



CV184X SDK 编译使用手册

Version: 1.1.0

Release date: 2025-09-04

©2025 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

1	声明	2
2	建构 CVITEK 软件编译环境	3
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	13
2.5	文件结构	13
2.6	编译	13
2.6.1	环境变量说明	13
2.6.2	SDK 编译	14
2.7	编译组态设定	14
2.7.1	单系统或双系统的编译设定	16
2.7.2	工具链切换	18
2.7.3	修改分区	19
2.7.4	内存映射设定	19
2.7.4.1	内存映射	19
2.7.4.2	内存映射修改	21
3	烧录说明	22
3.1	使用前准备	22
3.2	操作过程	22
3.3	操作实例	22
3.4	打包烧录器烧录镜像	24
3.5	注意事项	24
4	EVB 接口说明	25
5	根文件系统 (rootfs)	27
5.1	根文件系统简介	27
5.2	Rootfs	28
5.2.1	Pre-build rootfs 架构	28
5.2.2	编译来自 buildroot 的 rootfs	30
5.2.3	将 rootfs 包装成可烧录映像档	35
5.2.4	Linux kernel 自动加载 rootfs	36

6	使用 NFS 加速开发	37
6.1	Ubuntu Server 端设置说明:	37
6.2	EVB 板端 mount 说明:	37
6.3	注意事项:	38

修订记录

Revision	Date	Description
1.0.0	2025/02/26	Initial.
1.0.1	2025/05/30	Update boards, github download, configuration, figures
1.0.2	2025/06/26	Update Partition, Memmap Configuration
1.1.0	2025/09/04	Update configuration for dual_os and single_os

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**。

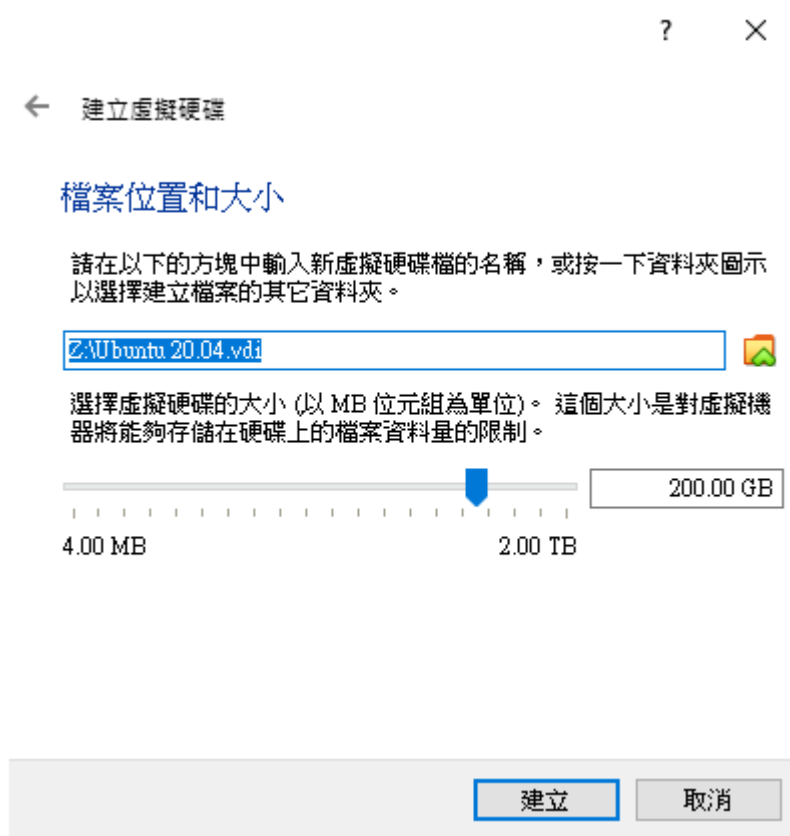
MB

4 MB 16384 MB

下一個(N)

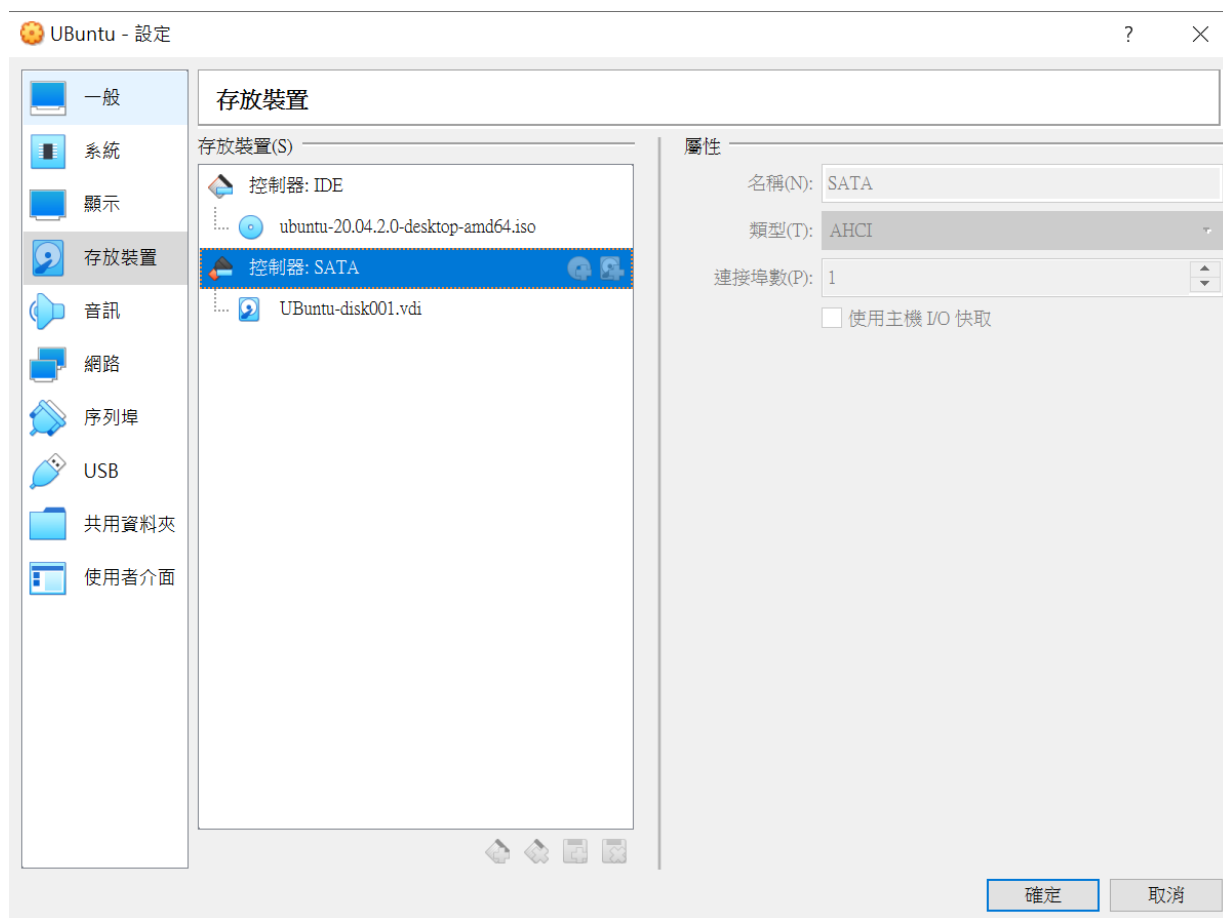
取消

- 预留 200GB 硬盘空间，供后续存放 SDK 用。



2.1.2 Ubuntu 开机设定

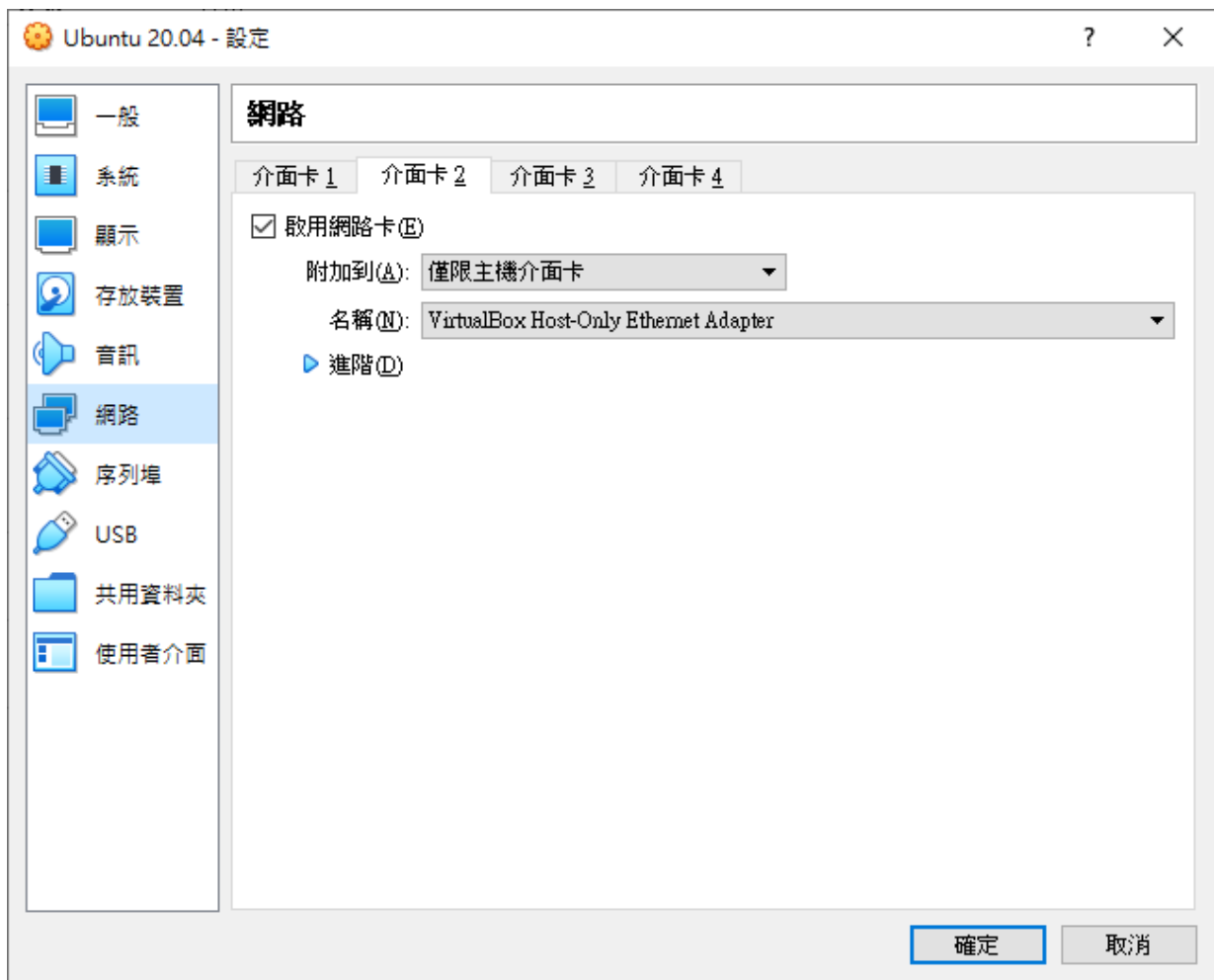
- 第一次开机需要挂载安装光盘 ISO 档案



· 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

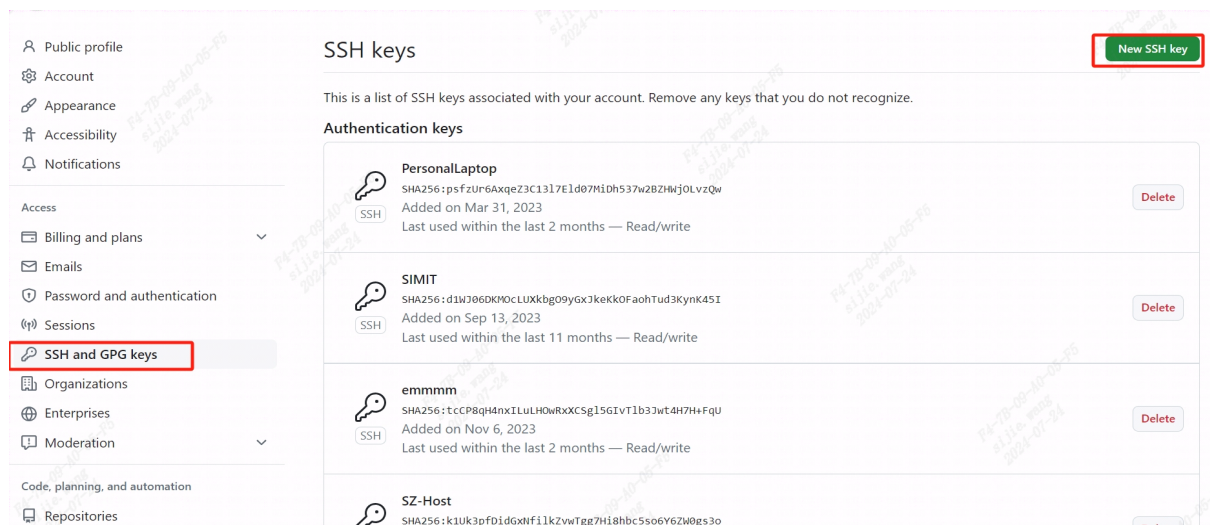
1. 设置账号邮箱

```
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
./sophpi/scripts/repo_clone.sh --gitclone sophpi/scripts/subtree_cv184x-v6.x.xml
```

2.5 文件结构

SDK 文件结构说明如下：

```
top
├── build                ## 编译脚本集合
│   └── media
│       └── SensorSupportList  ## sensor 驱动
├── fsbl                ## First Stage Boot Loader
├── host-tools          ## 编译工具链
├── isp_tuning          ## isp tuning工具
├── linux_5.10          ## linux内核
├── cvi_mpi             ## 多媒体及isp库
├── osal               ## 操作系统抽象层
├── osdrv              ## kernel space 驱动
├── oss                ## 开源代码库
├── ramdisk            ## ram disk
├── u-boot-2021.10      ## uboot
├── libsophon          ## tpu库及tpu驱动
├── cvi_rtsp           ## 多媒体hal层
├── buildroot-2021.05   ## Buildroot
├── isp-tool-daemon     ## isp调试工具
├── cvi_alios          ## AliOS
├── freertos           ## FreeRTOS
├── rt-thread          ## RT-Thread
└── tdl_sdk            ## AI Models SDK
```

2.6 编译

2.6.1 环境变量说明

编译前置动作最主要是为了设置三个环境变量：\$CHIP, \$BOARD, \$DUAL_OS

\$CHIP 变量是需要根据用户的 SOC 来做设置。

\$BOARD 变量是针对每张 EVB, 有不同的驱动，必须要正确设置。

\$DUAL_OS 变量是根据用户需要确定每张 EVB 运行单系统或双系统，每张 EVB 有默认的配置，可以通过 menuconfig 命令切换。单双系统的说明参阅下文编译单系统或双系统的设定。

2.6.2 SDK 编译

编译请在 docker 环境中执行

```
source build/envsetup_soc.sh # 初始化环境

defconfig cv1842hp_wevb_0014a_spinor # 设定编译组态, cv184x内建支持的EVB板卡配置命名方式为
→$CHIP_$BOARD

clean_all # 清除旧的编译目标文件

build_all # 全编译
```

在 source 命令初始化环境之前, 可以通过 export 命令按需设置如下环境变量等:

```
export ENABLE_BOOTLOGO=y # 配置logo.jpg打包到档案的环境变量
export TPU_REL=1 # 加入tdl_sdk SDK编译
```

编译出的档案会放置于 install/soc_\$CHIP_\$BOARD 之下。

部分编译:

注: 需要在全编译一次后才能支持部分编译

```
$ clean_uboot && build_uboot # 只更新uboot, 对应fip.bin和fip_spl.bin

$ clean_kernel && build_kernel # 只更新kernel, 对应boot.spinor

$ clean_osdrv && build_osdrv && pack_rootfs # 只更新osdrv, 生成到system/*, 对应rootfs.spinor

$ clean_middleware && build_middleware && pack_rootfs # 只更新cvi_mpi(cvi_test / sample_
→dsi), 生成到system/*, 对应rootfs.spinor

$ clean_libsophon && build_libsophon # 只更新libsophon, 生成到libsophon/build/*

$ clean_tdl_sdk && build_tdl_sdk # 只更新tdl_sdk
```

- 单独编译 cvi_mpi: build_middleware 会针对 Sensor driver (位于 cvi_mpi/component/isp/下) 以及 sample application (位于 cvi_mpi/sample/下) 重新编译
- 重新打包文件系统: pack_rootfs 会将变更后的 driver 以及 application 包装成可烧录映像档案。

2.7 编译组态设定

初始化之后, 编译组态的设定档案路径如下, 以 cv1842hp_wevb_0014a_spinor 为例

```
$ cat build/boards/cv184x/cv1842hp_wevb_0014a_spinor/cv1842hp_wevb_0014a_spinor_defconfig
CONFIG_CHIP_cv1842hp=y
CONFIG_BOARD_wevb_0014a_spinor=y
CONFIG_DDR_CFG_ddr3_2133_x16=y
CONFIG_ARCH="arm"
```

(下页继续)

(续上页)

```

CONFIG_CC_OPTIMIZE_FOR_SIZE=y
CONFIG_KERNEL_ENTRY_HACK=y
CONFIG_TOOLCHAIN_MUSL_ARM=y
CONFIG_FLASH_SIZE_SHRINK=y
CONFIG_BOOT_IMAGE_SINGLE_DTB=y
CONFIG_STORAGE_TYPE_spinor=y
.....
CONFIG_DUAL_OS=y
CONFIG_ENABLE_ALIOS=y
CONFIG_RTOS_BUILD_IN_FIP=n
# Memory Map Configuration
CONFIG_DRAM_SIZE=0x10000000
.....
#
# Partition Configuration
#
CONFIG_PARTITION_COUNT=8
.....

```

可以通过 defconfig 打印 CV184x 内建支持的 EVB 编译组态，然后指定具体的 EVB。

```

$ defconfig cv184x
* cv184x * the available cvitek EVB boards
cv184x - cv184x_fpga [FPGA]
    cv184x_palladium [PALLADIUM]
    cv184x_palladium_c906 [PALLADIUM_c906]
cv1841c - cv1841c_wevb_0015a_emmc [CA53 + EMMC + QFN SIP 128MB]
    cv1841c_wevb_0015a_spinand [CA53 + SPINAND + QFN SIP 128MB]
    cv1841c_wevb_0015a_spinor [CA53 + SPINOR + QFN SIP 128MB]
cv1842cp - cv1842cp_wevb_0015a_spinand [CA53 + SPINAND + QFN SIP 256MB]
    cv1842cp_wevb_0015a_spinor [CA53 + SPINOR + QFN SIP 256MB]
cv1842hp - cv1842hp_wevb_0014a_emmc [CA53 + EMMC + BGA SIP 256MB]
    cv1842hp_wevb_0014a_spinand [CA53 + SPINAND + BGA SIP 256MB]
    cv1842hp_wevb_0014a_spinor [CA53 + SPINOR + BGA SIP 256MB]
cv1843hp - cv1843hp_wevb_0014a_emmc [CA53 + EMMC + BGA SIP 512MB]

```

下列介绍两种方式进行编译组态设定。

- 使用 command line - setconfig 方式

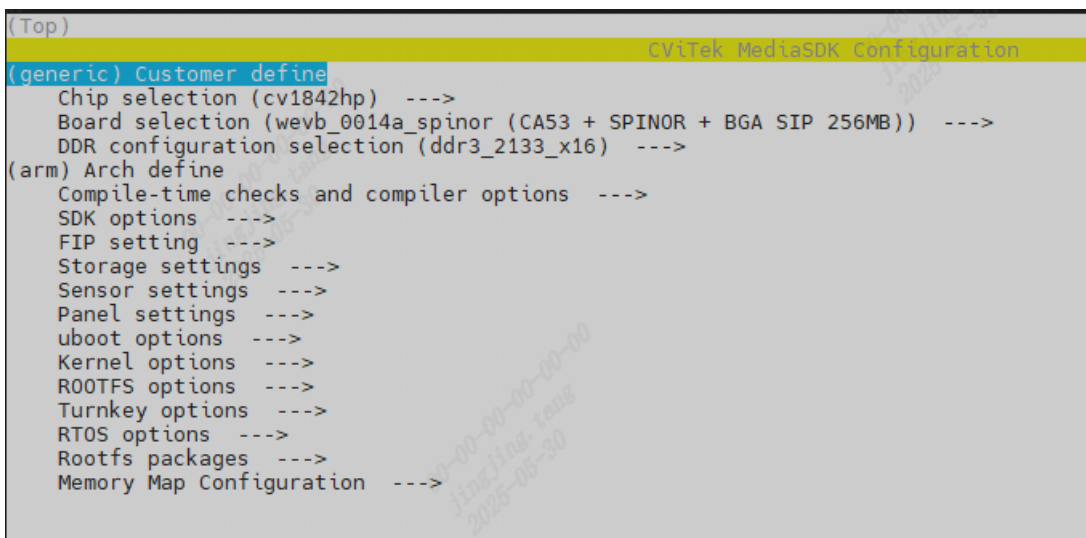
```

$ setconfig ARCH=arm
$ setconfig CROSS_COMPILE_KERNEL=arm-none-linux-musleabihf-
$ setconfig TOOLCHAIN_MUSL_ARM=y

```

- 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，DUAL_OS 等等。

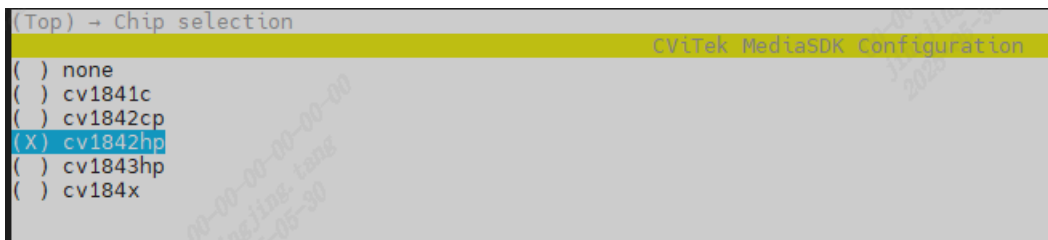


配置过程可以透过 [Enter] [Space] [ESC] 等进行设置/返回的动作

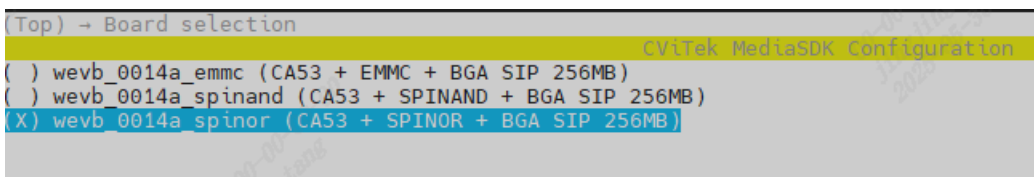
配置完毕后, 按下 [S] 储存配置文件, 接着按下 [Q] 离开图形化接口

(或者按下 ESC, 会自动弹出是否需要储存的图形)

选取 IC (以 cv1842cp 为例)



EVB 版本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。

2.7.1 单系统或双系统的编译设定

CV184X 板卡是双核处理器, 大核是基于 ARM 架构的处理器, 小核是基于 RISC-V 架构的处理器。

单双系统是根据多媒体功能需求不同划分的, 主要在小核的使用不同。- 单系统配置时, 大核上运行 Linux 系统, 小核上运行 FreeRTOS 或 RT-Thread 系统 (v6.3.0 之后默认是 RT-Thread)。- 双系统配置时, 大核上运行 Linux 系统, 小核上运行 AliOS 系统。

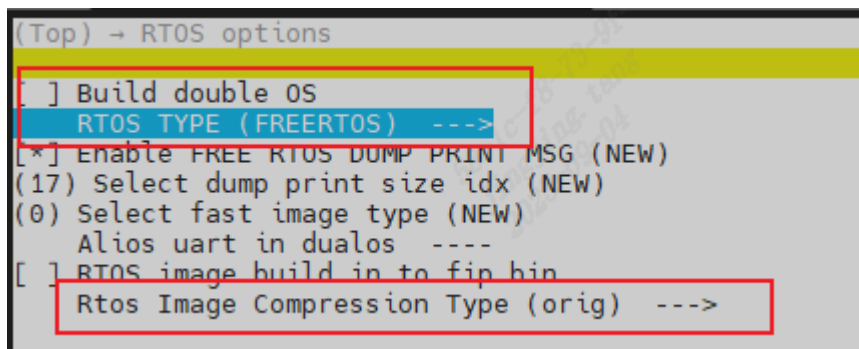
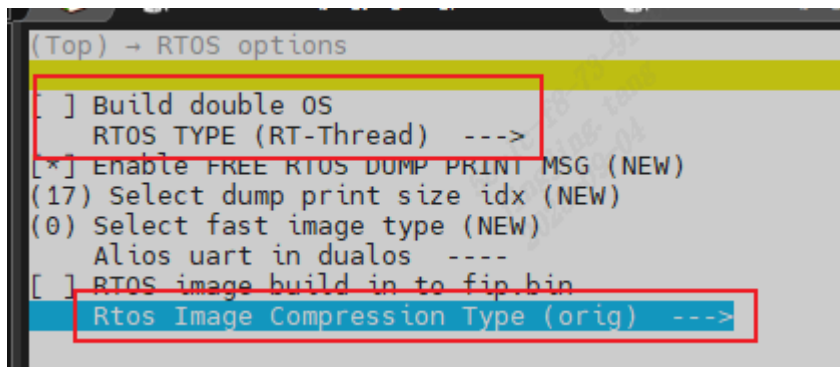
通过 setconfig 命令设定, 单系统设定为:

```
# 选择 RT-Thread 系统
$ setconfig DUAL_OS=n RTOS_COMPRESS_ORIG=y
Run setconfig function
Loaded configuration '.config'
Configuration saved to '.confi'
...
# 选择 FreeRTOS 系统
$ setconfig DUAL_OS=n ENABLE_FREERTOS=y
Run setconfig function
Loaded configuration '.config'
Configuration saved to '.confi'
...
```

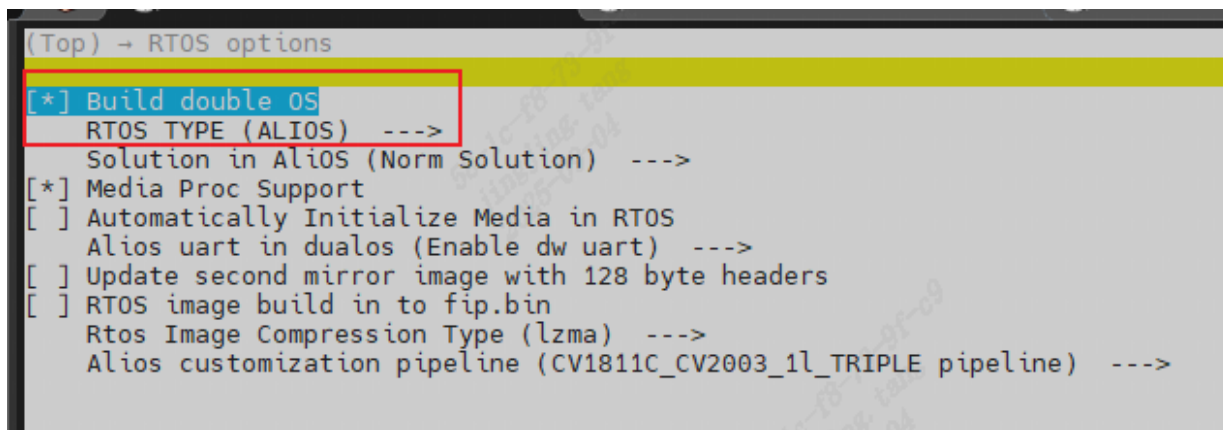
双系统设定为:

```
# 默认 AliOS 系统
$ setconfig DUAL_OS=y
Run setconfig function
Loaded configuration '.config'
Configuration saved to '.confi'
...
```

执行 menuconfig 命令在菜单中选择, 单系统设定为:



双系统设定为:



2.7.2 工具链切换

CV184x 提供了基于以下三种 Arch 的编译工具链：

Arch	Libc	Toolchain	Target Platform
Arm	glibc	arm-gnu-toolchain-11.3.rel1-x86_64-arm-none-linux-gnueabi	arm-none-linux-gnueabi
Arm	musl	arm-gnu-toolchain-11.3.rel1-x86_64-arm-none-linux-musleabi	arm-none-linux-musleabi
Aarch64	glibc	arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu	aarch64-none-linux-gnu

Linux 内核可以通过配置如下组态，从而切换不同的编译工具链：

- arm glibc

```
$ setconfig ARCH=arm
$ setconfig CROSS_COMPILE_KERNEL=arm-none-linux-gnueabi-
$ setconfig TOOLCHAIN_GLIBC_ARM=y
```

- arm musl

```
$ setconfig ARCH=arm
$ setconfig CROSS_COMPILE_KERNEL=arm-none-linux-musleabi-
$ setconfig TOOLCHAIN_MUSL_ARM=y
```

- aarch64 glibc

```
$ setconfig ARCH=aarch64
$ setconfig CROSS_COMPILE_KERNEL=aarch64-none-linux-gnu-
$ setconfig TOOLCHAIN_GLIBC_ARM64=y
```

以上均可使用图像化界面配置。

U-boot 使用 aarch64-none-linux-gnu-工具链。

AliOS 使用玄铁交叉编译工具链 Xuantie-900-gcc-elf-newlib-x86_64-V2.6.1

2.7.3 修改分区

请参阅 CV184x Flash 分区工具使用手册 - 第六章 分区修改

2.7.4 内存映射设定

2.7.4.1 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
- RTOS_SYS_TOTAL: 小核使用内存, 如果使用 alios 作为小核系统, RTOS_SYS_TOTAL_SIZE= 大小核的共享内存 +RTOS_SYS_SIZE; 使用 freertos, 则 RTOS_SYS_TOTAL_SIZE=RTOS_SYS_SIZE。KERNEL: KERNEL 系统内存, KERNEL_ENTRY_HACK
- ION: 大核 ION 驱动使用的内存
- RTOS_ION: 小核 ION 使用的内存

			KERNEL_MEMORY_ADDR = DRAM_BASE KERNEL_MEMORY_SIZE = DRAM_SIZE		
	DRAM_BASE	MONITOR	ATF:640K		ATF区域
	FSBL_C906L_START	RTOS_SYS_TOTAL	RTOS_SYS:4M	(text, data, bss, heap, stack)	小核系统区域
	FSBL_C906L_START + RTOS_SYS_SIZE		RTOS_LOG:128k	only extern in alios (0xe0000)	alios与kernel公共内存
			SHARE_MEM:128k		
			SHARE_PARAM:64k		
			SHARE_PARAM_BAK:64k		
	RTOS_LOGO_ADDR=FSBL_C906L_START + RTOS_SYS_TOTAL_SIZE		PQBIN:1024k		kernel内存
			RTOS_LOGO		
	KERNEL_ENTRY_HACK 双系统:0x80708000 单系统:0x80108000	KERNEL			临时使用内存 系统启动后被kernel接管
			SPL_FDT:1M	CVI_MMC_SKIP_TUNING:1K	
			CVI_UPDATE_HEADER:1K		uboot临时使用 系统启动后被kernel接管
	UIIMAGE_ADDR		UIIMAGE\FSBL_UNZIP:4M		
	UIIMAGE_ADDR + UIIMAGE_SIZE				
	BLCP_2ND_COMP_ADDR = 0x81ea0000	KERNEL	RTOS_COMPRESS_BIN		大核ION内存
					小核ION内存
	loader_2nd_header->runaddr		CONFIG_SYS_INIT_SP:3M		
			CONFIG_SYS_TEXT	uboot.bin	
	SYS_TEXT_BASE=0x83800000	ION		Uboot relocat_addr_SP	
				Uboot relocat_addr_TEXT	
				H26X_BITSTREAM:0K	
				H26X_ENC_BUFF:0K	
	RTOS_ION_ADDR - ION_SIZE	ION	ION_SIZE:user-defined	ISP_MEM:0K	
				BOOTLOGO:900k	
	RTOS_ION_ADDR				
		RTOS ION	RTOS_ION_SIZE:user-defined		
	DRAM_BASE + DRAM_SIZE			RTOS_ION_ADDR + RTOS_ION_SIZE	

使用 python 脚本 (memmap.py) 定义具体的 ADDR 和 SIZE, 在编译时解析 memmap.py 生成对应的 cvi_board_memmap.* 文件提供给 linux、u-boot、fsbl 等编译使用。

```
$ cat build/boards/default/memmap/memmap.py
import os
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 = 510 * SIZE_1M
.....
```

· cvi_board_memmap.h

```
$ cat build/output/cv1842hp_wevb_0014a_spinor/cvi_board_memmap.h
#ifndef __BOARD_MMAP__a149c034__
#define __BOARD_MMAP__a149c034__
#define CONFIG_SYS_TEXT_BASE 0x83800000 /* offset 56.0MiB */
.....
```

· cvi_board_memmap.conf

```
$ cat build/output/cv1842hp_wevb_0014a_spinor/cvi_board_memmap.conf
CONFIG_SYS_TEXT_BASE=0x83800000
CVIMMAP_ATF_SIZE=0xa0000
CVIMMAP_BOOTLOGO_ADDR=0x89e3e000
CVIMMAP_BOOTLOGO_SIZE=0x1c2000
.....
```

· cvi_board_memmap.ld

```
$ cat build/output/cv1842hp_wevb_0014a_spinor/cvi_board_memmap.ld
CONFIG_SYS_TEXT_BASE = 0x83800000;
CVIMMAP_ATF_SIZE = 0xa0000;
CVIMMAP_BOOTLOGO_ADDR = 0x89e3e000;
CVIMMAP_BOOTLOGO_SIZE = 0x1c2000;
CVIMMAP_CONFIG_SYS_INIT_SP_ADDR = 0x83500000;
.....
```

· cvi_board_memmap.txt

```
$ cat build/output/cv1842hp_wevb_0014a_spinor/cvi_board_memmap.txt
FBSL stage:
| Name          | Start Address | End Address | Size      | Size(M/K/B) |
|-----|-----|-----|-----|-----|
.....
```

不同 EVB 的内存分配情况略有不同, 主要单双系统在小核上的使用不同。

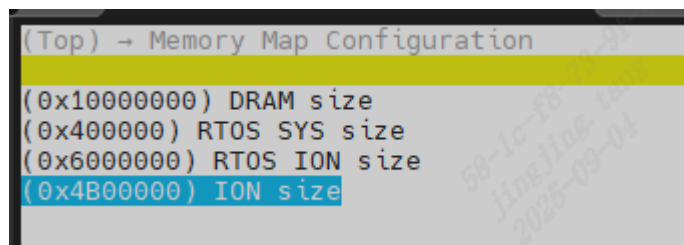
因此, 将主要的 ADDR 和 SIZE 定义成可设置的变量, 默认值定义在 EVB defconfig 文件中 (v6.3.0 前后有所不同, 请详见代码)。在生成 cvi_board_memmap.* 文件时, 读取配置文件中的

值。

```
$ cat build/boards/cv184x/cv1842hp_wevb_0014a_spinor/cv1842hp_wevb_0014a_spinor_defconfig
.....
CONFIG_DUAL_OS=y
CONFIG_ENABLE_ALIOS=y
CONFIG_RTOS_BUILD_IN_FIP=n
CONFIG_DRAM_SIZE=0x10000000
CONFIG_ION_SIZE=0x4B00000
CONFIG_RTOS_SYS_SIZE=0x400000
CONFIG_RTOS_ION_SIZE=0x6000000
.....
```

2.7.4.2 内存映射修改

通过 menuconfig 图形化菜单界面可以修改部分 ADDR 和 SIZE，以满足开发的需求。



3 烧录说明

3.1 使用前准备

- 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，文件路径在 `install/soc_${CHIP}_${BOARD}/` 下，将目录下 `upgrade.zip` 解压出来的内容和 `ramboot.itb` 拷贝到 SD 卡中。

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 CV184X 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。

```
## Resetting to default environment
Start SD downloading...
mmc1 : finished tuning, code:61
switch to partitions #0, OK
mmc0 is current device
361984 bytes read in 14 ms (24.7 MiB/s)
SF: Detected EN25QX128A with page size 256 Bytes, erase size 4 KiB, total 16 MiB
.....
Header Version:1
2501256 bytes read in 29 ms (82.3 MiB/s)
device 0 offset 0x300000, size 0x262a48
2501192 bytes written, 0 bytes skipped in 22.889s, speed 111877 B/s
sf update speed 0.109 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: 2
OK
cv184x#
```

将平台断电，拔出 SD 卡，再重开机，可以看到如下信息，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）

```
Starting kernel ...

...

Starting app...
Starting klogd: OK
Starting syslogd: OK

Welcome to CviLinux.
[root@cvitek]~#
```

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 `fiip_spl.bin` 特殊处理,`fiip.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 `fiip.bin`) 两个文件

在其他 shell 脚本中调用示例：

```
source build/envsetup_soc.sh
defconfig cv1842cp_sm3_81_spinand
pack_prog_img 0x95c8
```

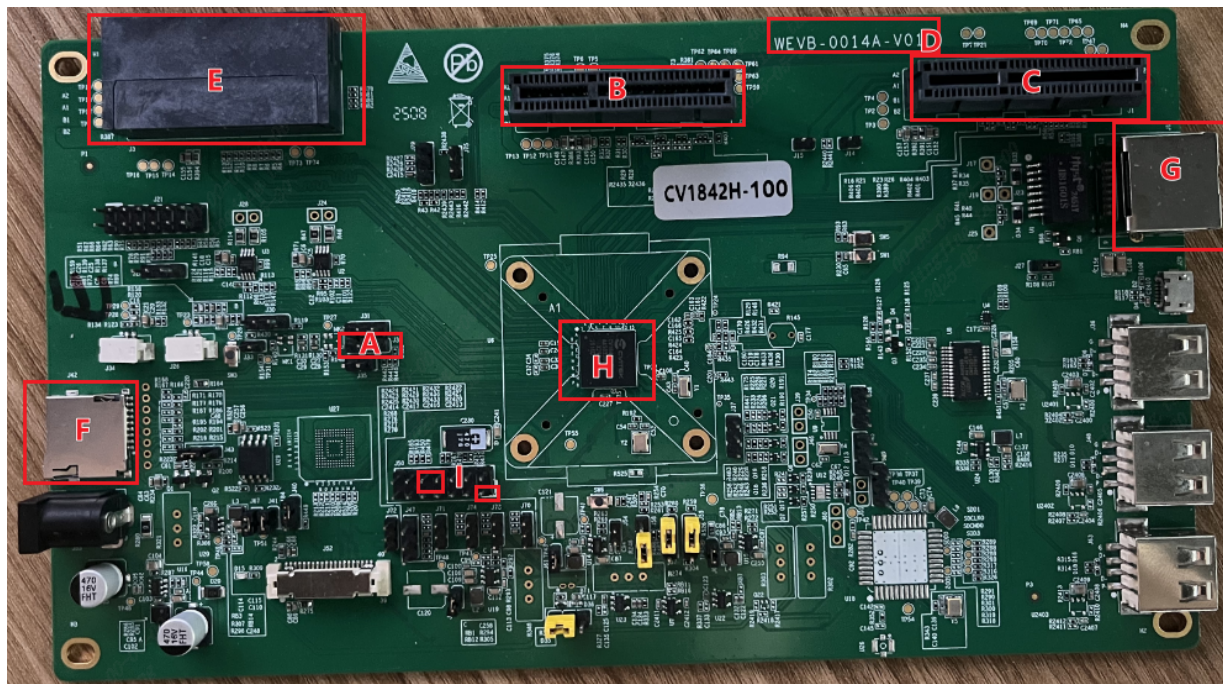
3.5 注意事项

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 CV184x BGA EVB

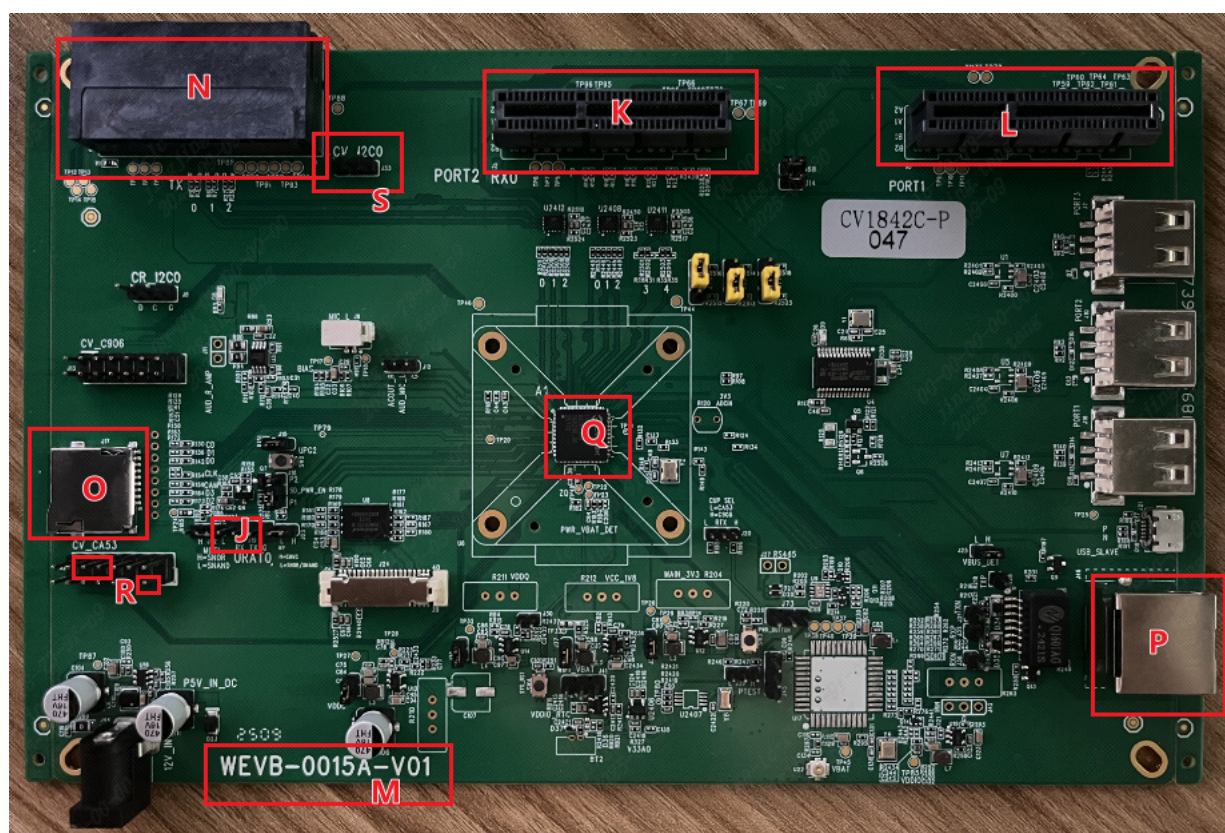
- A. UART Debug Port
- B. Sensor 0 EVB 插槽
- C. Sensor 1 EVB 插槽
- D. EVB 版本号
- E. Panel EVB 插槽
- F. SD 卡插槽
- G. Ethernet 0 连接口
- H. 主 IC CV184x
- I. UART1 Debug Port



下方图片为 CV184x QFN EVB

- J. UART Debug Port

- K. Sensor 0 EVB 插槽
- L. Sensor 1 EVB 插槽
- M. EVB 版本号
- N. Panel EVB 插槽
- O. SD 卡插槽
- P. Ethernet 0 连接口
- Q. 主 IC CV184x
- R. UART1 Debug Port (sdk v6.3.0 及其之前版本)
- S. UART1 Debug Port (sdk v6.3.1 及其之后版本)



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三种 Arch 产生之 pre-build rootfs 档案

Arch	Libc	Pre-build ramdisk path
Arm	glibc	<code>ramdisk/rootfs/common_arm/</code>
Arm	musl	<code>ramdisk/rootfs/common_musl_arm/</code>
Aarch64	glibc	<code>ramdisk/rootfs/common_arm64/</code>

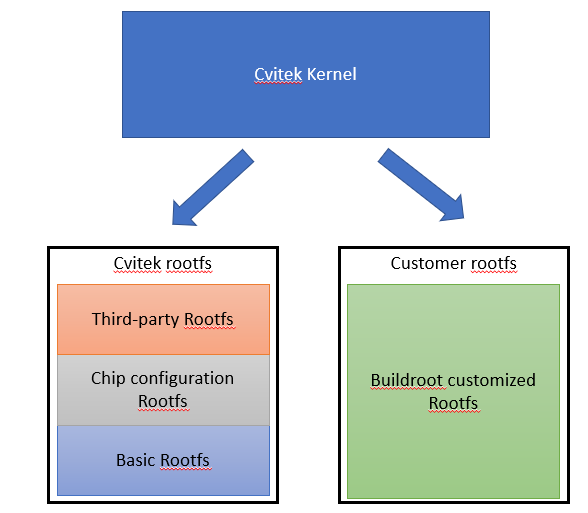
- **Chip configuration rootfs:**

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/`



可以使用 `menuconfig` 命令通过图形化菜单的方式决定需要那些 Third-party software 要放置进 Rootfs

```

(Top)
CViTek MediaSDK Configuration
(generic) Customer define
  Chip selection (cv1842hp) --->
  Board selection (wevb_0014a_spinor (CA53 + SPINOR + BGA SIP 256MB)) --->
  DDR configuration selection (ddr3_2133_x16) --->
(arm) Arch define
  Compile-time checks and compiler options --->
  SDK options --->
  FIP setting --->
  Storage settings --->
  Sensor settings --->
  Panel settings --->
  uboot options --->
  Kernel options --->
  ROOTFS options --->
  Turnkey options --->
  RTOS options --->
  Rootfs packages --->
  Memory Map Configuration --->

```

```

(Top) → Rootfs packages
CViTek MediaSDK Configuration
[ ] Target addb
[ ] Target package of AP6201BM fw files
[ ] Target package bluetooth
[ ] Target package cvitracer
[*] Target package dropbear
[*] Target package gdbserver
[ ] Target package libbtrace
[*] Target package libcrypto
[ ] Target package libcurl
[ ] Target package libevent
[ ] Target package libiperf
[ ] Target package libiw
[ ] Target package libprotobuf
[*] Target package libz
[*] Target package mtd-utils
[ ] Target package nanomsg
[ ] Target package openssl
[*] Target package ota server
[ ] Target package parted
[ ] Target package procrank
[ ] Target package procps
[ ] Target package python3.7
[ ] Target package rsyslog
[ ] Target package secure_image
[ ] Target package wifi
[*] Target package busybox-syslogd script
[ ] Target package mt7603u conf files
[ ] Target package libtirpc
[ ] Target package libnfs
[ ] Target package bash
[*] Target package crontabs
[ ] Target package iperf3 tools
[ ] Target package e2fsprogs
[ ] Target package gattord
[ ] Target package stress-ng
[ ] Target package htop
|||||
[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。

从 CV184x v6.2.0 版本开始，可以基于 CV184x 的编译环境，使用 Buildroot 编译 rootfs。

1. 拉取 buildroot 仓库，与 CV184x build 目录同级，并切换到 buildroot-2023.11 分支

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

2. 编译

```
$ source build/envsetup_soc.sh # 初始化环境

$ defconfig cv1842hp_wevb_0014a_spinor #
→ 设定编译组态, cv184x内建支持的EVB板卡配置命名方式为 $CHIP_$BOARD

$ setconfig BUILDROOT_FS=y # 设定使用Buildroot 编译rootfs

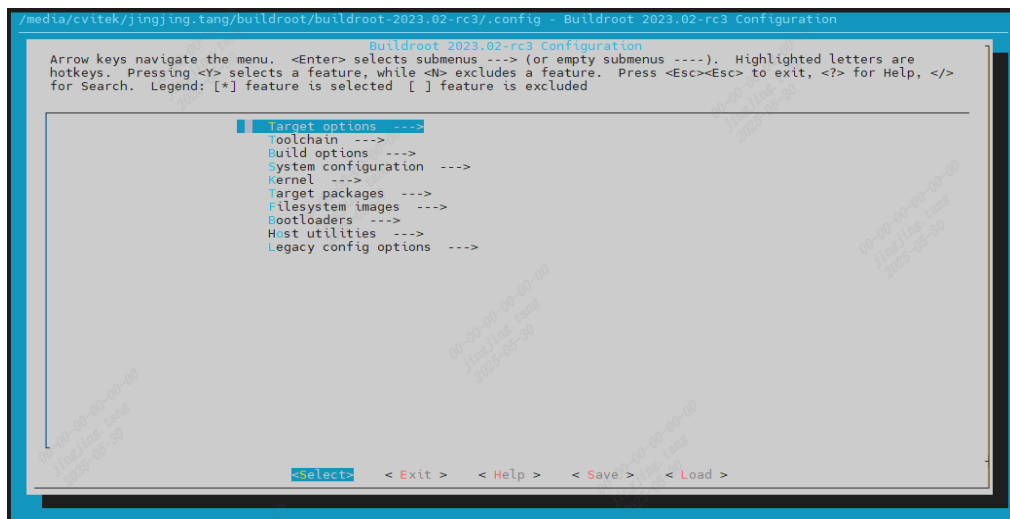
$ cp buildroot-2021.05/configs/cvitek_CV184X_musl_defconfig buildroot-2021.05/.config #
→ 设定Buildroot编译配置项

$ clean_all && build_all # 全编译

$ clean_rootfs && pack_rootfs # 仅更新rootfs
```

3. 图形化菜单界面配置 Buildroot

```
$ menuconfig_buildroot
```



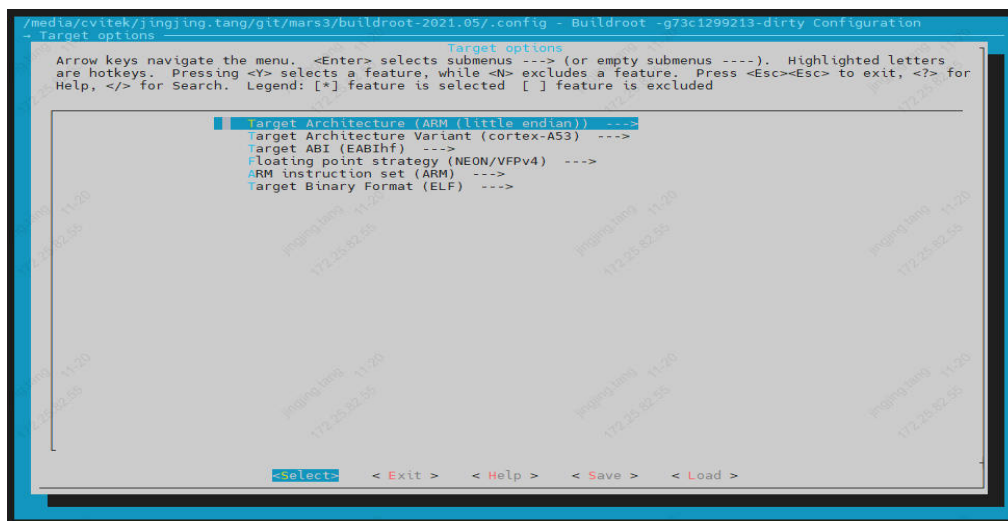
4. 设置 Arch Info & Toolchain & Packages

从 CV184x v6.2.0 版本开始，可以基于 CV184x 的编译环境，使用 Buildroot 编译 rootfs。configs 目录下有 CV184x 的默认配置文件 cvitek_CV184X_musl_defconfig。

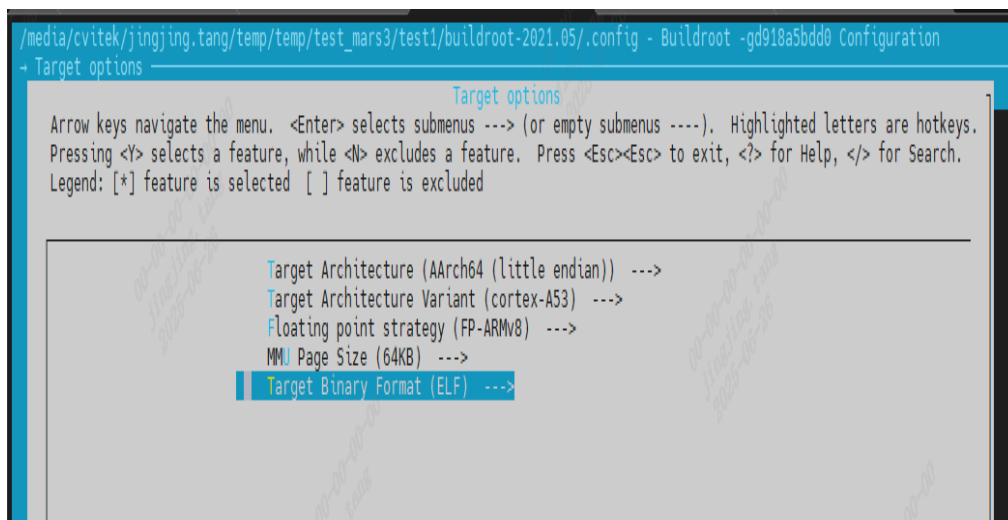
初始化编译环境和选定 EVB 后，可以通过 menuconfig_buildroot 命令快速配置

设置架构

ARM

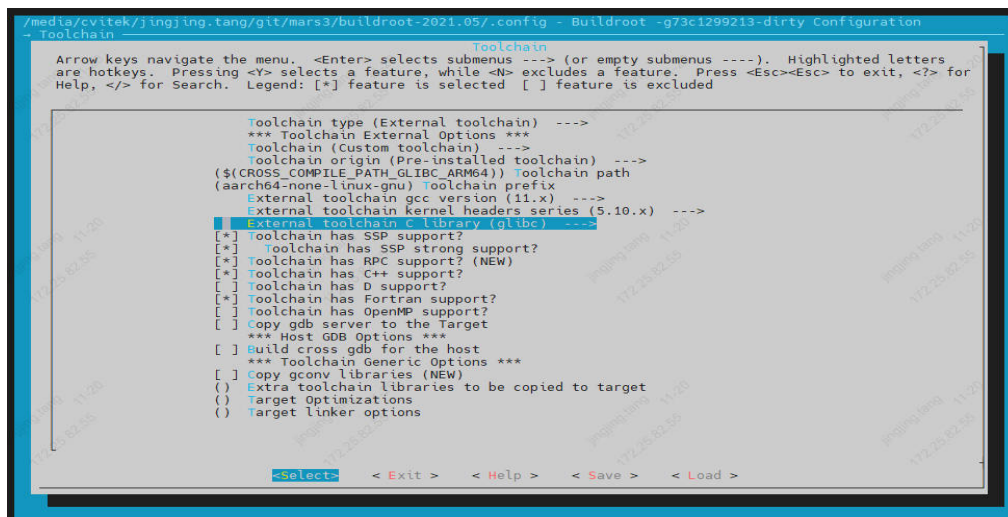


AARCH64



设置工具链

GLIBC AARCH64



GLIBC ARM

```
/media/cvitek/jingjing.tang/git/mars3/buildroot-2021.05/.config - Buildroot -g73c1299213-dirty Configuration
- Toolchain -
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_GLIBC_ARM}) Toolchain path
(arm-none-linux-gnueabi) Toolchain prefix
External toolchain gcc version (11.x) --->
External toolchain kernel headers series (5.10.x) --->
[*] External toolchain C library (glibc) --->
[*] Toolchain has SSP support?
[*] Toolchain has SSP strong support?
[*] Toolchain has RPC support? (NEW)
[*] 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
*** Toolchain Generic Options ***
[ ] Copy gconv libraries (NEW)
[ ] Extra toolchain libraries to be copied to target
[ ] Target Optimizations
[ ] Target linker options

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

MUSL ARM

```
/media/cvitek/jingjing.tang/git/mars3/buildroot-2021.05/.config - Buildroot -g73c1299213-dirty Configuration
- Toolchain -
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_ARM}) Toolchain path
(arm-none-linux-musleabi) Toolchain prefix
External toolchain gcc version (11.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
*** Host GDB Options ***
[ ] Build cross gdb for the host
*** Toolchain Generic Options ***
[ ] Extra toolchain libraries to be copied to target
[ ] Target Optimizations
[ ] Target linker options

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

MUSL AARCH64

```
/media/cvitek/jingjing.tang/git/mars3/buildroot-2021.05/.config - Buildroot -g73c1299213-dirty Configuration
- Toolchain -
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_ARM64}) Toolchain path
(aarch64-none-linux-musl) Toolchain prefix
External toolchain gcc version (11.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
*** Host GDB Options ***
[ ] Build cross gdb for the host
*** Toolchain Generic Options ***
[ ] Extra toolchain libraries to be copied to target
[ ] Target Optimizations
[ ] Target linker options

<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 submenu ---). 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>

```

5. 产生新的 ROOTFS 可烧录映像档

```
$ pack_rootfs
```

buildroot 编译 rootfs 参考 build/Makefile 之 br-rootfs-prepare 和 br-rootfs-pack:

```

menuconfig-br2:
    ${Q}${MAKE} -C ${BUILDDROOT_PATH} menuconfig

savedefconfig-br2:
    ${Q}${MAKE} -C ${BUILDDROOT_PATH} savedefconfig

# BR_OVERLAY_DIR
# BR_ROOTFS_RAWIMAGE
br-rootfs-prepare:export CROSS_COMPILE_KERNEL=$(patsubst "%",%,$(CONFIG_CROSS_
→COMPILE_KERNEL))
br-rootfs-prepare:export CROSS_COMPILE_SDK=$(patsubst "%",%,$(CONFIG_CROSS_
→COMPILE_SDK))
br-rootfs-prepare:
    $(call print_target)
    # copy ko and mmf libs
    ${Q}mkdir -p $(BR_OVERLAY_DIR)/system
    ${Q}cp -arf $(OUTPUT_DIR)/rootfs/system/* $(BR_OVERLAY_DIR)/system/
    # copy usr/share/fw_vcodec
    # ${Q}mkdir -p $(BR_OVERLAY_DIR)/usr/share
    # ${Q}cp -rf $(RAMDISK_PATH)/rootfs/$(ROOTFS_BASE)/usr/share/fw_vcodec $(BR_
→OVERLAY_DIR)/usr/share
    # strip
    ${Q}find $(BR_OVERLAY_DIR) -name "*.ko" -type f -printf 'stripping %p\n' -exec $(CROSS_
→COMPILE_KERNEL)strip --strip-unneeded {} \;
    ${Q}find $(BR_OVERLAY_DIR) -name "*.so*" -type f -printf 'stripping %p\n' -exec $(CROSS_
→COMPILE_KERNEL)strip --strip-all {} \;
    ${Q}find $(BR_OVERLAY_DIR) -executable -type f ! -name "*.sh" ! -path "*etc*" ! -path "*.ko
→" -printf 'stripping %p\n' -exec $(CROSS_COMPILE_SDK)strip --strip-all {} 2>/dev/null \;

br-rootfs-pack:export TARGET_OUTPUT_DIR=$(BR_DIR)/output/$(BR_BOARD)
br-rootfs-pack:
    $(call print_target)
    ${Q}${MAKE} -C $(BR_DIR) $(BR_DEFCONFIG)

```

(下页继续)

(续上页)

```

    ${Q}${MAKE} -j${NPROC} -C $(BR_DIR)
    # ${Q}rm -rf $(BR_ROOTFS_DIR)/*
    # copy rootfs to rawimg dir
ifeq (${CONFIG_ROOTFS_RW},y)
ifeq ($(STORAGE_TYPE),spinor)
    $(warning spi nor flash is not support rw filesystem)
else ifeq (${STORAGE_TYPE},spinand)
    ${Q}cp $(BR_DIR)/output/images/rootfs.ubifs $(OUTPUT_DIR)/rawimages/
    ${Q}python3 $(COMMON_TOOLS_PATH)/spinand_tool/mkubiimg.py --ubionly $(FLASH_
→PARTITION_XML) ROOTFS $(OUTPUT_DIR)/rawimages/rootfs.ubifs $(OUTPUT_DIR)/
→rawimages/rootfs.spinand -b $(CONFIG_NANDFLASH_BLOCKSIZE) -p $(CONFIG_
→NANDFLASH_PAGESIZE)
    ${Q}rm $(OUTPUT_DIR)/rawimages/rootfs.ubifs
else
    ${Q}cp $(BR_DIR)/output/images/rootfs.ext4 $(OUTPUT_DIR)/rawimages/rootfs.
→$(STORAGE_TYPE)
endif
else
ifeq (${STORAGE_TYPE},spinand)
    ${Q}cp $(BR_DIR)/output/images/rootfs.squashfs $(OUTPUT_DIR)/rawimages/
    ${Q}python3 $(COMMON_TOOLS_PATH)/spinand_tool/mkubiimg.py --ubionly $(FLASH_
→PARTITION_XML) ROOTFS $(OUTPUT_DIR)/rawimages/rootfs.squashfs $(OUTPUT_DIR)/
→rawimages/rootfs.spinand -b $(CONFIG_NANDFLASH_BLOCKSIZE) -p $(CONFIG_
→NANDFLASH_PAGESIZE)
    ${Q}rm $(OUTPUT_DIR)/rawimages/rootfs.squashfs
else
    ${Q}cp $(BR_DIR)/output/images/rootfs.squashfs $(OUTPUT_DIR)/rawimages/rootfs.
→$(STORAGE_TYPE)
endif
endif
    $(call raw2cimg ,rootfs.$(STORAGE_TYPE))

# TODO A/B boot is currently not supported when CONFIG_BUILDROOT_FS is enabled
ifeq ($(CONFIG_BUILDROOT_FS),y)
rootfs:br-rootfs-prepare
rootfs:br-rootfs-pack
else
rootfs:rootfs-pack
rootfs:
    $(call print_target)
    $(call raw2cimg ,rootfs.$(STORAGE_TYPE))
endif

```

6. 透过第三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:export OSDRV_BUILD_IN:=$(CONFIG_OSDRV_BUILD_IN)
rootfs-pack:$(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'
    ${Q}${BUILD_PATH}/boards/${CHIP_ARCH_L}/${PROJECT_FULLNAME}/rootfs_
→script/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_SDK)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 (${CONFIG_ROOTFS_RW},y)
    $(call pack_image,rootfs,$(ROOTFS_DIR),71M)
else

ifeq (${STORAGE_TYPE},spinor)
ifeq (${CONFIG_ROOTFS_FORMAT_OPTIMIZATION},y)
    ${Q}mksquashfs $(ROOTFS_DIR) $(OUTPUT_DIR)/rawimages/rootfs.sqsh -root-owned -
→comp gzip
else
    ${Q}mksquashfs $(ROOTFS_DIR) $(OUTPUT_DIR)/rawimages/rootfs.sqsh -root-owned -
→comp xz
endif
else
    ${Q}mksquashfs $(ROOTFS_DIR) $(OUTPUT_DIR)/rawimages/rootfs.sqsh -root-owned -
→comp xz -e mnt/cfg/*
endif
ifeq (${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
```

(下页继续)

(续上页)

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。