软件开发指南
ECK10-13xA 核心板

成都亿佰特电子科技有限公司
Chengdu Ebyte Electronic Technology Co.,Ltd.

目录

免责申明和版权公告

本文中的信息，如有变更，恕不另行通知。 文档“按现状”提供，不负任何担保责任，包括对适销性、适用于特定用途或非侵权性的任何担保，和任何提案、规格或样品在他处提到的任何担保。本文档不负任何责任，包括使用本文档内信息产生的侵犯任何专利权行为的责任。本文档在此未以禁止反 言或其他方式授予任何知识产权使用许可，不管是明示许可还是暗示许可。

文中所得测试数据均为亿佰特实验室测试所得，实际结果可能略有差异。

文中提到的所有商标名称、商标和注册商标均属其各自所有者的财产，特此声明。

最终解释权归成都亿佰特电子科技有限公司所有。

注 意 ：

由于产品版本升级或其他原因，本手册内容有可能变更。亿佰特电子科技有限公司保留在没有任何通知或者提示的情况下对本手册的内容进行修改的权利。本手册仅作为使用指导，成都亿佰特电子科技有限公司尽全力在本手册中提供准确的信息，但是成都亿佰特电子科技有限公司并不确保手册内容完全没有错误，本手册中的所有陈述、信息和建议也不构成任何明示或暗示的担保。

1. 概述

本文主要介绍基于亿佰特核心板定制一个完整的嵌入式 Linux 系统的完整流程，其中包括开发环境的准备，代码的获取，以及如何进行 Bootloader, Kernel 的移植，定制适合自身应用需求的 Rootfs 等。我们首先介绍如何基于我们提供的源代码构建适用于 ECB10-TB13X 开发板的系统镜像，如何将构建好的镜像烧录到开发板。针对那些基于 ECK10-135A5M1G-I 核心板进行项目开发的用户，我们重点介绍了将这一套系统移植到用户的硬件平台上的方法和一些要点，并通过一些实际的移植案例和 Rootfs 定制的案例，使用户能够迅速定制适合自己硬件的系统镜像。

1.1. 软件资源

ECB10-TB13X 搭载基于 Linux 6.1.28 版本内核的操作系统，开发板出厂附带嵌入式 Linux 系统开发所需要的交叉编译工具链，TF-A 源代码，Optee-os 源码，U-boot 源代码，Linux 内核和各驱动模块的源代码，以及适用于 Windows 桌面环境和 Linux 桌面环境的各种开发调试工具，应用开发样例等。

2. 开发环境准备

本节将介绍如何搭建适用于 STM32MP1XX 系列处理器平台的开发环境，通过阅读本章节，您将了解相关硬件工具，软件开发调试工具的安装和使用。并能快速的搭建相关开发环境，为后面的开发和调试做准备。

主机硬件：i7-12700 和 VMware 虚拟机，可供参考。

操作系统：ubuntu18.04 64bit，在亿佰特提供的镜像中，路径为： 03_Tools→ubuntu-18.04.5-desktop-amd64.iso。

2.1. 开启虚拟机的 tftp 服务

sudo apt-get install vsftpd

等待软件自动安装，安装完成以后使用如下 VI 命令打开/etc/vsftpd.conf，命令如下：

sudo vi /etc/vsftpd.conf

打开以后 vsftpd.conf 文件以后找到如下两行：

local_enable=YES

write_enable=YES

确保上面两行前面没有“#”，有的话就取消掉，完成以后如图所示：

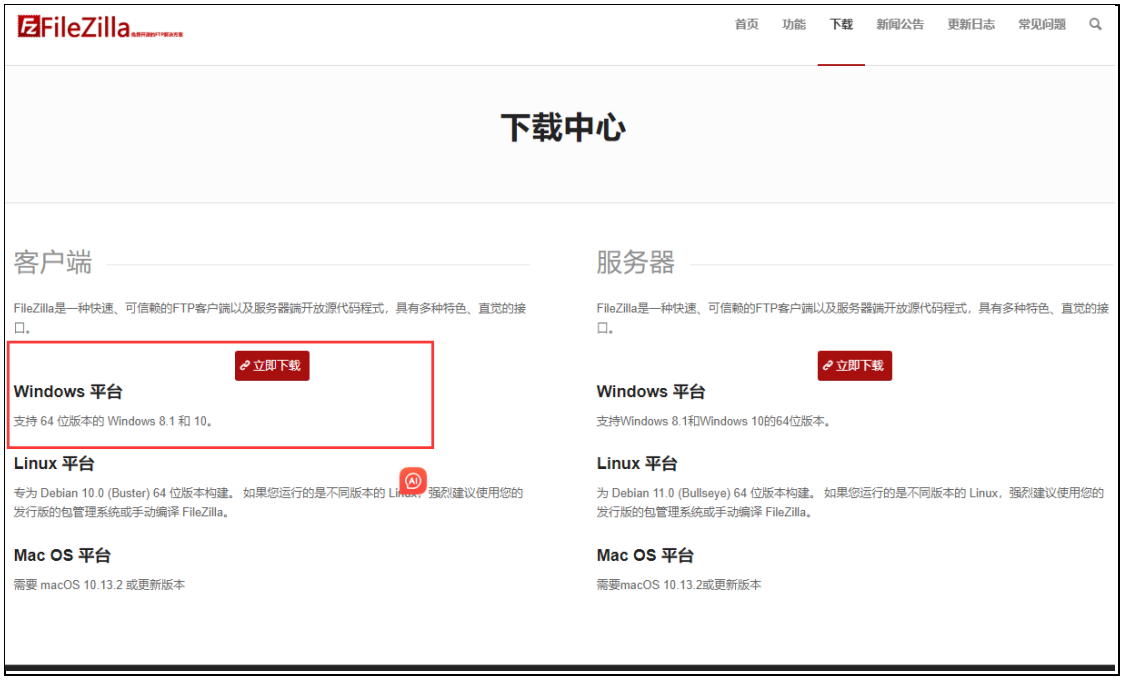
```
27 # Uncomment this to allow local users to log in.
28 local_enable=YES
29 #
30 # Uncomment this to enable any form of FTP write command.
31 write_enable=YES
```

修改完 vsftpd.conf 以后保存退出，使用如下命令重启 FTP 服务：

sudo /etc/init.d/vsftpd restart

2.2. Windows 下 FTP 客户端安装

Windows 下 FTP 客户端我们使用 FileZilla，这是个免费的 FTP 客户端软件，可以在 FileZilla 官网下载，下载地址如下：https://www.filezilla.cn/download，下载界面如图 4.1.2 所示：

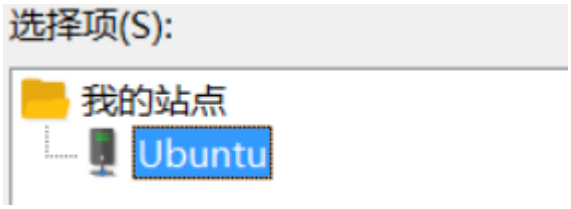


我们已经下载好了 64 位版本的 FileZilla 并放到开发板中了，路径为:03_Tools->FileZilla_3.51.0_win64-setup.exe，双击安装即可。安装完成以后找到安装目录，找到图标，然后发送图标快捷方式到桌面，完成以后如图所示：

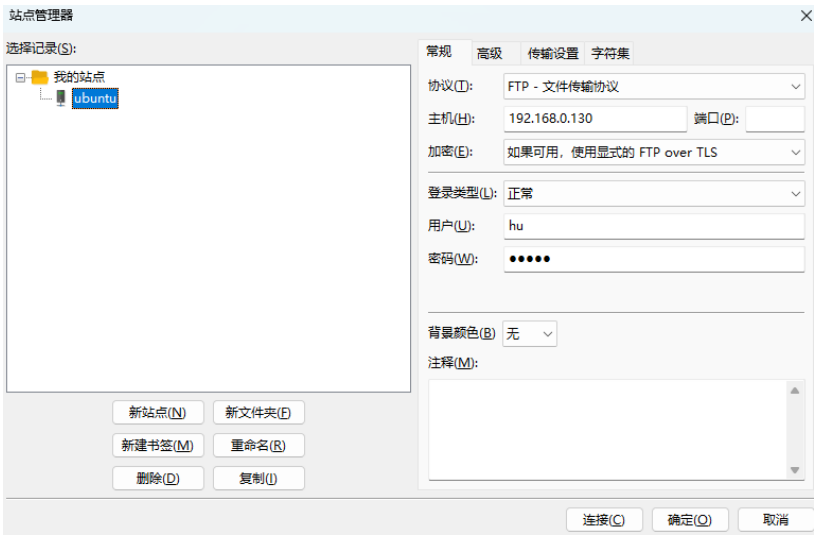
2.3. Filezilla 软件设置

Ubuntu 作为 FTP 服务器， FileZilla 作为 FTP 客户端，客户端肯定要连接到服务器上，打开站点管理器，点击：文件->站点管理器，打开以后如图所示：

点击上图中的“新站点(N)”按钮来创建站点，新建站点以后就会在“我的站点”下出现新建的这个站点，站点的名称可以自行修改，比如我将新的站点命名为“Ubuntu”，如图所示。

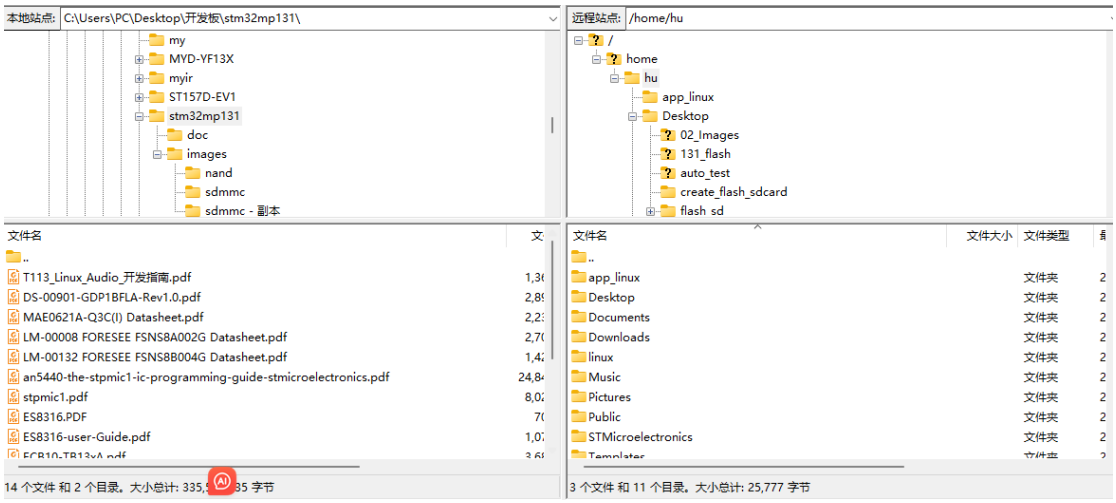


选中新创建的“Ubuntu”站点，然后对站点的“常规”选项进行设置，设置如图所示



连接成功以后如图所示，其中左边就是 Windows 文件目录，右边是 Ubuntu 文件目录，

默认进入用户家目录下。



2.4. Ubuntu 下 NFS 和 SSH 服务开启

后面进行从网络启动需要用 NFS 启动，因此要先安装并开启 Ubuntu 中的 NFS 服务，使用如下命令安装 NFS 服务：

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

等待安装完成，安装完成以后在用户根目录下创建一个名为“linux”的文件夹，以后所有的东西都放到这个“linux”文件夹里面，在“linux”文件夹里面新建一个名为“nfs”的文件夹，如图所示：

创建的 nfs 文件夹供 nfs 服务器使用，以后我们可以在开发板上通过网络文件系统来访问 nfs 文件夹，要先配置 nfs，使用如下命令打开 nfs 配置文件/etc/exports：

```
sudo vi /etc/exports
```

打开/etc/exports 以后在后面添加如下所示内容：

```
File  Edit  View  Search  Terminal  Help
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subt
ree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#
#/home/hu/linux/ubuntu *(rw,sync,no_root_squash)
/home/hu/linux/nfs *(rw,sync,no_root_squash)
~
~
~
~
```

重启 NFS 服务，使用命令如下：

sudo /etc/init.d/nfs-kernel-server restart

2.5. SSH 服务开启

开启 Ubuntu 的 SSH 服务以后我们就可以在 Windwos 下使用终端软件登陆到 Ubuntu，比如使用 MobaXterm， Ubuntu 下使用如下命令开启 SSH 服务：

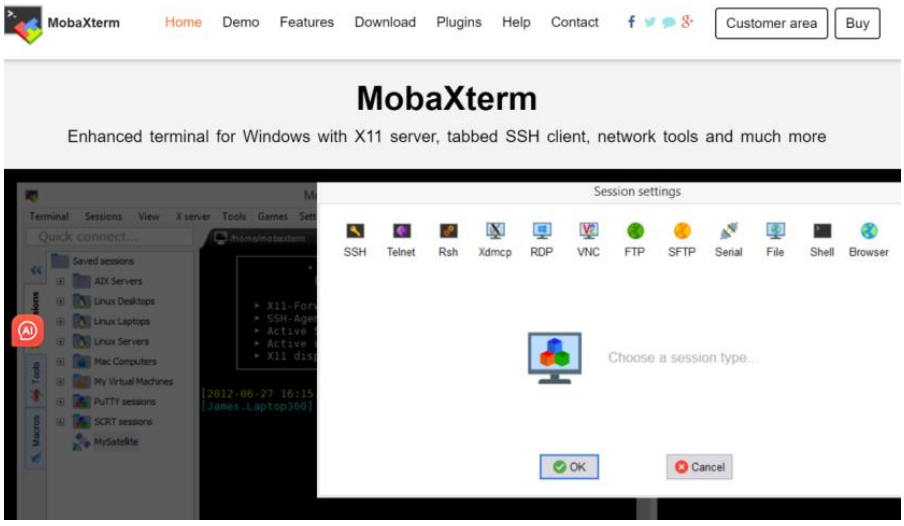
sudo apt-get install openssh-server

上述命令安装 ssh 服务， ssh 的配置文件为/etc/ssh/sshd_config，使用默认配置即可。

2.6. MobaXterm

2.6.1. 软件下载安装

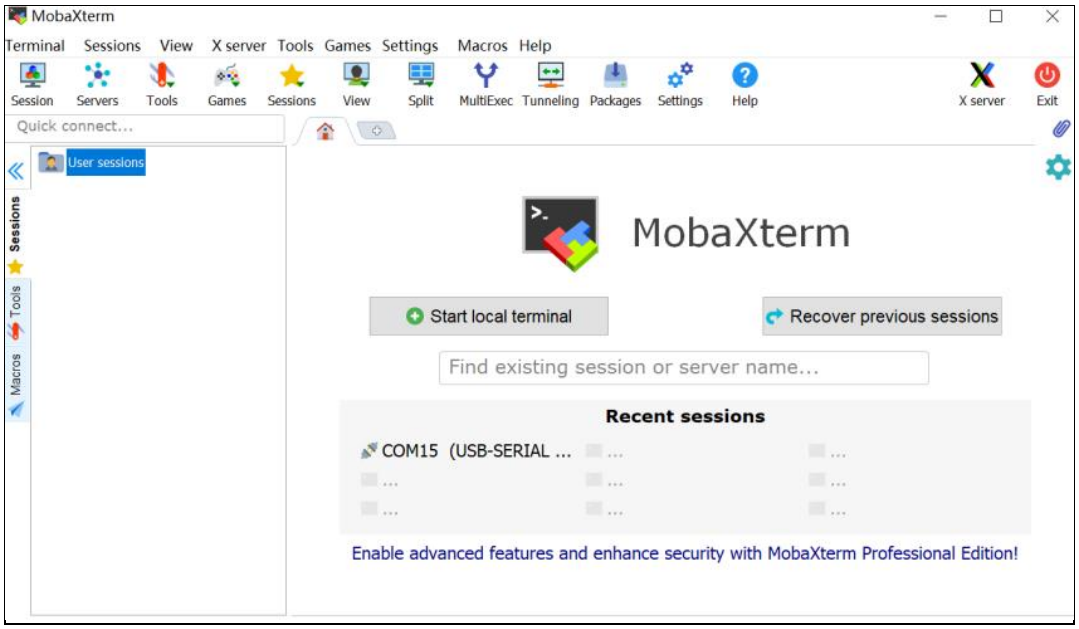
MobaXterm 是一款终端软件，功能强大而且免费，推荐使用此软件作为终端调试软件，MobaXterm 软件在其官网下载即可，地址为 https://mobaxterm.mobatek.net/，如图所示：



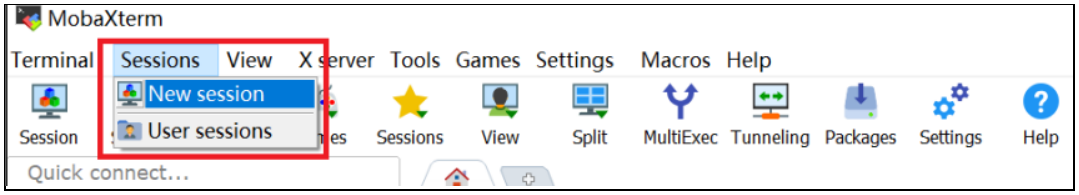
安装包已经放到了开发板中，路径为： 03_Tools ->MobaXterm_Installer_v22.3.zip。打开此压缩包即可。

2.6.2. 软件使用

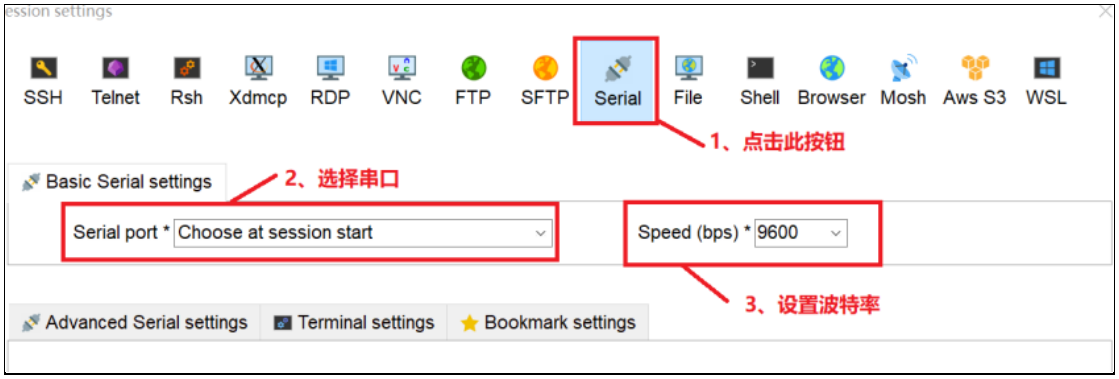
双击 MobXterm 图标，打开此软件，软件界面如图所示：



点击菜单栏中的“Sessions->New session”按钮，打开新建会话窗口，如图所示



建立 Serial 连接，也就是串口连接，因为我们使用 MobaXterm 的主要目的就是作为串口终端使用。点击图中的“Serial”按钮，打开串口设置界面，如图所示：



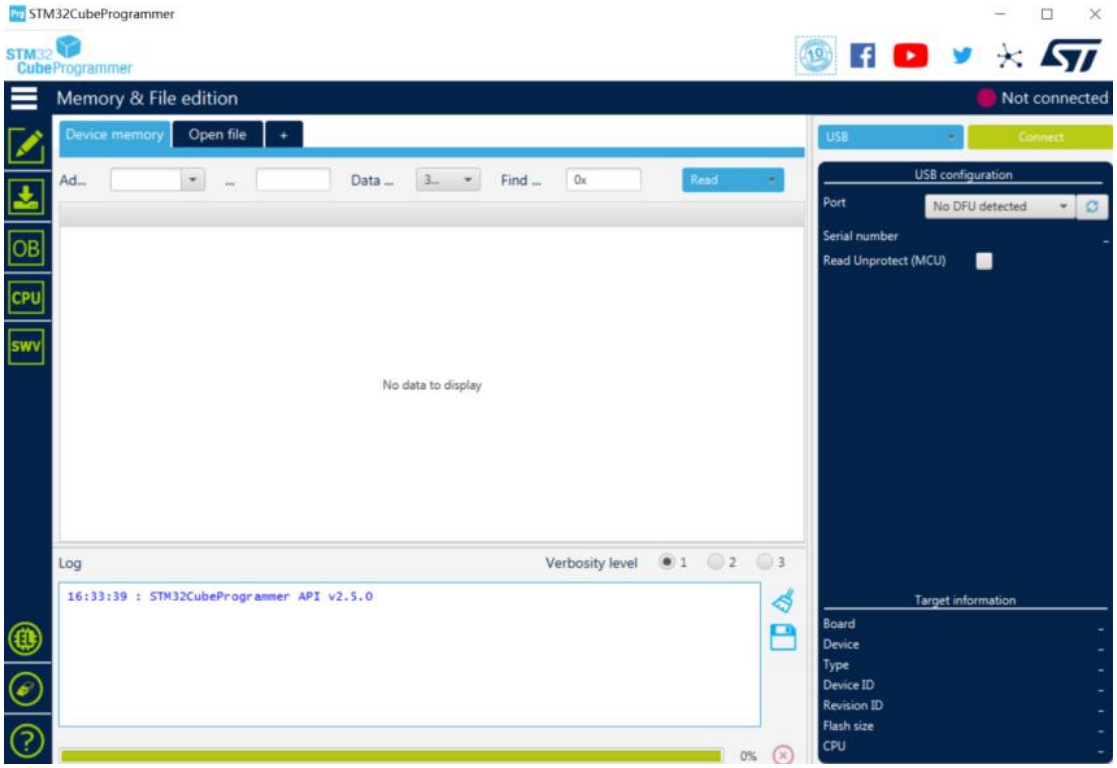
打开串口设置窗口以后先选择要设置的串口号，因此要先用串口线将开发板连接到电脑上上，然后设置波特率为 115200(根据自己实际需要设置)。

2.7. 安装软件开发工具

2.7.1. STM32CubeProg

ST 推出的高集成度的编程工具 STM32CubeProgrammer，它可以在烧录的过程中对未分区的的存储设备进行分区，一旦分区就可以使用已经编译好的二进制文件对某个分区单独进行更新。用户可以根据需要选择使用合适的版本，下载地址如下：https://www.st.com/zh/development-tools/stm32cubeprog.html 。路径为 03_Tools → en.stm32cubeprog_v2-5-0.zip。

解压开发板中的 STM32CubeProgrammer 安装压缩包，然后双击解压出来的“SetupSTM32CubeProgrammer-2.5.0.exe”，安装过程很简单，根据提示进行安装即可，双击图标打开 STM32CubeProgrammer，如图所示

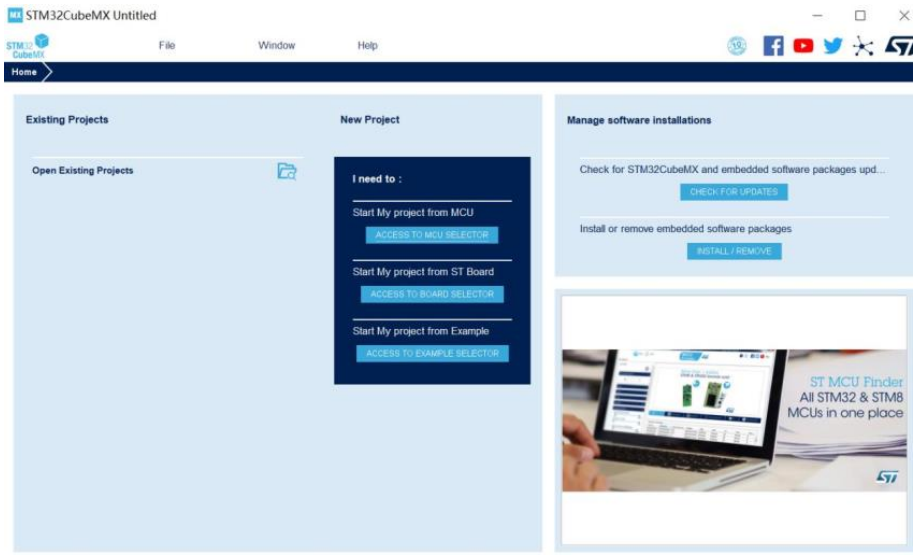


2.7.2. STM32CubeMX

STM32CubeMX 是 ST 公司推出的专门用于生成 STM32 的 HAL 代码的代码生成软件。它利用可视化界面来进行 STM32 时钟、定时器、DMA、串口、GPIO 等各种资源的配置。目前 ST 已经将 STM32MPU 系列 CPU 添加进 STM32CubeMX，我们也可以用此工具来配置 TF-A，Optee，U-boot 以及 Kernel 的设备树和时钟，所以它可以用来在 Cortex A7 开发时生成设备树和时钟，下载地址如下：

https://www.st.com/zh/development-tools/stm32cubemx.html 。 路 径 为 03_Tools → en.stm32cubemx_v6-0-1.zip。

双击 CubeMX 图标，打开以后如图所示：



2.7.3. JAVA 环境

在安装 STM32CbeMX 和 STM32CubeIDE 前我们要先安装 Java 的环境， Java 运行环境版本必须是 V1.7 及以上，否则会导致上述两个应用程序无法使用。 https://www.java.com/zh-CN/download/manual.jsp 查找下载最新的 64 位 Java 软件。 路径为 03_Tools→jre-8u271-windows-x64.exe。

安装完 Java 运行环境之后，为了检测是否正常安装，我们可以打开 Windows 的 cmd 命令输入框，输入如下命令：

java -version //命令查询 Java 版本

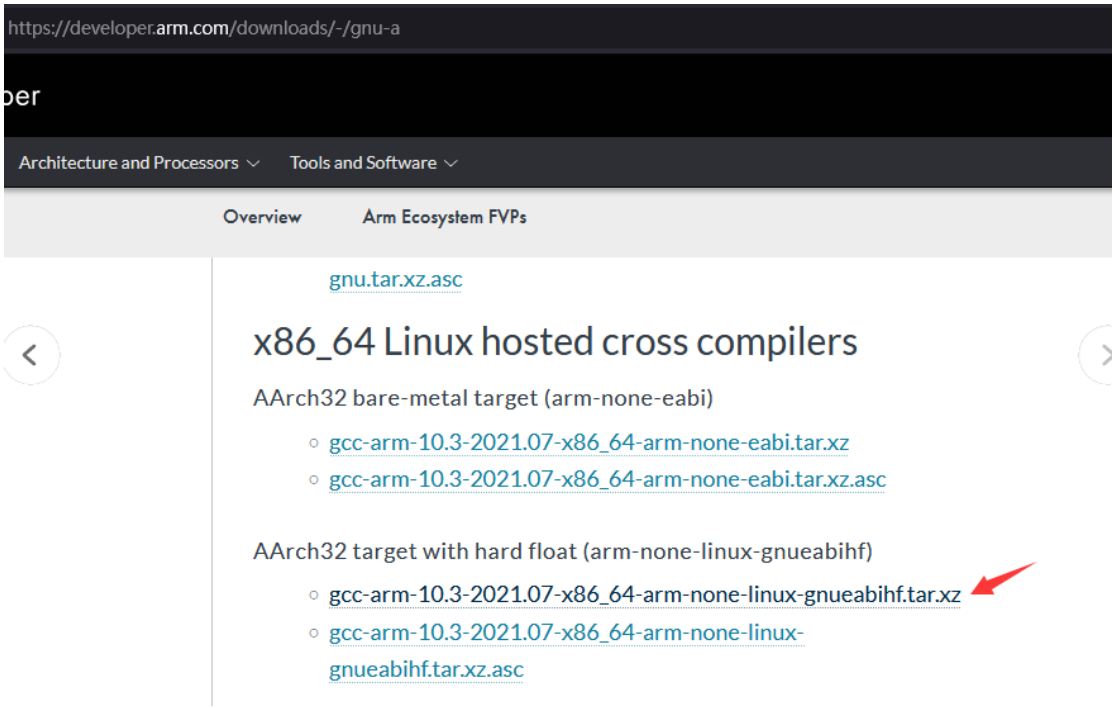
如果安装成功的话就会打印出 Java 的版本号，如图所示：

```
C:\Users\PC>java -version
java version "1.8.0_271"
Java(TM) SE Runtime Environment (build 1.8.0_271-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.271-b09, mixed mode)
```

2.8. 安装交叉编译工具链

用户可以直接使用这个交叉编译工具链来单独编译 Bootloader，Kernel 或者编译自己的应用程序，具体过程在后面的章节中将会详细介绍。这里先介绍交叉编译工具链的安装步骤。

使用 ARM 官方出品的交叉编译器，编译器下载地址如下：https://developer.arm.com/tools-and-software/open-source-software/developertools/gnu-toolchain/gnu-a/downloads，打开后如图所示。



路径为 03_Tools→gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz。

在 Ubuntu 中创建目录： /usr/local/arm

sudo mkdir /usr/local/arm

将交叉编译器放到上面的目录中并解压

sudo cp gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz /usr/local/arm/ -f

sudo tar -vxf gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz

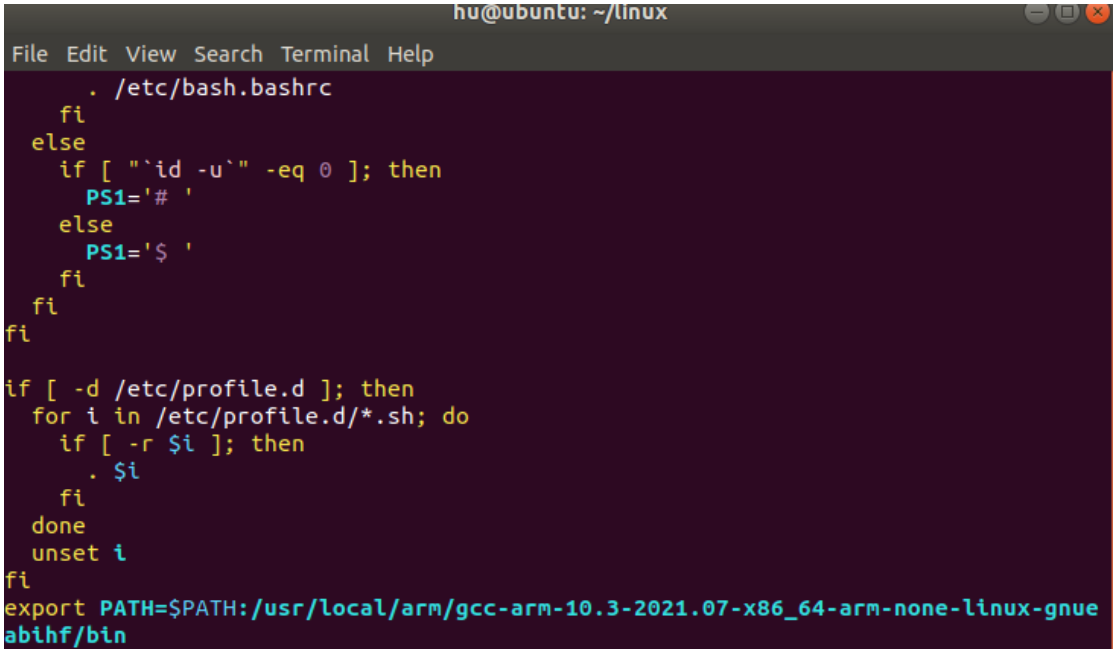
等待解压完成，解压完成以后会生成一个名为“gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf”的文件夹，这个文件夹里面就是我们的交叉编译工具链。

修改环境变量，使用打开/etc/profile 文件，命令如下：

sudo vi /etc/profile

打开/etc/profile 以后，在最后面输入如下所示内容：

Export PATH=$PATH:/usr/local/arm/gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf/bin



修改好以后就保存退出，重启 Ubuntu 系统，交叉编译工具链(编译器)就安装成功了。

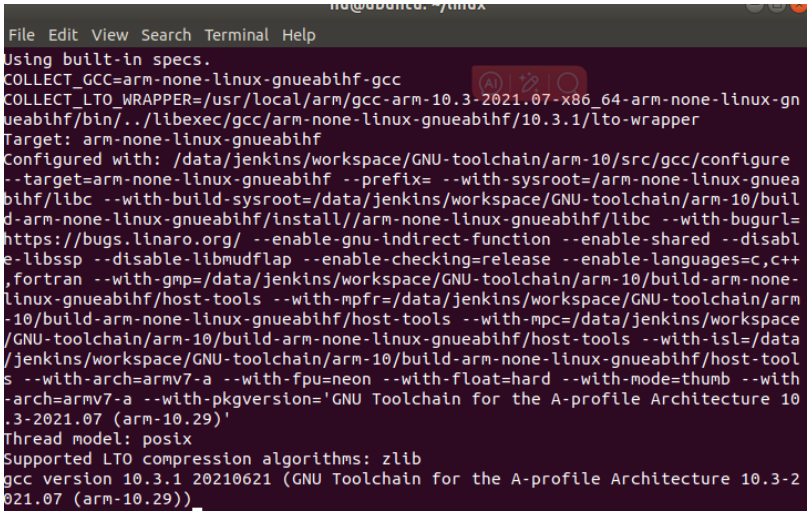
2.9. 安装相关库

sudo apt-get update

sudo apt-get install lsb-core lib32stdc++6

安装验证

arm-none-linux-gnueabihf-gcc -v



“arm-none-linuxgnueabihf-gcc”的含义如下：

1、 arm 表示这是编译 arm 架构代码的编译器。

2、 none 表示厂商，一般半导体厂商会修改通用的交叉编译器，此处就是半导体厂商的名字， ARM 自己做的交叉编译这里为 none，表示没有厂商。

3、 linux 表示运行在 linux 环境下。

4、 gnueabihf 表示嵌入式二进制接口，后面的 hf 是 hard float 的缩写，也就是硬件浮点运算，说明此交叉编译工具链支持硬件浮点运算。

5、 gcc 表示是 gcc 工具。

3. 如何烧录系统镜像

3.1. 使用 STM32CubeProg 烧写

3.1.1. 烧写工具和材料

开发板软件镜像包

USB Type_C 一根

电源适配器

STM32CubeProgrammer 官方软件

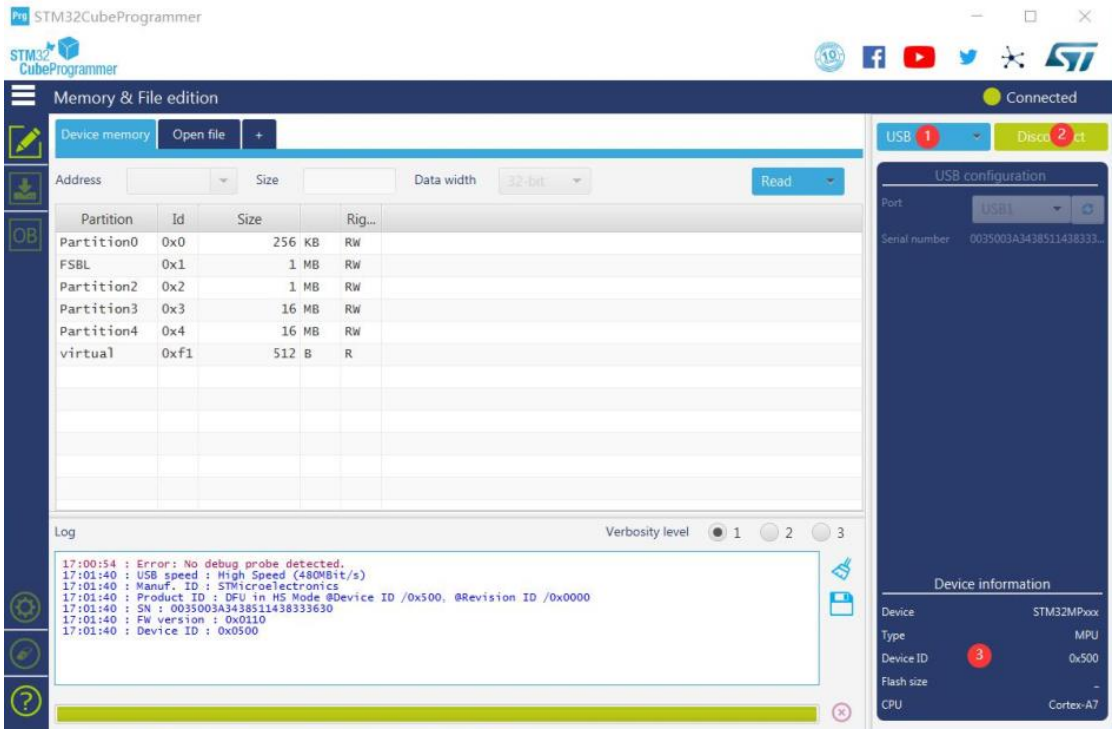
软件镜像包

3.1.2. 设置硬件和 programer

将拨码开关拨到 Download 模式（B2/B1/B0 : 0 0 0）。连接好硬件，将开发板的 OTG 接口与电脑连接，插入电源适配器。

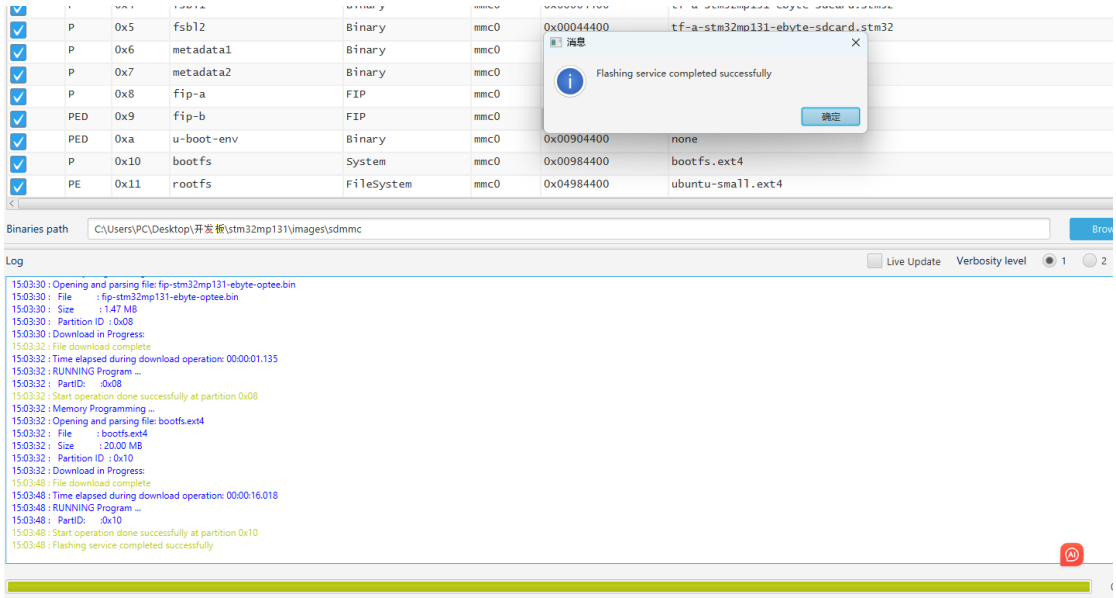
打开 Windows 平台已经安装好的 STM32CubeProgrammer 软件，选择 USB 烧录方式。

点击 connect 按钮连接。检查能否正常显示开发板的信息，如下图。



3.1.3. 选择配置文件

选择打开 02_Images/flashlayout-nandflash.tsv，点击 download 进行烧录。



烧录完成之后就可以根据烧录的具体位置启动镜像了。

3.2. 制作 SD 卡启动器

3.2.1. 准备材料

SD 卡

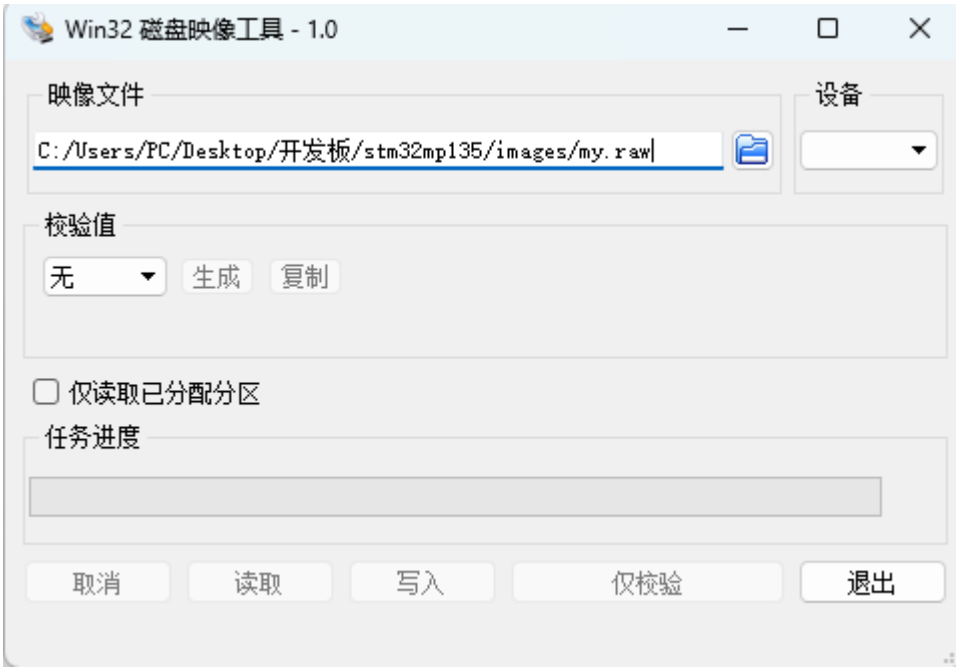
镜像制作工具（03_Tools/ Win32DiskImager-1.0.0-binary/Win32DiskImager.exe）

3.2.2. 制作 SD 卡启动

格式化 SD 卡

打开 Win32DiskImager-1.0.0-binary

选择镜像文件 flashlayout-sdcard.raw



文件类型要选择*.*才能找到烧录文件

点击写入进行烧录，等待写入完成，大约 1 分钟完成，此速度取决于 SD 的读写速度。

4. 如何构建镜像

前面的章节已经比较完整的讲述了将镜像烧录到开发板上的完整流程。由于 ECB10-TB13X 核心板的很多管脚都具有多种功能复用的特性，所以实际项目中基于 ECB10-TB13X 核心板设计的底板与 ECK10-135A5M1G-I 相比总会有一些差异。这些差异可能是去掉显示，增加更多 GPIO，或者需要增加更多串口，还有可能通过 SPI，I2C，USB 等扩展一些外设等等，本章从一个系统开发人员的角度来讲述开发和定制自己系统的具体过

程。

4.1. TF-A

在使用 Trusted Boot Chain（关于 Boot Chain 的详细信息，请查看 https://wiki.st.com/stm32mpu/wiki/Boot_chain_overview）方式启动时，TF-A 用作 FSBL（First Stage Boot Loader），主要实现 DDR, Clock 等核心资源的初始化，基于 ECB10-TB13X 核心板设计的硬件一般不需要修改 TFA，如果特殊情况下，需要修改某个时钟，或者需要对某个管脚进行早期的控制时，可以参照下面的方法进行移植。

4.1.1. stm32wrapper4dbg 工具安装

编译 TF-A 或者 Uboot 的时候需要用到 stm32wrapper4dbg 这个工具，否则编译会报错。 ST 提供了这个工具的源码，我们需要在 Ubuntu 下编译并安装这个源码，源码的下载地址为： https://github.com/STMicroelectronics/stm32wrapper4dbg，这个我们已经下载下来并放到了开发板中，路径为：03_Tools→stm32wrapper4dbg-master.zip。 将源码压缩包拷贝到 Ubuntu 下，然后进行解压，命令如下：

unzip stm32wrapper4dbg-master.zip

cd stm32wrapper4dbg-master

make

编译完成以后就会得到一个名为“stm32wrapper4dbg”的工具，拷贝到 Ubuntu 的/usr/bin 目录下

sudo cp stm32wrapper4dbg /usr/bin

在终端验证安装是否成功

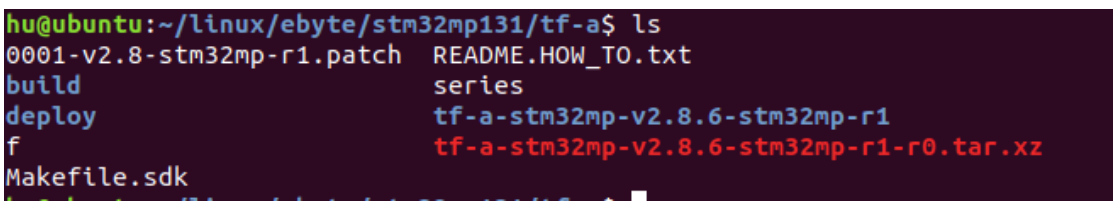
stm32wrapper4dbg -s

4.1.2. 获取源码并编译

首先安装设备树编译相关命令，输入如下命令：

sudo apt-get install device-tree-compiler

获取源码后进行解压，源码路径为 04_Source/tf-a.tar.gz，解压后如图所示



Makefile.sdk 文件主要定义了一些编译属性，比如要使用的交叉编译器、编译的一些选项等等， Makefile.sdk 最终会调用 TF-A 内部的 Makefile 来编译 TF-A 源码。出厂源码经过修改，在 Makefile.sdk 里面指定了使用第一章安装的 arm-none-linux-gnueabihf 交叉编译工具链，且指定编译“stm32mp131-ebyte,stm32mp135-ebyte”设备树文件，还有一些其他配置等。

```
File  Edit  View  Search  Terminal  Help
#TF_A_EXTRA_OPTFLAGS_trusted ?= AARCH32_SP=sp_min
#TF_A_BINARY_trusted ?= tf-a
#TF_A_MAKE_TARGET_trusted ?= bl32 dtbs

# Init default config settings
TF_A_DEVICETREE_emmc ?=  stm32mp131-ebyte stm32mp135-ebyte
TF_A_EXTRA_OPTFLAGS_emmc ?= STM32MP_EMMC=1 PSA_FWU_SUPPORT=1
TF_A_BINARY_emmc ?= tf-a
TF_A_MAKE_TARGET_emmc ?= all

# Init default config settings
TF_A_DEVICETREE_nand ?=  stm32mp131-ebyte stm32mp135-ebyte
TF_A_EXTRA_OPTFLAGS_nand ?= STM32MP_RAW_NAND=1 STM32MP_FORCE_MTD_START_OFFSET=0x
00200000
TF_A_BINARY_nand ?= tf-a
TF_A_MAKE_TARGET_nand ?= all

# Init default config settings
#TF_A_DEVICETREE_nor ?=  stm32mp157a-ev1 stm32mp157c-ev1 stm32mp157d-ev1 stm32mp
157f-ev1
#TF_A_EXTRA_OPTFLAGS_nor ?= STM32MP_SPI_NOR=1 PSA_FWU_SUPPORT=1 STM32MP_FORCE_MT
D_START_OFFSET=0x00080000
#TF_A_BINARY_nor ?= tf-a
```

打开源码根目录后，执行编译脚本

./build.sh

```
#!/bin/sh
# script-name: build.sh
# version: 1.0
# date: 2024-04-08
# author: ebyte
#本脚本用作编译TF-A
#

#删除上层目录的编译文件
rm ../build/ ../deploy/ -rf

#运行编译
make -f ../Makefile.sdk all
~
```

编译完成后会在上一层目录，也就是 tf-a 目录下生成两个目录“build” 和“deploy”，如下图所示。会在上层目录的 deploy 目录下生成编译文件

├── fwconfig

```
|   ├── stm32mp131-ebyte-fw-config-optee.dtb
|   └── stm32mp135-ebyte-fw-config-optee.dtb
├── metadata.bin
├── tf-a-stm32mp131-ebyte-nand.stm32
├── tf-a-stm32mp131-ebyte-sdcard.stm32
├── tf-a-stm32mp131-ebyte-usb.stm32
├── tf-a-stm32mp135-ebyte-nand.stm32
├── tf-a-stm32mp135-ebyte-sdcard.stm32
└── tf-a-stm32mp135-ebyte-usb.stm32
```

4.2. Optee-os

OP-TEE 是实现 Arm TrustZone 技术的开源可信执行环境 (TEE)，与安全领域相关 。源码目录在 04_Source/optee.tar.gz，解压之后如下图所示。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/optee$ ls
0001-3.19.0-stm32mp-r1.patch  optee-os-stm32mp-3.19.0-stm32mp-r1
build                         optee-os-stm32mp-3.19.0-stm32mp-r1-r0.tar.xz
deploy                        README.HOW_TO.txt
fonts.tar.gz                  series
Makefile.sdk
```

由上图可见， 和 TF-A 源码类似， 解压后出现一个 Makefile.sdk 文件和 OP-TEE 源码目录。其中 Makefile.sdk 文件主要定义了一些编译属性，比如要使用的交叉编译器、编译的一些选项等等， Makefile.sdk 最终会调用 OP-TEE 内部的 Makefile 来编译 OP-TEE 源码。出厂源码经过修改，在 Makefile.sdk 里面指定了使用第一章安装的 arm-buildroot-linux-gnueabihf 交叉编译工具链，且指定编译“stm32mp135-ebyte”设备树文件，还有一些其他配置等。

```
# Set default path
SRC_PATH ?= $(PWD)
BLD_PATH ?= $(SRC_PATH)/../build
DEPLOYDIR ?= $(SRC_PATH)/../deploy

# Set default optee-os config
CFG_EMBED_DTB_SOURCE_FILE ?= stm32mp135-ebyte
OPTEE_DRAMSIZE ?= 0x20000000
OPTEE_DRAMSIZE_EV ?= 0x40000000
OPTEE_DRAMSIZE_DK ?= 0x20000000

# Remove default variables
LDFLAGS =
CFLAGS =
CPPFLAGS =
# Define default make options
EXTRA_OEMAKE = PLATFORM=stm32mp1 CROSS_COMPILE_core=arm-none-linux-gnueabihf- CR
OSS_COMPILE_ta_arm64=arm-none-linux-gnueabihf- CFG_ARM32_core=y CROSS_COMPILE_ta
_arm32=arm-none-linux-gnueabihf- NOWERROR=1 LDFLAGS= CFG_TEE_CORE_LOG_LEVEL=2 CF
G_TEE_CORE_DEBUG=y
```

编译出厂源码时，用户不必修改任何内容。 适配于 ECK10-135A5M1G-I 开发板硬件资源的 OPTEE 源码设备树路径为 core/arch/arm/dts/，设备树文件为 stm32mp135-ebyte.dts、stm32mp13-pinctrl-ebyte.dtsi。

获取源码并解压之后执行编译脚本会生成镜像文件

./build.sh

```
#!/bin/sh
# script-name: build.sh
# version: 1.0
# date: 2023-04-08
# author: ebyte
#本脚本用作编译OP-TEE
#

#删除上层目录的编译文件
rm ../build/ ../deploy/ -rf/

#运行编译
make -f ../Makefile.sdk -j8 all
~
~
```

和 TF-A 编译结果类似， OP-TEE 源码编译完成后会在上一层目录，也就是 optee 目录下生成两个目录“build” 和“deploy”，如下图所示。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/optee$ ls
0001-3.19.0-stm32mp-r1.patch  optee-os-stm32mp-3.19.0-stm32mp-r1
build                         optee-os-stm32mp-3.19.0-stm32mp-r1-r0.tar.xz
deploy                        README.HOW_TO.txt
fonts.tar.gz                  series
Makefile.sdk
```

build 目录包含编译过程生成的所有文件，比如.a、 .o 等格式文件，这些文件大部分对用户来说也是不需要接触到的。

而 deploy 目录下的文件则很重要，它是由 Makefile.sdk 脚本对 build 目录里面生成的

二进制文件进行二次处理，复制到 deploy 目录下。它包含 3 个文件 tee-header_v2-stm32mp135-ebyte.bin 、 tee-pageable_v2-stm32mp135-ebyte.bin 和 tee-pager_v2-stm32mp135-ebyte.bin。这 3 个文件，就是编译出厂 OP-TEE 源码的最终产物

~/linux/ebyte/stm32mp131/optee/deploy$ ls

tee-header_v2-stm32mp135-ebyte.bin

tee-pageable_v2-stm32mp135-ebyte.bin

tee-pager_v2-stm32mp135-ebyte.bin

出厂 OP-TEE 源码已编译完成，下面介绍 U-Boot 源码编译。

4.3. U-boot

uboot 的全称是 Universal Boot Loader， uboot 是一个遵循 GPL 协议的开源软件，uboot 是一个裸机代码，可以看作是一个裸机综合例程。现在的 uboot 已经支持液晶屏、网络、 USB 等高级功能。 uboot 官网为 http://www.denx.de/wiki/U-Boot/。

4.3.1. 获取源码并编译

在编译 uboot 前需要安装一下库，输入如下命令

sudo apt-get install bison flex

在 ubuntu 的目录下创建一个名为“uboot” 的目录，然后把亿佰特出厂 U-Boot 源码包 04_Source/uboot.tar.gz 拷贝到 uboot 这个目录下。然后进行解压，解压之后如图。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/uboot$ ls
0001-v2022.10-stm32mp-r1-MACHINE.patch
0002-v2022.10-stm32mp-r1-BOARD.patch
0003-v2022.10-stm32mp-r1-MISC-DRIVERS.patch
0004-v2022.10-stm32mp-r1-DEVICETREE.patch
0005-v2022.10-stm32mp-r1-CONFIG.patch
0099-Add-external-var-to-allow-build-of-new-devicetree-fi.patch
build
deploy
Makefile.sdk
README.HOW_TO.txt
series
u-boot-stm32mp-v2022.10-stm32mp-r1
u-boot-stm32mp-v2022.10-stm32mp-r1-r0.tar.xz
```

由上图可见，和前面源码类似， 解压后出现一个 Makefile.sdk 文件和 U-Boot 源码 u-boot-stm32mp-v2022.10-stm32mp-r1 目录。

这里 Makefile.sdk 文件主要定义了一些编译属性，比如 U-Boot 默认源码配置、 指定编译设备树名称、编译的一些选项等等，如下图所示。Makefile.sdk 最终会调用 U-Boot 内部的 Makefile 来编译 UBoot 源码。

```
# Set default path
SRC_PATH ?= $(PWD)
BLD_PATH ?= $(SRC_PATH)/../build
DEPLOYDIR ?= $(SRC_PATH)/../deploy

# Default U-Boot overall settings to null
UBOOT_CONFIG ?=
UBOOT_DEFCONFIG ?=
UBOOT_BINARY ?=
UBOOT_DEVICETREE ?=

# Init default config settings
# UBOOT_CONFIGS += trusted
# UBOOT_DEFCONFIG_trusted += stm32mp15_defconfig
# UBOOT_BINARY_stm32mp15_defconfig ?= u-boot.dtb
# UBOOT_DEVICETREE_stm32mp15_defconfig ?=  stm32mp157c-ed1 stm32mp157f-ed1 stm32
mp157a-ev1 stm32mp157c-ev1 stm32mp157d-ev1 stm32mp157f-ev1 stm32mp157a-dk1 stm32
mp157d-dk1 stm32mp157c-dk2 stm32mp157f-dk2
# Init default config settings
UBOOT_CONFIGS += trusted
UBOOT_DEFCONFIG_trusted += stm32mp13_defconfig
UBOOT_BINARY_stm32mp13_defconfig ?= u-boot.dtb
UBOOT_DEVICETREE_stm32mp13_defconfig ?=  stm32mp135-ebyte
```

编译出厂源码时，用户不必修改任何内容。 适配于 ECK10-135A5M1G-I 开发板硬件资源的 uboot 源码设备树路径为 arch/arm/dts/，设备树文件为 stm32mp135-ebyte.dts、stm32mp13-pinctrl-ebyte.dtsi。

进入 U-Boot 源码目录 u-boot-stm32mp-v2022.10-stm32mp-r1 之后执行./build.sh 脚本会在上级目录的 deploy 下生产镜像文件。

```
#!/bin/sh
# script-name: build.sh
# version: 1.0
# date: 2023-04-08
# author: ebyte
#本脚本用作编译U-boot
#

#删除上层目录的编译文件
rm ../build/ ../deploy/ -rf

#运行编译
make CROSS_COMPILE=arm-none-linux-gnueabihf- -f ../Makefile.sdk all -j8
~
~
~
```

和前面源码类似， U-Boot 源码编译完成后会在上一层目录，也就是 uboot 目录下生成两个目录“build” 和“deploy”，如下图所示。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/uboot$ ls
0001-v2022.10-stm32mp-r1-MACHINE.patch
0002-v2022.10-stm32mp-r1-BOARD.patch
0003-v2022.10-stm32mp-r1-MISC-DRIVERS.patch
0004-v2022.10-stm32mp-r1-DEVICETREE.patch
0005-v2022.10-stm32mp-r1-CONFIG.patch
0099-Add-external-var-to-allow-build-of-new-devicetree-fi.patch
build
deploy
Makefile.sdk
README.HOW_TO.txt
series
u-boot-stm32mp-v2022.10-stm32mp-r1
```

build 目录包含编译过程生成的所有文件，这些文件大部分对用户来说是不需要接触到的。而 deploy 目录下的文件则很重要，它是由 Makefile.sdk 脚本对 build 目录里面的二进制文件进行二次处理，复制到 deploy 目录下。它包含 2 个文件 u-boot-nodtb-stm32mp13.bin 和 u-boot-stm32mp135-ebyte-trusted.dtb。这 2 个文件，就是编译出厂 U-Boot 源码的最终产物，也就是 ECB10-TB13X 核心板的 U-Boot 固件。

4.3.2. 设置 Uboot 启动内核方式

4.3.2.1. bootcmd

bootcmd 保存着 uboot 默认命令， uboot 倒计时结束以后就会执行 bootcmd 中的命令。这些命令一般都是用来启动 Linux 内核的，比如读取 EMMC 或者 NAND Flash 中的 Linux 内核镜像文件和设备树文件到 DRAM 中，然后启动 Linux 内核。可以在 uboot 启动以后进入命令行设置 bootcmd 环境变量的值。

bootcmd 的默认值就是 CONFIG_BOOTCOMMAND，bootargs 的默认值就是 CONFIG_BOOTARGS。我们可以直接在 stm32mp1.h 文件中通过设置宏 CONFIG_BOOTCOMMAND 来设置 bootcmd 的默认值。

4.3.2.2. bootargs

bootargs 保存着 uboot 传递给 Linux 内核的参数，比如指定 Linux 内核所使用的 console、指定根文件系统所在的分区等，如下面 bootargs 环境变量值：

console=ttySTM0,115200 root=/dev/mmcblk2p3 rootwait rw

console 用来设置 linux 终端(或者叫控制台)，也就是通过什么设备来和 Linux 进行交互，是串口还是 LCD 屏幕？如果是串口的话应该是串口几等等。一般设置串口作为 Linux 终端，这样我们就可以在电脑上通过 MobaXterm 来和 linux 交互了。这里设置 console 为 ttySTM0，因为 linux 启动以后 STM32MP1 的串口 4 在 linux 下的设备文件就是 /dev/ttySTM0，在 Linux 下，一切皆文件。ttySTM0 后面有个“,115200”，这是设置串口

的波特率， console=ttySTM0,115200 综合起来就是设置 ttySTM0（也就是串口 4）作为 Linux 的终端，并且串口波特率设置为 115200。使用调试串口终端，开发板启动到 uboot 启动之后通过在 uboot 中设置 bootargs 和 bootcmd 来控制启动方式。设置完成后需要使用命令保存环境变量。后续就可以自动从设置的环境变量来启动内核和文件系统了。

root 用来设置根文件系统的位置， root=/dev/mmcblk2p3 用于指明根文件系统存放在 mmcblk2 设备的分区 3 中。STM32MP1 核心板启动 linux 以后会存在/dev/mmcblk1、/dev/mmcblk2 、/dev/mmcblk1p1 、/dev/mmcblk1p2 、/dev/mmcblk2p1 、/dev/mmcblk2p2 和/dev/mmcblk2p3 这 样 的 文 件 ， 其 中 /dev/mmcblkx(x=0~n) 表 示 mmc 设备，而 /dev/mmcblkxpy(x=0~n,y=1~n)表示 mmc 设备 x 的分区 y。在 STM32MP1 开发板中 /dev/mmcblk2 表示 EMMC，而/dev/mmcblk2p3 表示 EMMC 的分区 3。

root 后面有“rootwait rw”， rootwait 表示等待 mmc 设备初始化完成以后再挂载，否则的话 mmc 设备还没初始化完成就挂载根文件系统会出错的。 rw 表示根文件系统是可以读写的，不加 rw 的话可能无法在根文件系统中进行写操作，只能进行读操作。

rootfstype 此选项一般配合 root 一起使用， rootfstype 用于指定根文件系统类型，如果根文件系统为 ext 格式的话此选项无所谓。如果根文件系统是 yaffs、 jffs 或 ubifs 的话就需要设置此选项，指定根文件系统的类型。

在设置完 bootcmd 和 bootargs 环境变量之后都需要保存。

saveenv

```
STM32MP> saveenv
Saving Environment to MMC... Writing to MMC(0)... OK
```

下面分别介绍三种不同的启动方式。

4.3.2.3. 从 SD 卡启动

setenv bootargs 'console=ttySTM0,115200 root=/dev/mmcblk0p9 rootwait rw'

setenv bootcmd 'ext4load mmc 0:8 c2000000 uImage;ext4load mmc 0:8 c4000000 stm32mp135-ebyte.dtb;bootm c2000000 - c4000000'

4.3.2.4. 从 nandflash 启动

setenv bootcmd 'ubifsmount ubi:boot;ubifsload c2000000 uImage;ubifsload c4000000 stm32mp135-ebyte.dtb;bootm c2000000 - c4000000;ubifsmount ubi:rootfs'

setenv bootargs 'ubi.mtd=UBI rootfstype=ubifs root=ubi:rootfs rootwait rw console=ttySTM0,115200'

4.3.2.5. 从网络启动

从网络启动需要先设置网卡，并配置好服务端（虚拟机）的 nfs 服务和 tftp 服务。

```
setenv ipaddr 192.168.0.250
setenv ethaddr 00:04:9f:04:d2:35
setenv gatewayip 192.168.0.1
setenv netmask 255.255.255.0
setenv serverip 192.168.0.130
```

设置从网络启动

setenv bootcmd 'tftp c2000000 uImage;tftp c4000000 stm32mp135-ebyte.dtb;bootm c2000000 - c4000000'

setenv bootargs 'console=ttySTM0,115200 root=/dev/nfs nfsroot=192.168.0.130:/home/hu/linux/ubuntu/ubuntu-small,proto=tcp rw ip=192.168.0.250:192.168.0.130:192.168.0.1:255.255.255.0::eth0:off

4.3.2.6. 代码设置自动启动

uboot 源码下 include/configs/stm32mp13_st_common.h：用于 STM32MP13x 线路的 STMicroelectronics 板。

我们可以在这里更改 uboot 启动内核的方式，从而实现烧录固件之后自动启动内核，而不需要使用上面章节的方式启动 uboot 之后，手动设置启动方式。

```
#define ST_STM32MP13_BOOTCMD "bootcmd_stm32mp=" \
 "echo \"Boot over ${boot_device}${boot_instance}!\";" \
 "if test ${boot_device} = serial || test ${boot_device} = usb;" \
 "then stm32prog ${boot_device} ${boot_instance}; " \
 "else " \
  "run env_check;" \
  "if test ${boot_device} = mmc;" \
  "then env set bootargs \"console=ttySTM0,115200 root=/dev/mmcblk0p9 rootwait rw\";" \
  "ext4load mmc 0:8 c2000000 uImage;"\
  "ext4load mmc 0:8 c4000000 stm32mp135-ebyte.dtb;"\
  "bootm c2000000 - c4000000;"\
  "fi;" \
```

```
"if test ${boot_device} = nand ||" \
   " test ${boot_device} = spi-nand ;" \
"then env set bootargs \"ubi.mtd=UBI rootfstype=ubifs root=ubi:rootfs rootwait rw console=ttySTM0,115200\";" \
"ubifsmount ubi:boot;"\
"ubifsload c2000000 uImage;"\
"ubifsload c4000000 stm32mp135-ebyte.dtb;"\
"bootm c2000000 - c4000000;"\
"fi;" \
"if test ${boot_device} = nor;" \
"then env set boot_targets mmc0; fi;" \
"run distro_bootcmd;" \
"fi;\0"
```

如上述代码所示，如果是从 sd 卡启动（mmc），就从 sd 卡的分区来加载设备树，内核和根文件系统。如果是从 nand 启动，就从 nand 来进行加载。

至此出厂 U-Boot 源码已编译完成，下面介绍使用 fiptool 工具封装 OP-TEE 和 U-Boot 固件。

4.4. fiptool 封装 OP-TEE 和 U-Boot 固件

本节介绍如何使用 fiptool 工具来将 OP-TEE 和 U-Boot 等固件文件打包在一起，封装成单个镜像文件 fip-stm32mp135d-atk-optee.bin， 后续烧写到开发板。

FIP，英文全称 Firmware Image Package，直译为固件镜像包。 FIP 是 TF-A 使用的一种打包格式，用于将引导加载程序镜像文件（bootloader） 以及其他有效载荷文件打包为单个二进制文件。 现在 ST 发布的生态系统开发包中推荐使用 FIP 来启动 STM32MP1 平台。只有 TF-A（带有 STM32 头部结构信息)镜像被 ROM 代码加载，而其他二进制文件比如 OP-TEE、 U-Boot 及其各自的设备树文件则封装为 FIP 镜像包形式。 TF-A 运行时能够加载并鉴权 FIP 镜像。

故 ECB10-TB13X 核心板的出厂系统启动、运行过程，可通俗地概述为：

ROM 代码->TF-A->FIP 镜像（含 OP-TEE 和 U-Boot） ->Linux 内核->文件系统。

在 TF-A 源码中，提供了一个实现 FIP 功能的工具 fiptool。下面我们进入 TF-A 的

fiptool 源码目录进行编译， 来生成 fiptool 工具，编译指令如下：

cd tf-a-stm32mp-v2.8.6-stm32mp-r1 / //进入到自己的 TF-A 源码目录

cd tools/fiptool //进入 fiptool 源码

make //编译 fiptool 源码

```
hu@ubuntu:~/linux/ebyte/stm32mp131/tf-a/tf-a-stm32mp-v2.8.6-stm32mp-r1/tools/fip
tool$ make
  HOSTCC  fiptool.c
  HOSTCC  tbbr_config.c
  HOSTLD  fiptool

Built fiptool successfully

hu@ubuntu:~/linux/ebyte/stm32mp131/tf-a/tf-a-stm32mp-v2.8.6-stm32mp-r1/tools/fip
tool$ ls
fiptool     fiptool.o           Makefile.msvc  tbbr_config.o
fiptool.c   fiptool_platform.h  tbbr_config.c  win_posix.c
fiptool.h   Makefile            tbbr_config.h  win_posix.h
```

将上面编译生成的 fiptool 工具复制到 ubuntu 的/usr/bin/目录下

sudo cp fiptool /usr/bin/

```
hu@ubuntu:/usr/bin$ ls fiptool
fiptool
```

执行如下指令，查看 fiptool 工具帮助信息；

```
hu@ubuntu:/usr/bin$ fiptool help
usage: fiptool [--verbose] <command> [<args>]
Global options supported:
  --verbose      Enable verbose output for all commands.

Commands supported:
  info           List images contained in FIP.
  create         Create a new FIP with the given images.
  update         Update an existing FIP with the given images.
  unpack         Unpack images from FIP.
  remove         Remove images from FIP.
  version        Show fiptool version.
  help           Show help for given command.
```

下面我们使用 fiptool 工具对 OP-TEE、U-Boot 等二进制镜像文件进行打包，封装 FIP 镜像包 fip-stm32mp135-ebyte-optee.bin，

首先我们在 ubuntu 目录下创建一个名为“fip” 的目录， 分别将 TF-A、OP-TEE 及 U-Boot 编译后生成的 6 个镜像文件，复制到 fip 目录下，所需文件如下表所示。

源码	所在目录	文件名
TF-A	TF-A/deploy/fwconfig	stm32mp135-ebyte-fw-config-optee.dtb
Optee	Optee/deploy	tee-header_v2-stm32mp135-ebyte.bin tee-pager_v2-stm32mp135-ebyte.bin

		tee-pageable_v2-stm32mp135-ebyte.bin
U-Boot	Uboot/deploy	u-boot-stm32mp135-ebyte-trusted.dtb u-boot-nodtb-stm32mp13.bin

确保 fip 目录下有上述文件

fip$ tree -l

├── stm32mp135-ebyte-fw-config-optee.dtb

├── tee-header_v2-stm32mp135-ebyte.bin

├── tee-pageable_v2-stm32mp135-ebyte.bin

├── tee-pager_v2-stm32mp135-ebyte.bin

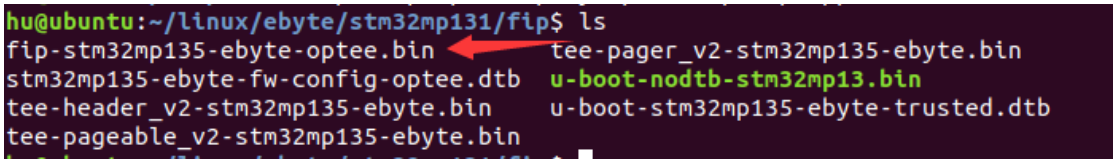
├── u-boot-nodtb-stm32mp13.bin

└── u-boot-stm32mp135-ebyte-trusted.dtb

然后进入 fip 目录，执行下述命令

fiptool create --nt-fw u-boot-nodtb-stm32mp13.bin --hw-config u-boot-stm32mp135-ebyte-trusted.dtb --fw-config stm32mp135-ebyte-fw-config-optee.dtb --tos-fw tee-header_v2-stm32mp135-ebyte.bin --tos-fw-extra1 tee-pager_v2-stm32mp135-ebyte.bin --tos-fw-extra2 tee-pageable_v2-stm32mp135-ebyte.bin fip-stm32mp135-ebyte-optee.bin

会生成 fip-stm32mp135-ebyte-optee.bin 文件。这个二进制文件包含了 OP-TEE 固件和 U-Boot 固件。在开发板出厂系统启动时， TF-A 会加载并鉴权该固件，接着运行下一阶段 OP-TEE 和 U-Boot。



4.5. Kernel

4.5.1. 获取源码并编译

在 ubuntu 的 目录下创建一个名为“linux” 的目录，然后把出厂 linux 源码包 04_Source/linux-6.1.28.tar.gz 拷贝到 linux 这个目录下。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/linux-6.1.28$ ls
arch              include      modules.builtin          System.map
block             init         modules.builtin.modinfo  tmp
build.sh          io_uring     modules.order            tools
built-in.a        ipc          Module.symvers           usr
certs             Kbuild       net                      virt
CONTRIBUTING.md   Kconfig      README                   vmlinux
COPYING           kernel       rust                     vmlinux.a
CREDITS           lib          samples                  vmlinux.o
crypto            LICENSES     scripts                  vmlinux.symvers
Documentation     MAINTAINERS  security
drivers           Makefile     SECURITY.md
fs                mm           sound
```

编译出厂源码时，用户不必修改任何内容。 适配于 ECK10-135A5M1G-I 开发板硬件资源的 uboot 源码设备树路径为 arch/arm/boot/dts/，设备树文件为 stm32mp135-ebyte.dts、stm32mp13-pinctrl-ebyte.dtsi。方便大家的编译，提供了编译脚本，第一次编译的时候可以直接运行脚本编译，脚本如下图所示：

```
make distclean #清除编译      You, 1秒钟前 • Uncommitted changes
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- stm32mp1_ebyte_defconfig

# make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- menuconfig
#编译内核
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- uImage vmlinux dtbs LOADADDR=0xC2000040 -j8

#编译内核模块
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- modules -j16

#在当前目录下新建一个tmp目录，用于存放编译后的目标文件
if [ ! -e "./tmp" ]; then
    mkdir tmp
fi
rm -rf tmp/*

#将编译好的模块安装到tmp目录，通过INSTALL_MOD_STRIP=1移除模块调试信息
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabihf- modules_install INSTALL_MOD_PATH=tmp INSTALL_MOD_STRIP=1
#删除模块目录下的source目录
rm -rf tmp/lib/modules/6.1.28/source
#删除模块的目录下的build目录
rm -rf tmp/lib/modules/6.1.28/build

#跳转到模块目录
cd tmp/lib/modules
#压缩内核模块
tar -jcvf ../../modules.tar.bz2 .
cd -
rm -rf tmp/lib

#拷贝uImage到tmp目录
cp arch/arm/boot/uImage tmp

#拷贝所有编译的设备树文件到当前的tmp目录下
cp arch/arm/boot/dts/stm32mp135-ebyte.dtb tmp
```

然后执行./build.sh 编译脚本，第一次编译可能会比较慢。编译之后会在根目录下的 tmp 文件夹中生成 uImage 镜像和 stm32mp135-ebyte.dtb 设备树文件。

4.5.2. 系统镜像打包

4.5.2.1. 新建 ext4 格式磁盘

首先新建一个 ext4 格式的磁盘文件，然后挂载这个 ext4 格式的磁盘，将

stm32mp135-ebyte.dtb 和 uImage 拷贝到这个 ext4 磁盘即可。

进入到 bootfs 文件夹，然后输入如下命令创建 ext4 磁盘：

cd bootfs

dd if=/dev/zero of=bootfs.ext4 bs=1M count=10

mkfs.ext4 -L bootfs bootfs.ext4

4.5.2.2. 将系统镜像拷贝到 ext4 磁盘中

首先创建一个目录用来挂载前面制作制作出来的 bootfs.ext4，比如我这里创建目录/mnt/bootfs，命令如下：

sudo mkdir /mnt/bootfs

再挂载上面创建的 bootfs.ext4 到 bootfs

sudo mount bootfs.ext4 /mnt/bootfs/

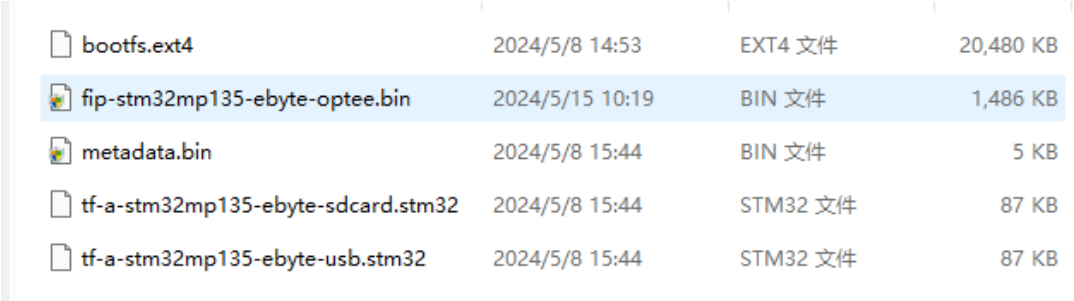
然后将 uImage 和 stm32mp135-ebyte.dtb 拷贝到 bootfs，再取消挂载即可

sudo cp uImage stm32mp135-ebyte.dtb /mnt/bootfs/

sudo umount /mnt/bootfs

4.6. 小结

通过上述步骤，我们得到了所有部分的镜像文件，从中取出我们需要的文件来进行烧写，如下图所示。



其中 metadata.bin，tf-a-stm32mp135-ebyte-sdcard.stm32 和 tf-a-stm32mp135-ebyte-usb.stm32 是来自 tf-a 的 deploy 目录下。然后按照第三章的烧写步骤进行烧写即可。

5. 如何适配不同的硬件平台

5.1. 在设备树中添加硬件资源

用户需要对 CPU 的芯片手册，以及 ECB10-TB13X 核心板的产品手册，管脚定义有比

较详细的了解，以便于根据实际的功能对这些管脚进行正确的配置和使用。

用户可以在 BSP 源码里创建自己的设备树，一般情况下不需要修改 Bootloader 部分中的 TF-a、optee 和 u-boot。用户只需要根据实际的硬件资源对 Linux 内核设备树进行适当的调整即可。在此将 ECB10-TB13X 的 BSP 各个部分中的设备树列表罗列出来，方便用户开发。

项目	设备树	说明
TF-A	stm32mp13x-base.dts	资源设备树
	stm32mp131-ebyte.dts stm32mp131-ebyte-fw-config.dts stm32mp135-ebyte.dts stm32mp135-ebyte-fw-config.dts	核心板资源描述
	stm32mp13-ddr3-1x2Gb-1066-binF.dtsi stm32mp13-ddr3-1x4Gb-1066-binF.dtsi	单片 256MB DDR3 描述 单片 512MB DDR3 描述
	stm32mp13-pinctrl-ebyte.dtsi	管脚复用描述
Optee	stm32mp13x-base.dts	资源设备树
	stm32mp131-ebyte.dts stm32mp135-ebyte.dts	核心板资源描述
	stm32mp13-pinctrl-ebyte.dtsi	管脚复用描述
U-Boot	stm32mp13x-base.dts	资源设备树
	stm32mp131-ebyte.dts stm32mp131-ebyte-u-boot.dtsi stm32mp135-ebyte.dts stm32mp135-ebyte-u-boot.dtsi	核心板资源描述
	stm32mp13-pinctrl-ebyte.dtsi	管脚复用描述
Kernel	stm32mp1_ebyte_defconfig	内核配置文件
	stm32mp13x-base.dts	资源设备树
	stm32mp131-ebyte.dts	ECB10-TB131A 设备树
	stm32mp135-ebyte.dts	ECB10-TB13X 设备树
	stm32mp13-pinctrl-ebyte.dtsi	ECK10-131A2M1G-I

用户一般只需要在内核的设备树中添加不同的硬件资源，再根据具体的硬件资源适配驱动即可。

5.2. 设备树的添加

Linux 内核设备树是一种数据结构，它通过特有的语法格式描述片上片外的设备信息。由 BootLoader 传递给 kernel，kernel 进行解析后形成和驱动程序关联的 dev 结构供驱动代码使用。

在内核源码下 arch/arm/boot/dts 下可以看到大量的平台设备树。如适合 ECK10-135A5M1G-I 的设备树，可在当前路径下增加自定义设备树，如：ebyte-stm32mp135a-xxx.dts。如下图所示。

```
hu@ubuntu:~/linux/ebyte/stm32mp131/linux-6.1.28/arch/arm/boot/dts$ ls stm32mp13*
stm32mp131.dtsi       stm32mp133.dtsi       stm32mp135-ebyte.dts          stm32mp135f-dk.dtb          stm32mp13-pinctrl.dtsi  stm32mp13xc.dtsi
stm32mp131-ebyte.dtb  stm32mp135.dtsi       stm32mp135f-dk-a7-examples.dtb  stm32mp135f-dk.dts          stm32mp13xa.dtsi        stm32mp13xd.dtsi
stm32mp131-ebyte.dts  stm32mp135-ebyte.dtb  stm32mp135f-dk-a7-examples.dts  stm32mp13-ebyte-pinctrl.dtsi  stm32mp13x-base.dts     stm32mp13xf.dtsi
```

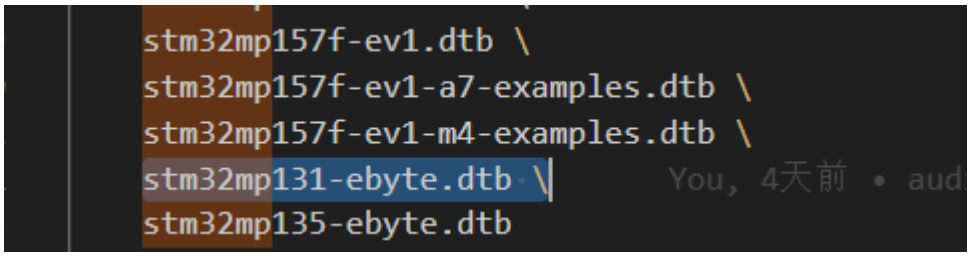
我们将 ECB10-TB13X 核心板相关的资源编写进 stm32mp135-base.dts。其它扩展的接口和设备可以对它们进行引用，如下所示（仅供参考）：

```
/dts-v1/;

#include "stm32mp13x-base.dts"

/ {
    model = "EBYTE TECH STM32MP131 Discovery Board";
    compatible = "st,stm32mp131-ebyte", "st,stm32mp135";

    memory@c0000000 {
        device_type = "memory";
        reg = <0xc0000000 0x10000000>;
    };

    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;

        optee_framebuffer@cd000000 {
            reg = <0xcd000000 0x1000000>;
            no-map;
        };
        optee@ce000000 {
            reg = <0xce000000 0x2000000>;
            no-map;
        };
    };

```

用户增加了新的设备树源文件之后，还需要在同目录下的 Makefile 里添加设备树编译信息，这样就可以在编译内核的时候生成对应的设备树二进制文件。

在 arch/arm/boot/dts/Makefile 中

dtb-$(CONFIG_ARCH_STM32) += \的配置项下添加 stm32mp131-ebyte.dtb \



增加完成后即可增加设备驱动的板载描述，通过 4.5.1 节的方法编译生成设备树 dtb 文件 stm32mp131-ebyte.dtb。

5.3. 如何根据硬件配置 CPU 功能管脚

实现一个功能引脚的控制是一个较为复杂的系统开发过程之一，其中包含了引脚的配置，驱动的开发，应用的实现等等步骤，本节不具体分析每个部分的开发过程，而是以实例来讲解功能管脚的控制实现。

5.3.1. GPIO 管脚的配置方法

GPIO: General-purpose input/output，通用的输入输出口，在嵌入式设备中是一个十分重要的资源，可以通过它们输出高低电平或者通过它们读入引脚的状态-是高电平或是低电平。STM32MP1 封装大量的外设控制器，这些外设控制器与外部设备交户一般是通过控制 GPIO 来实现，而将 GPIO 被外设控制器使用我们称为复用（Alternate Function），给它们赋予了更多复杂的功能，如用户可以通过 GPIO 口和外部硬件进行数据交互(如 UART)，控制硬件工作(如 LED、蜂鸣器等),读取硬件的工作状态信号（如中断信号）等。所以 GPIO 口的使用非常广泛。

STM32MPU 的 GPIO 管脚配置方法一般使用 STM32CubeMX 配置或使用 Datasheet 查表的方式来配置。

5.3.1.1. STM32CubeMX 配置方法

目前 ST 已经将 STM32MPU 系列 CPU 添加进 STM32CubeMX，我们也可以用此工具来配置 TF-A，U-boot 以及 Kernel 的 GPIO 功能设备树和外设时钟。本节不重点讲解其使用方法，您可以通过官方网站获取详细的开发指导说明。

WIKI: https://wiki.st.com/stm32mpu/wiki/STM32CubeMX

ST 官方:https://www.st.com/zh/development-tools/stm32cubemx.html

5.3.1.2. 查询手册方式

GPIO 的配置可通过亿佰特整理的 Datasheet 找到描述文件（01_Documents\Datasheet\STM32MP135A&D 数据手册.pdf），示例如下：

```
i2c1_pins_a: i2c1-0 {
    pins {      You, 上个月 • add ebyte board support
        pinmux = <STM32_PINMUX('B', 8, AF4)>, /* I2C1_SCL */
        <STM32_PINMUX('D', 3, AF5)>; /* I2C1_SDA */
        bias-disable;
        drive-open-drain;
        slew-rate = <0>;
    };
};
```

通过查询数据手册可以得到 I2C_SCL 在 PB8 管脚的复用关系为 AF4，I2C_SDA 在 PD3 引脚的复用关系为 AF5。

PB7	-	TIM17_CH1N	TIM4_CH2	-	-	I2S4_CK	I2C4_SDA	-
PB8	-	TIM16_CH1	TIM4_CH3	-	I2C1_SCL	I2C3_SCL	DFSDM1_DATIN1	-
PB9	TRACED3	-	TIM4_CH4	-	-	-	I2C4_SDA	-

PD2	TRACED4	-	TIM3_ETR	-	I2C1_SMBA	SPI3_NSS/I2S3_WS	SAI2_D1	USART3_RX
PD3	-	-	TIM2_CH1/TIM2_ETR	USART2_CTS/USART2_NSS	DFSDM1_CKOUT	I2C1_SDA	SAI1_D3	-
PD4	-	-	-	USART2_RTS/	-	SPI3_MISO/	DFSDM1_	-

5.3.2. 设备树中引用 GPIO

配置功能管脚为 I2C 功能实例，i2c 主要使用两个 GPIO 分别用作 I2C_SCL 和 I2C_SDA，在 pinctrl 子系统中定义好 i2c 的引脚位置和配置之后，在板级设备树中添加对其的引用即可，如下图所示。这里的 i2c 使用了两种引脚配置，分别是正常使用的默认状态和休眠时的引脚状态，分别为 default 和 sleep，系统根据状态进行自动切换。

```
&i2c1 {
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c1_pins_a>;
    pinctrl-1 = <&i2c1_sleep_pins_a>;
    i2c-scl-rising-time-ns = <100>;
    i2c-scl-falling-time-ns = <7>;
    status = "okay";
    /delete-property/dmas;
    /delete-property/dma-names;
```

在 pinctrl 系统对 i2c 引脚的定义如下图所示。

```
i2c1_pins_a: i2c1-0 {       You, 上个月 • add ebyte board support
    pins {
        pinmux = <STM32_PINMUX('B', 8, AF4)>, /* I2C1_SCL */
        <STM32_PINMUX('D', 3, AF5)>; /* I2C1_SDA */
        bias-disable;
        drive-open-drain;
        slew-rate = <0>;
    };
};

i2c1_sleep_pins_a: i2c1-sleep-0 {
    pins {
        pinmux = <STM32_PINMUX('B', 8, AF4)>, /* I2C1_SCL */
        <STM32_PINMUX('D', 3, AF5)>; /* I2C1_SDA */
    };
};
```

5.3.3. 如何使用自己配置的管脚

5.3.3.1. U-boot 中使用 GPIO 管脚

uboot 可以直接在终端使用命令来控制 GPIO 的设置。如设置 GPIOB6，可使用下列命令。

```
STM32MP> gpio set GPIOB6
gpio: pin GPIOB6 (gpio 22) value is 1
STM32MP> gpio clear GPIOB6
gpio: pin GPIOB6 (gpio 22) value is 0
```

5.3.3.2. 用户空间使用 GPIO 管脚

Linux 操作系统的体系架构分为用户态和内核态（或者用户空间和内核）。用户态即上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源，包括 CPU 资源、存储资源、I/O 资源等。为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口：即系统调用。

Shell 是一个特殊的应用程序，俗称命令行，本质上是一个命令解释器，它下通系统调用，上通各种应用。使用 Shell 脚本，通常短短的几行 Shell 脚本就可以实现一个非常大的功能，原因就是这些 Shell 语句通常都对系统调用做了一层封装。为了方便用户和系统交互。

本节将讲述在用户态如何使用 GPIO 管脚的控制三种基本方式。

5.3.3.2.1. Shell 命令

GPIO 的测试是通过文件系统 sysfs 接口来实现的，下面内容以 PB6 为例说明 GPIO 的使用过程。

pin 脚的编号定义为 #define PIN_NO(port, line) (((port) - 'A') * 0x10 + (line))，其中 port 为 gpio 端口，line 为该 gpio 对应引脚，((port)-'A')代表 ASCII 码相减。如 PB6 对应的 pin 引脚编号为:PIN_NO('B',6)=(0x42-0x41)*0x10+6=(66-65)*16+6=22。

1） 导出 GPIO

echo 94 > /sys/class/gpio/export

导出成功后会在/sys/class/gpio/目录下生成 PB6 这个目录。

2） 设置/查看 GPIO 方向

设置输入

echo "in" > /sys/class/gpio/PB6/direction

设置输出

echo "out" > /sys/class/gpio/PB6/direction

查看 GPIO 方向

cat /sys/class/gpio/PB6/direction

out

3） 设置/查看 GPIO 的值

设置输出低

echo "0" > /sys/class/gpio/PB6/value

设置输出高

echo "1" > /sys/class/gpio/PB6/value

查看 GPIO 的值

cat /sys/class/gpio/PB6/value

可以看到 PB6 输出高电平，可以用万用表测量扩展 IO 的 PB6 引脚，可以看到电压为 3.3V 左右。

另外用户如果需要在应用程序控制 GPIO，可以参考一下 wiki：

https://wiki.st.com/stm32mpu/wiki/How_to_control_a_GPIO_in_userspace

5.3.3.2.2. 库函数

从 Linux 4.8 版本开始，Linux 引入了新的 gpio 操作方式，GPIO 字符设备。不再推荐使用以前 SYSFS 方式在"/sys/class/gpio"目录下来操作 GPIO，而是基于"文件描述符"的字符设备，每个 GPIO 组在"/dev"下有一个对应的 gpiochip 文件，例如"/dev/gpiochip0 对应 GPIOA, /dev/gpiochip1 对应 GPIOB"等等。

Libgpiod 库函数实现基于 gpiochip 的方式，提供了一些工具和更简易的 C API 接口。Libgpiod (Library General Purpose Input/Output device) 提供了完整的 API 给开发者，同时还提供了一些用户空间下的应用来操作 GPIO。

Libgpiod 常用基本接口描述：

gpiodetect - 列出系统中出现的所有 gpiochip，它们的名称，标签和 GPIO 行数。

gpioinfo - 列出指定的 gpiochips 的所有行、它们的名称、使用者、方向、活动状态和附加标志。

gpioget -读取指定的 GPIO 行值。

gpioset - 设置指定的 GPIO 行值，潜在地保持这些行导出并等待超时、用户输入或信号。

gpiofind - 查找给定行名称的 gpiochip 名称和行偏移量。

gpiomon - 等待 GPIO 行上的事件，指定要观察哪些事件，退出前要处理多少事件，或者是否应该将事件报告到控制台。

6. 制作文件系统

这是 Linux 系统移植的最后一步，根文件系统构建好以后就意味着我们已经拥有了一个完整的、可以运行的最小系统。以后我们就在这个最小系统上编写、测试 Linux 驱动，移植一些第三方组件，逐步的完善这个最小系统。最终得到一个功能完善、驱动齐全、相对完善的操作系统。

6.1. Buildroot 制作最小系统

6.1.1. 获取源码并编译

将 buildroot 源码 buildroot-2020.02.6.tar.bz2 拷贝到 ubuntu 中，拷贝完成以后对其进行解压。

tar -vxjf buildroot-2020.02.6.tar.bz2

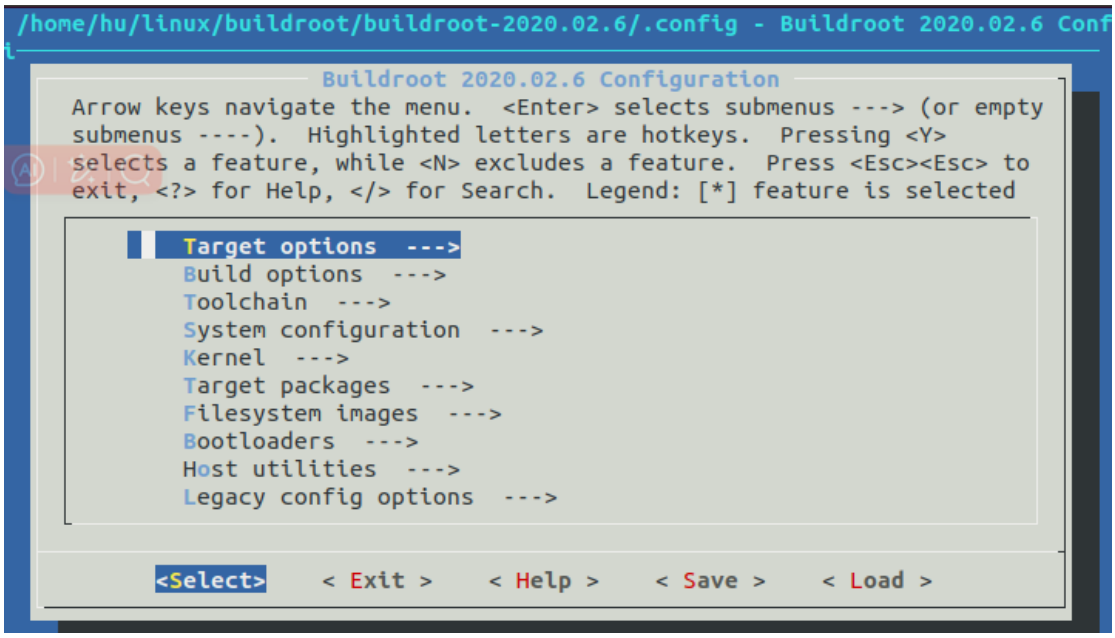
解压后如下图所示

```
hu@ubuntu:~/linux/buildroot$ cd buildroot-2020.02.6/
hu@ubuntu:~/linux/buildroot/buildroot-2020.02.6$ ls
arch              configs     linux            stm32mp1_ebyte_defconfig
board             COPYING     Makefile         stm32mp1_ebyte_defconfig.old
boot              DEVELOPERS  Makefile.legacy  support
CHANGES           dl          output           system
Config.in         docs        package          toolchain
Config.in.legacy  fs          README           utils
```

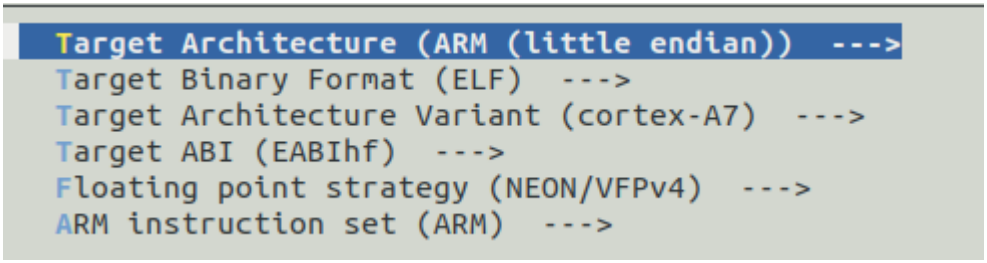
6.1.2. 配置 buildroot

我们使用图形界面配置 buildroot，在根目录下使用如下命令：

make menuconfig



配置 target options，配置结果如下所示。



配置 toolchain，配置结果如下所示

```
    Toolchain type (External toolchain)  --->
    *** Toolchain External Options ***
    Toolchain (Custom toolchain)  --->
    Toolchain origin (Pre-installed toolchain)  --->
(/usr/local/arm/gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf) Toolchain path
($(ARCH)-none-linux-gnueabihf) Toolchain prefix
    External toolchain gcc version (9.x)  --->
    External toolchain kernel headers series (4.20.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
    *** Toolchain Generic Options ***
[ ] Copy gconv libraries
()  Extra toolchain libraries to be copied to target
[*] Enable MMU support
()  Target Optimizations
()  Target linker options
[ ] Register toolchain within Eclipse Buildroot plug-in
```

配置 system configuration

-> System hostname = stm32mp1 //平台名字，自行设置

-> System banner = Welcome to STM32MP135 //欢迎语

-> Init system = BusyBox //使用 busybox

-> /dev management = Dynamic using devtmpfs + mdev //使用 mdev

-> [*] Enable root login with password (NEW) //使能登录密码

-> Root password = 123456 //登录密码为 123456

配置 Filesystem images

-> [*] ext2/3/4 root filesystem //如果是 EMMC 或 SD 卡的话就用 ext3/ext4

-> ext2/3/4 variant = ext4 //选择 ext4 格式

-> exact size =1G //ext4 格式根文件系统 1GB(根据实际情况修改)

-> [*] ubi image containing an ubifs root filesystem //如果使用 NAND 的话就用 ubifs

配置内核支持 UBIFS

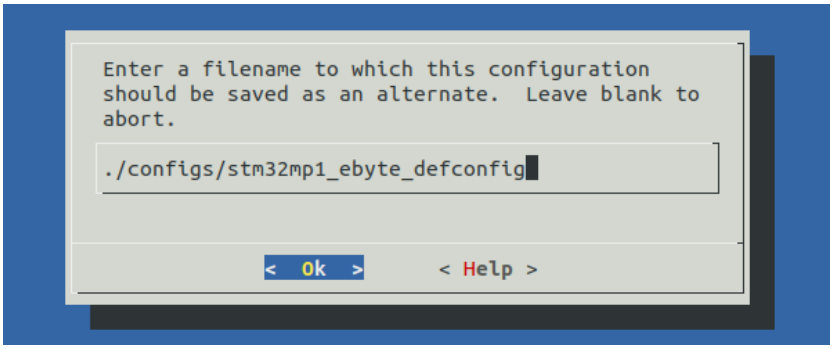
Device Drivers --->Memory Technology Device (MTD) support --->UBI - Unsorted block images --->Enable UBI

配置 mtd 支持 UBI 接口

File systems --->Miscellaneous filesystems --->UBIFS file system support

保存配置项

和 uboot、 kernel 一样，通过图形化界面配置好 buildroot 以后最好保存一下配置项，防止清除工程以后将配置项给删除掉。 buildroot 的默认配置项都保存在 configs 目录下，配置完成以后选择<Save>，然后输入要设置的配置项名字，如图所示：



编译 buildroot

直接使用 make -j8 进行编译即可。

6.2. 使用 ubuntu-base 制作系统

ubuntu-base 是 Ubuntu 官方构建的 ubuntu 最小文件系统，包含 debain 软件包管理器，基础包大小通常只有几十兆，其背后有整个 ubuntu 软件源支持，ubuntu 软件一般稳定性比较好，基于 ubuntu-base 按需安装 Linux 软件，深度可定制等，常用于嵌入式 rootfs 构建。

6.2.1. 获取源码

官网地址：

http://cdimage.ubuntu.com/ubuntu-base/releases/

获取 03_Tools/ubuntu-base-18.04.5-base-armhf.tar.gz 并解压

6.2.2. 准备 chroot 环境

安装模拟器

sudo apt-get install qemu-user-static

sudo cp /usr/bin/qemu-arm-static ./ubuntu-1804/usr/bin/

增加 DNS 配置，后期可直接使用网络更新包

echo "nameserver 8.8.8.8" > ./ubuntu-1804/etc/resolv.conf

制作挂载脚本

将下述脚本拷贝到 ch-mount.sh 文件中，并改变权限为可执行。

```
#!/bin/bash

function mnt() {
        echo "MOUNTING"
        sudo mount -t proc /proc ${2}/proc
        sudo mount -t sysfs /sys ${2}/sys
        sudo mount -o bind /dev ${2}/dev
        sudo chroot ${2}
}
function umnt(){
        echo "UNMOUNTING"
        sudo umount ${2}/proc
        sudo umount ${2}/sys
        sudo umount ${2}/dev
}
if [ "$1" == "-m" ] && [ -n "$2" ] ;
then
        mnt $1 $2
elif [ "$1" == "-u" ] && [ -n "$2" ];
then
        umnt $1 $2
else
        echo ""
        echo "Either 1'st, 2'nd or both parameters were missing"
        echo ""
        echo "1'st parameter can be one of these: -m(mount) OR -u(umount)"
        echo "2'nd parameter is the full path of rootfs directory(with trailing '/')"
        echo ""
        echo "For example: ch-mount -m /media/sdcard/"
        echo ""
        echo 1st parameter : ${1}
        echo 2nd parameter : ${2}
fi
```

6.2.3. 安装包文件

挂载系统

./ch-mount.sh -m ubuntu-1804

```
hu@ubuntu:~/linux/ubuntu$ ./ch-mount.sh -m ubuntu-1804
MOUNTING
[sudo] password for hu:
root@ubuntu:/# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@ubuntu:/#
```

挂载成功即可配置 ubuntu 文件系统与安装一些必要的软件。

基础包安装

apt update

apt install sudo

apt install language-pack-en-base

apt install vim

apt install ssh

apt install net-tools

apt install ethtool

apt install ifupdown

apt install iputils-ping

apt install rsyslog

apt install htop

添加 log,用户调试 ubuntu 系统的调试

#touch /var/log/rsyslog

#chown syslog:adm /var/log/rsyslog

#chmod 666 /var/log/rsyslog

#systemctl unmask rsyslog

#systemctl enable rsyslog

安装网络和语言包支持

#apt-get install synaptic

#apt-get install network-manager network-manager-gnome

#apt-get install language-pack-zh-hant language-pack-zh-hans

#apt-get install rfkill

#apt install -y --force-yes --no-install-recommends fonts-wqy-microhei

#apt install -y --force-yes --no-install-recommends ttf-wqy-zenhei

桌面系统的安装

Xfce4 桌面系统安装

#apt-get install xinit

#apt-get install xfce4

设置 root 密码

passwd root

创建一个用户名为：ebyte

adduser ebyte

设置登录串口

systemctl enable getty@ttySTM0.service

6.2.4. 卸载系统

以上步骤操作完成后即可卸载系统。直接在系统中输入 exit 退出系统，并使用命令来卸载。

```
root@ubuntu:/# exit
exit
hu@ubuntu:~/linux/ubuntu$ ./ch-mount.sh -u ubuntu-1804/
UNMOUNTING
```

7. 制作可烧录到 NandFlash 的镜像文件

确认 Nand 参数，可以通过 datasheet 和 uboot 命令来查询。

mtdlist

ubiinfo

```
UBI: MTD device name:            "UBI"
UBI: MTD device size:            1018 MiB
UBI: physical eraseblock size:   262144 bytes (256 KiB)
UBI: logical eraseblock size:    253952 bytes
UBI: number of good PEBs:        4068
UBI: number of bad PEBs:         4
UBI: smallest flash I/O unit:    4096
UBI: VID header offset:          4096 (aligned 4096)
UBI: data offset:                8192
UBI: max. allowed volumes:       128
UBI: wear-leveling threshold:    4096
UBI: number of internal volumes: 1
UBI: number of user volumes:     4
UBI: available PEBs:             0
UBI: total number of reserved PEBs: 4068
UBI: number of PEBs reserved for bad PEB handling: 76
UBI: max/mean erase counter: 2/0
```

根据 ST 的要求，ubi 镜像包含包含 uboot_config,uboot_config_r,boot.img,rootfs.img。

根据上面获取的参数填写命令

```
mkfs.ubifs -F -v -r /home/hu/linux/ubi_flash/boot -o boot.img -m 4096 -e 253952 -c 4072
mkfs.ubifs -F -v -r /home/hu/linux/ubi_flash/rootfs -o rootfs.img -m 4096 -e 253952 -c 4072
mkfs.ubifs -F -v -r /home/hu/linux/ubi_flash/uboot_config -o uboot_config.img -m 4096 -e 253952 -c 4072
mkfs.ubifs -F -v -r /home/hu/linux/ubi_flash/uboot_config -o uboot_config_r.img -m 4096 -e 253952 -c 4072
```

```
-m, --min-io-size=SIZE   minimum I/O unit size
-e, --leb-size=SIZE      logical erase block size
-c, --max-leb-cnt=COUNT  maximum logical erase block count
-o, --output=FILE        output to FILE
-F, --space-fixup        file-system free space has to be fixed up on first mount
-v, --verbose            verbose operation
-r, -d, --root=DIR       build file system from directory DIR
```

将制作好的各个镜像分成卷放到 ubifs

ubinize -v -o ebyte_135.ubi -m 4096 -p 256KiB -s 4096 ubinize.cfg

```
-o, --output=<file name>     output file name
-v, --verbose                be verbose
-m, --min-io-size=<bytes>    minimum input/output unit size of the flash
                             in bytes
-s, --sub-page-size=<bytes>  minimum input/output unit used for UBI
                             headers, e.g. sub-page size in case of NAND
                             flash (equivalent to the minimum input/output
                             unit size by default)
-p, --peb-size=<bytes>       size of the physical eraseblock of the flash
                             this UBI image is created for in bytes,
                             kilobytes (KiB), or megabytes (MiB)
                             (mandatory parameter)
```

ubinize.cfg 配置文件如下所示。

```
[uboot_config]
mode=ubi
image=uboot_config.img    #mkfs.ubifs生成的文件
vol_id=0           #卷序号
vol_size=4MiB    #卷大小
vol_name=uboot_config    #卷名
vol_type=static

[uboot_config_r]
mode=ubi
image=uboot_config_r.img    #mkfs.ubifs生成的文件
vol_id=1         #卷序号
vol_size=4MiB    #卷大小
vol_name=uboot_config_r    #卷名
vol_type=static

[boot]
mode=ubi
image=boot.img    #mkfs.ubifs生成的文件
vol_id=2           #卷序号
vol_size=12MiB    #卷大小
vol_type=dynamic  #动态卷
vol_alignment=1
vol_name=boot    #卷名

[rootfs]
mode=ubi
image=rootfs.img    #mkfs.ubifs生成的文件
vol_id=3           #卷序号
vol_size=10MiB    #卷大小
vol_type=dynamic  #动态卷
vol_alignment=1
vol_name=rootfs    #卷名
vol_flags=autoresize
```

上述文件的参数，比如卷大小需要根据实际生成的 img 文件大小来设置，一般需要比实际大一些。

8. 参考资料

- Linux kernel 开源社区：

https://www.kernel.org/

- STM32MPU 开发社区：

https://wiki.st.com/stm32mpu/wiki/Development_zone

- STM32MP131 数据手册
- STM32MP135 数据手册

9. 修订说明

修订说明表

版本	修改内容	修改时间	编制	校对	审批
V1.0	初稿	24-06-10	HSL	WYQ	WFX

10. 关于我们

销售热线：4000-330-990
技术支持：support@cdebyte.com 官方网站：https://www.ebyte.com
公司地址：四川省成都市高新西区西区大道 199 号 B5 栋

成都亿佰特电子科技有限公司
Chengdu Ebyte Electronic Technology Co.,Ltd.