --->
Kernel --->
Target packages --->
Filesystem images --->
Bootloaders --->
Host utilities --->
Legacy config options --->

<Select> <Exit> <Help> <Save> <Load>

```

3. 设置 Arch Info & Toolchain & Packages

configs 目录下有一些默认配置文件，可执行 `make cvitek_XXX_defconfig` 快速配置  
设置架构



## ARM

```

Target options
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

Target Architecture (AArch64 (little endian)) --->
Target Binary Format (ELF) --->
Target Architecture Variant (cortex-A53) --->
Floating point strategy (FP-ARMv8) --->

<Select> <Exit> <Help> <Save> <Load>

```

## RISCV

```

Target options
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

Target Architecture (RISCV) --->
Target Binary Format (ELF) --->
Target Architecture Variant (Custom architecture) --->
*** Instruction Set Extensions ***
[*] Integer Multiplication and Division (M)
-- Atomic Instructions (A)
[*] Single-precision Floating-point (F)
[*] Double-precision Floating-point (D)
[*] Compressed Instructions (C)
[*] Vector Instructions (Vector 0.7 Instructions (V0P7)) --->
[*] T-HEAD Extensions
Target Architecture Size (64-bit) --->
Target ABI (lp64d) --->

<Select> <Exit> <Help> <Save> <Load>

```

## 设置工具链

## ARM

```

Toolchain
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

Toolchain type (External toolchain) --->
*** Toolchain External Options ***
Toolchain (Custom toolchain) --->
Toolchain origin (Pre-installed toolchain) --->
$(CROSS_COMPILE_PATH_64) Toolchain path
(aarch64-linux-gnu) Toolchain prefix
External toolchain gcc version (6.x) --->
External toolchain kernel headers series (4.6.x) --->
External toolchain C library (glibc/eglibc) --->
[*] Toolchain has SSP support?
[*] Toolchain has SSP strong support?
[*] Toolchain has RPC support?
[*] Toolchain has C++ support?
[ ] Toolchain has D support?
[*] Toolchain has Fortran support?
[*] Toolchain has OpenMP support?
[ ] Copy gdb server to the Target
*** Host GDB Options ***
[ ] Build cross gdb for the host
i(+)

<Select> <Exit> <Help> <Save> <Load>

```

## RISCV

```

Toolchain
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

*** Toolchain type (External toolchain) --->
*** Toolchain External Options ***
Toolchain (Custom toolchain) --->
Toolchain origin (Pre-installed toolchain) --->
($CROSS_COMPILE_PATH_MUSL_RISCV64) Toolchain path
(riscv64-unknown-linux-musl) Toolchain prefix
External toolchain gcc version (10.x) --->
External toolchain kernel headers series (5.10.x) --->
External toolchain C library (musl (experimental)) --->
[*] Toolchain has SSP support?
[*] Toolchain has SSP strong support?
[*] Toolchain has C++ support?
[ ] Toolchain has D support?
[ ] Toolchain has Fortran support?
[*] Toolchain has OpenMP support?
[ ] Copy gdb server to the Target
*** Toolchain Generic Options ***
( ) Extra toolchain libraries to be copied to target
( ) Target Optimizations
i(+)

<Select> <Exit> <Help> <Save> <Load>

```

## 设置需要安装的软件包

```

Target packages
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] feature is selected [ ] feature is excluded

[*] BusyBox
(package/busybox/busybox.config) BusyBox configuration file to use?
( ) Additional BusyBox configuration fragment files
[ ] Show packages that are also provided by busybox
[ ] Individual binaries
[ ] Install the watchdog daemon startup script
Audio and video applications --->
Compressors and decompressors --->
Debugging, profiling and benchmark --->
Development tools --->
Filesystem and flash utilities --->
Fonts, cursors, icons, sounds and themes --->
Games --->
Graphic libraries and applications (graphic/text) --->
Hardware handling --->
Interpreter languages and scripting --->
Libraries --->
Mail --->
Miscellaneous --->
i(+)

<Select> <Exit> <Help> <Save> <Load>

```

## 4. 编译 buildroot

```
$ make
```

## 5. 得到 rootfs, 其位置在 output/images/rootfs.tar

## 6. 修改 build/Makefile 让 SDK 使用 buildroot 产生的 rootfs

```

buildroot-prepare:$(OUTPUT_DIR)/rootfs
$(call print_target)
#clean_all
rm -rf $(ROOTFS_DIR)/*
#extract buildroot roofs
tar xf $(BR_DIR)/output/images/rootfs.tar -C $(ROOTFS_DIR)
rootfs-pack:export CROSS_COMPILE_KERNEL=$(patsubst "%",%,$(CONFIG_CROSS_
→COMPILE_KERNEL))
rootfs-pack:export CROSS_COMPILE_SDK=$(patsubst "%",%,$(CONFIG_CROSS_COMPILE_
→SDK))
rootfs-pack:$(OUTPUT_DIR)/rawimages

```

(下页继续)



(续上页)

```
#rootfs-pack:rootfs-prepare
rootfs-pack:buildroot-prepare
rootfs-pack:
    $(call print_target)
    ${Q}printf '\033[1;36;40m Striping rootfs \033[0m\n'
ifeq (${FLASH_SIZE_SHRINK},y)
    ${Q}printf 'remove unneeded files'
    ${Q} $(COMMON_TOOLS_PATH)/spinand_tool/clean_rootfs.sh $(ROOTFS_DIR)
endif
    ${Q}find $(ROOTFS_DIR) -name "*.ko" -type f -printf 'striping %p\n' -exec $(CROSS_COMPILE_
→KERNEL)strip --strip-unneeded {} \;
    ${Q}find $(ROOTFS_DIR) -name "*.so*" -type f -printf 'striping %p\n' -exec $(CROSS_
→COMPILE_KERNEL)strip --strip-all {} \;
    ${Q}find $(ROOTFS_DIR) -executable -type f ! -name "*.sh" ! -path "*etc*" ! -path "*.ko" -printf
→'striping %p\n' -exec $(CROSS_COMPILE_SDK)strip --strip-all {} 2>/dev/null \;
```

### 7. 产生新的 ROOTFS 可烧录映像档

```
$ pack_rootfs
```

### 8. 透过第二章所提的步骤烧录到版端

## 5.2.3 将 rootfs 包装成可烧录映像档

将前述步骤产生之 rootfs folder 透过 mksquashfs 工具做最终打包, 压缩方式为 XZ, 最终产物即是可刻录于 Flash 上的 rootfs.spinor / rootfs.spinand / rootfs.emmc。

详细参考 build/Makefile 下之 rootfs-pack:

```
rootfs-pack:export CROSS_COMPILE_KERNEL=$(patsubst "%",%,$(CONFIG_CROSS_
→COMPILE_KERNEL))
rootfs-pack:export CROSS_COMPILE_SDK=$(patsubst "%",%,$(CONFIG_CROSS_COMPILE_
→SDK))
rootfs-pack:$(OUTPUT_DIR)/rawimages
rootfs-pack:rootfs-prepare
rootfs-pack:
    $(call print_target)
    ${Q}printf '\033[1;36;40m Striping rootfs \033[0m\n'
ifeq (${FLASH_SIZE_SHRINK},y)
    ${Q}printf 'remove unneeded files'
    ${Q} $(COMMON_TOOLS_PATH)/spinand_tool/clean_rootfs.sh $(ROOTFS_DIR)
endif
    ${Q}find $(ROOTFS_DIR) -name "*.ko" -type f -printf 'striping %p\n' -exec $(CROSS_COMPILE_
→KERNEL)strip --strip-unneeded {} \;
    ${Q}find $(ROOTFS_DIR) -name "*.so*" -type f -printf 'striping %p\n' -exec $(CROSS_
→COMPILE_KERNEL)strip --strip-all {} \;
    ${Q}find $(ROOTFS_DIR) -executable -type f ! -name "*.sh" ! -path "*etc*" ! -path "*.ko" -printf
→'striping %p\n' -exec $(CROSS_COMPILE_SDK)strip --strip-all {} 2>/dev/null \;
ifeq (${STORAGE_TYPE},spinor)
    ${Q}mksquashfs $(ROOTFS_DIR) $(OUTPUT_DIR)/rawimages/rootfs.sqsh -root-owned -comp xz
else
    ${Q}mksquashfs $(ROOTFS_DIR) $(OUTPUT_DIR)/rawimages/rootfs.sqsh -root-owned -comp xz -
→e mnt/cfg/*
```

(下页继续)

(续上页)

```

endif
ifeq ($(STORAGE_TYPE),spinand)
    ${Q}python3 $(COMMON_TOOLS_PATH)/spinand_tool/mkubiimg.py --ubionly $(FLASH_
    ↪PARTITION_XML) ROOTFS $(OUTPUT_DIR)/rawimages/rootfs.sqsh $(OUTPUT_DIR)/
    ↪rawimages/rootfs.spinand -b $(CONFIG_NANDFLASH_BLOCKSIZE) -p $(CONFIG_
    ↪NANDFLASH_PAGESIZE)
    ${Q}rm $(OUTPUT_DIR)/rawimages/rootfs.sqsh
else
    ${Q}mv $(OUTPUT_DIR)/rawimages/rootfs.sqsh $(OUTPUT_DIR)/rawimages/rootfs.
    ↪$(STORAGE_TYPE)
endif

```

## 5.2.4 Linux kernel 自动加载 rootfs

Linux kernel 会根据 uboot 设定之 bootargs 内的 root= 变量决定 rootfs 位于哪个 device

'root=...'

This argument tells the kernel what device is to be used as the root filesystem while booting. The default of this setting is determined at compile time, and usually is the value of the root device of the system that the kernel was built on. To override this value, and select the second floppy drive as the root device, one would use 'root=/dev/fd1'.

The root device can be specified symbolically or numerically. A symbolic specification has the form /dev/XXYN, where XX designates the device type (e.g., 'hd' for ST-506 compatible hard disk, with Y in 'a'-'d'; 'sd' for SCSI compatible disk, with Y in 'a'-'e'), Y the driver letter or number, and N the number (in decimal) of the partition on this device.

Note that this has nothing to do with the designation of these devices on your filesystem. The '/dev/' part is purely conventional.

The more awkward and less portable numeric specification of the above possible root devices in major/minor format is also accepted. (For example, /dev/sda3 is major 8, minor 3, so you could use 'root=0x803' as an alternative.)

Ref: <https://man7.org/linux/man-pages/man7/bootparam.7.html>

# 6 使用 NFS 加速开发

## 6.1 Ubuntu Server 端设置说明:

安装 nfs-kernel-server

```
sudo apt-get install nfs-kernel-server
```

建立 mount 文件夹

```
例: mkdir /home/nfs_server
```

修改/etc/exports 文件, 添加如下内容

```
/home/nfs_server *(rw,sync,no_subtree_check,no_root_squash)
```

restart nfs 服务

```
/etc/init.d/rpcbind restart  
/etc/init.d/nfs-kernel-server restart
```

## 6.2 EVB 板端 mount 说明:

在/mnt/data 文件系统内建立 mount point

```
mkdir /mnt/data/nfs
```

mount nfs

```
mount -t nfs -o nolock 192.168.1.103:/home/nfs_server /mnt/data/nfs/
```

## 6.3 注意事项:

PC 和板子连接在同一局域网。