Manhattan_platform 编译系统使用指南

北京君正集成电路股份有限公司
Ingenic Semiconductor Co.,Ltd.

Manhattan_platform 编译系统使用指南

Release history

Date	Revision	Revision History
2016.07	V1.0	First released

Disclaimer

This documentation is provided for use with Ingenic products. No license to Ingenic property rights is granted. Ingenic assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement except as provided for by Ingenic Terms and Conditions of Sale.

Ingenic products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. Ingenic may make changes to this document without notice. Anyone relying on this documentation should contact Ingenic for the current documentation and errata.

Ingenic Semiconductor Co., Ltd.

Ingenic Headquarters, East Bldg. 14, Courtyard #10, Xibeiwang East Road, Haidian Dist., Beijing, China, 100193

Tel: 86-10-56345000

Fax: 86-10-56345001

Http: //www.ingenic.cn

目录

1 绪论

1.1 简介

Manhattan 平台是君正开发的一套 linux 系统的发布、开发平台，平台的编译系统支持 c 文件，cpp 文件的编译，同时也支持动态库，静态库以及可执行 bin 文件的编译，Manhattan 平台作为 linux 系统的发布平台，最大的特点就是能够很方便的集成、添加第三方应用、库。

平台的编译系统以指导编译为思想，各个需编译的模块通过 Build.mk 进行指导性编译。工程支持模块的单独编译、选择性编译。极大的方便定制自己的 SDK。

1.2 目录结构介绍

```
├── build          //工程的编译系统所在目录，实现文件系统的整体编译功能
├── development      //包含一些开发中需要的库（本地），服务，以及工具等
│   ├── libutils     //底层工具以及 IPC 接口
│   ├── libcutils    //IPC 接口
│   ├── service      //提供了一些服务，包含休眠唤醒服务、watchdog 等
│   ├── source       //封装了一些硬件接口，上层可以调用，用户也可以参考该
│                      目录下的代码实现自己的接口。
│   └── tools        //包含一些开发中用到的工具，如 gdb，adb 等。
├── device //板级存放目录，包含文件系统公共的部分 common 以及板级自己
│          的 patch
│   ├── common
│   └── phoenix
├── docs    //工程说明文档
│   ├── doc //开发人员编译的使用文档
│   ├── html   //动态生成的 API 文档
├── external    //第三方模块，包括：库和工具
│   ├── alsa-lib
│   ├── alsa-utils
│   ├── e2fsprogs
│   ├── gtest
│   ├── libbacktrace
│   ├── libjpeg-turbo
│   ├── libunwind
│   ├── minigui
│   ├── tinyalsa
├── hardware   //hardware 所在目录，集成了一些操作硬件的接口
│   ├── init
```

```
|   ├── SampleOnTime
|   ├── tests
├── kernel      //kernel 代码
├── Makefile
├── out         //编译生成目录
|   ├── host    //编译生成的 host 端库以及可执行程序
|   └── product    //编译生成的 device 端库以及可执行程序
├── packages      //Sample、App 及测试用例
├── prebuilts     //烧录工具及编译工具链
|   ├── burnertools//烧录工具
|   |   ├── eclipse_install
|   ├── toolchains          //编译工具链
|   |   └── win-tool          //windows 工具
└── u-boot   //u-boot 代码
```

1.3 特别注意

该文档所有内容均以 X1000 平台下的 phoenix 开发板为例进行说明，开发人员可根据自己实际下载到的代码进行参考。

2 工程环境

2.1开发环境要求

系统：Ubuntu14.04 64bit
系统可用空间：50G 以上

2.2工程环境搭建

执行如下命令配置系统的编译环境：
$ source build/envsetup.sh （初始化编译环境）
$ autoenvsetup （搭建工程环境）

3 编译工程

3.1编译整个工程

执行如下命令开始编译整个工程
$ source build/envsetup.sh (编译环境初始化后，此命令可以跳过)
$ lunch
$ make
编译完成后，生成 out 目录,结构如下：

```
├─── host                  //host 端生成文件的存放路径
│    ├─── include          //host 端编译过程所需头文件路径
│    ├─── static           //host 端生成的静态库文件路径
│    │    └─── lib
│    └─── tools            //host 端编译过程中所需工具的存放路径
└─── product
     └─── phoenix          // 产品
          ├─── image       // 生成的目标文件（uboot、kernel、文件系统）
          ├─── include     //target 编译过程中所需头文件的存放路径
          ├─── obj    //模块编译过程中生成的中间文件及目标文件（strip 前）
          ├─── static      //target 端生成的静态库文件（strip 前）
          └─── system      //文件系统（包含 strip 后的模块）
```

注：本编译系统会将生成的可执行文件及库统一 strip 到 system 路径，暂不支持控制单个模块的 strip 动作。

目标文件

$ ls out/product/开发板名/image/

u-boot-with-spl.bin

uImage　　//kernel

System.jffs2 //file system

使用烧录工具“cloner”将上诉三个文件烧入开发板中。

Manhattan 工程的编译系统暂不支持 make　-j 多线程编译的操作，但是你可以修改 device/板级/device.mk 文件中的 MAKE_JLEVEL 的值，提高每个模块的多线程编译，以期加快工程的整体编译速度。

3.2Make clean 工程

执行如下命令：

$ make clean

即可 clean 整个工程。

注：本编译系统暂不支持 make distclean 操作，clean 操作执行的是各个模块在相应 Build.mk 中定义的 clean 动作。

3.3编译工程单个模块

本编译系统支持对单个模块的编译及对单个模块的 clean 操作。工程的编译系统初始化后，存在两套单模块的编译方法，一种是通用的模块编译方法 mma 命令，另外一种是区分 target 与 host 分目标编译方法。

3.3.1 mma 命令

此命令为通用单模块编译方法，不区分此模块为 host 端还是 target 端，此命令的使用方法为：

1、进入到所要编译模块目录下，此模块下含有 Build.mk 文件。

2、执行 mma 命令，此模块以及此模块所依赖的模块都会被编译。

注：执行该操作之前需要在工程主目录下执行 source buiild/envsetup.sh *(编译环境初始化后，此命令可以跳过)*

3.3.2 Target 端模块编译

Target 单个模块编译规则如下，在工程的顶层目录下执行：

$ make $(LOCAL_MODULE)

即 make “模块名”就可单独编译单个模块，以 packages/example/App/grab 下的 grab 测试模块为例，此模块 LOCAL_MODULE（Build.mk 中）赋值为：grab，执行如下命令即可单独编译 grab 模块：

$ make grab

3.3.3 Host 端模块编译

Host 端单模块编译规则如下，在工程的顶层目录下执行：

$ make HOST-$(LOCAL_MODULE)

以 external/e2fsprogs/ 下 e2fsprogs 模块为例，此模块的 LOCAL_MODULE (Build.mk)赋值为：e2fsprogs 所以此模块的单独编译命令如下：

$ make HOST-e2fsprogs

3.3.4 Target 端模块 clean

Target 端模块 clean 规则如下，在工程的顶层目录下执行：

$ make $(LOCAL_MODULE)-clean

还以上述的 grab 模块为例，grab 模块的单独 clean 命令如下：

$ make grab-clean

3.3.5 Host 端模块 clean

Host 端单模块 clean 规则如下，在工程的顶层目录下执行：

$ make HOST-$(LOCAL_MODULE)-clean

还以上述的 e2fsprogs 模块为例，其 clean 命令如下：

$ make HOST-e2fsprogs-clean

3.4单独编译文件系统镜像

单独编译文件系统之所以在此单独阐述，是因为工程的编译系统设计上促使文件系统的编译不在上述介绍的两种方法之中，编译文件系统的方法为，工程目录下执行：

$ make systemimage

就会在 out 目录下重新生成 system 的镜像（前提为之前的 system 镜像已被清除）。

4 基于工程的开发

4.1创建板级

4.1.1 创建板级目录

进入工程 device 目录，然后创建以板级名称命名的目录，以 phoenix 为例。

$ mkdir phoenix

4.1.2 创建板级目录下的存储介质目录

在创建完板级目录以后，需要在板级目录下创建下存储介质目录，如 phoenix 用 norflash 的介质，就创建一个目录叫 norflash。

$ mkdir norflash

4.1.3 创建不同的 model

进入板级存储介质目录下，新建 model1 目录和 model2 目录（至少要有 model1 目录），并且要分别在 model1 目录和 model2 目录下新建一个文件为 vendorsetup.sh。

$ mkdir norflash

$ vim vendorsetup.sh

注：其中 model1 表示的模式为 normal 模式，在命名的时候也可以根据需要命名为其他名称，但是要与 device.mk 中的 MODEL 指定的名称保持一致。model2 表示的模式为 ota 模式，名称也可以根据需要命名

4.1.4 编辑板级 device.mk

创建好板级目录以后，要在板级目录下新建并编辑 device.mk 文件。在编辑板级的 device.mk 时，其中要指定 uboot，kernel 编译路径，kernel 编译默认配置文件，kernel 生成的镜像名，uboot 编译生成的镜像名以及文件系统类型等信息。以下对其中主要的宏进行解释。

UBOOT_BUILD_DIR #需要编译的 uboot 目录

KERNEL_BUILD_DIR #需要编译的 kernel 目录

KERNEL_COFIG_FILE #编译 kernel 时 make munuconfig 所需要的配置文件

MODEL #指定的默认编译 model

- 编译多线程配置：

MAKE_JLEVEL := 4 #配置编译时用几个进程

- 硬件配置：

TARGET_BOOTLOADER_BOARD_NAME #需要编译的 kernel 目录
TARGET_PRODUCT_BOARD
TARGET_BOARD_PLATFORM #该板级属于的平台，如 phoenix 为 X1000
TARGET_BOARD_ARCH #体系结构，mips

- 设备目录文件配置：

UBOOT_BUILD_CONFIG #编译 uboot 需要的配置文件名
KERNEL_BUILD_CONFIG #编译 kernel 需要的配置文件名
KERNEL_TARGET_IMAGE #编译 kernel 的类型，如 uImage
KERNEL_IMAGE_PATH #生成的 kernel 镜像存放目录
UBOOT_TARGET_FILE #生成的 uboot 镜像文件名
FILE_SYSTEM_TYPE #文件系统类型（目前只支持 jffs2，ubi，ext4 这三种文件系统类型）
ROOTFS_JFFS2_NORFLASH_ERASESIZE:= 0x8000 #文件系统 flash 擦除大小
ROOTFS_JFFS2_SIZE:= 0xc80000 #文件系统大小
MODEL #文件系统模式（model1 或者 model2）

➢ 示例一：

本示例为工程 device/phoenix/device.mk

```
PRODUCT:= product
UBOOT_BUILD_DIR := $(TOPDIR)u-boot
KERNEL_BUILD_DIR := $(TOPDIR)kernel
KERNEL_COFIG_FILE := $(KERNEL_BUILD_DIR)/.config
MODEL := model1

#################### make -j config ########################
MAKE_JLEVEL := 4

################ hardware config ############################
TARGET_BOOTLOADER_BOARD_NAME:=$(TARGET_DEVICE)
TARGET_PRODUCT_BOARD:=$(TARGET_DEVICE)
TARGET_BOARD_PLATFORM:= "x1000"
TARGET_BOARD_ARCH:="mips"

################  target_device config ####################
ifeq ($(strip $(TARGET_STORAGE_MEDIUM)),norflash)
#nor flash config
UBOOT_BUILD_CONFIG := phoenix_v10_uImage_sfc_nor
KERNEL_BUILD_CONFIG := phoenix_linux_defconfig
KERNEL_TARGET_IMAGE :=uImage
KERNEL_IMAGE_PATH:=arch/mips/boot

UBOOT_TARGET_FILE:= u-boot-with-spl.bin
```

FILE_SYSTEM_TYPE:=jffs2
ROOTFS_JFFS2_NORFLASH_ERASESIZE:= 0x8000
ROOTFS_JFFS2_SIZE:= 0xc80000

ifeq ($(strip $(TARGET_EXT_SUPPORT)),ota) #OTA
OTA:=y
MODEL:=model2
FILE_SYSTEM_TYPE:=cramfs
UBOOT_BUILD_CONFIG := phoenix_v10_xImage_sfc_nor
KERNEL_IMAGE_PATH := arch/mips/boot/compressed
KERNEL_TARGET_IMAGE := xImage
endif #OTA
endif #norflash

ifeq ($(strip $(TARGET_STORAGE_MEDIUM)),spinand)
#nand flash config
UBOOT_BUILD_CONFIG := phoenix_v10_uImage_sfc_nand
KERNEL_BUILD_CONFIG := phoenix_linux_sfcnand_ubi_defconfig
KERNEL_TARGET_IMAGE :=uImage
KERNEL_IMAGE_PATH:=arch/mips/boot

UBOOT_TARGET_FILE:= u-boot-with-spl.bin

FILE_SYSTEM_TYPE:=ubi
ROOTFS_UBIFS_LEBSIZE := 0x1f000
ROOTFS_UBIFS_MAXLEBCNT := 2048
ROOTFS_UBIFS_MINIOSIZE := 0x800

endif #spinand

4.1.5 新建并编辑板级_base.mk 文件

板级_base.mk 文件中包含的模块是板级所有编译模式（eng，user，userdebug）都需要编译的基本包模块。并且 LOCAL_MODULE_TAGS 为 optional 的模块添加到板级_base.mk 才会被编译（LOCAL_MODULE_TAGS 详情请参考 4.2.1 Lunch 选项详解）。

在 device/板级/目录，如 phoenix_base.mk。把需要添加的编译到产品文件系统的模块名写到变量 PRODUCT_PACKAGES 中。以 phoenix 为例：

- 示例一：

本示例为工程 device/phoenix/phoenix_base.mk

```
RUNTESTDEMO_UTILS := RuntestScript AmicScript WificonnectScript bash CleanfileScript
DmicScript TfcardcopyScript
CAMERA_UTILS := CameraScript cimutils
PRODUCT_MODULES +=    $(RUNTESTDEMO_UTILS)\
                      $(CAMERA_UTILS)\
```

注：该文件中添加的模块必须是 LOCAL_MODULE_TAGS 为 optional 的模块，否则即使在 PRODUCT_MODULES 宏中的模块也不会被编译。

4.2 添加自己的 lunch 操作

4.2.1 Lunch 选项详解

本编译系统提供如下几个规则实现每个模块的选择性编译：

A. lunch 选择编译的模式

a) eng 开发者模式编译

b) user 用户模式编译

c) userdebug 调试模式（此模式下会编译所有属于 eng、user、及 userdebug 模式的模块，且被编译模块可使用 gdb 进行调试，一些调试操作可在此模式下实现）

B. 板级配置文件 device/板级目录/“板级_base.mk”文件，如：device/phoenix/phoenix_base.mk，定义该产品必须编译的模块。

C. 模块的 build.mk 中 LOCAL_MODULE_TAGS 指定当前模块所属的开发编译模式。

a) eng 指定当前模块所属 eng 开发模式，即 lunch 选择 eng 编译时，编译当前模块

b) user 指定当前模块所属 user 开发模式，即 lunch 选择 user 编译时，编译当前模块

c) optional 指定当前模块编译可选，若模块指定为该模式，只有模块在上述 B 项描述的“板级_base.mk”中被包含了，此模块才会被编译。

各个开发模式的关系可概括为如下逻辑：

1. 当 lunch 选择的编译模式为 eng 时，被编译的模块集合为：
   所有 LOCAL_MODULE_TAGS 为 eng 的模块 + 所有 LOCAL_MODULE_TAGS 为 user 的模块 + 板级_base.mk”包含的所有模块
2. 当 lunch 选择的编译模式为 user 时，被编译的模块集合为：
   所有 LOCAL_MODULE_TAGS 为 user 的模块 + 板级_base.mk”包含的所有模块

注：文中所提到的创建板级特指创建 linux 工程的板级，不包含 uboot 以及 kernel 中的板级创建步骤。

4.2.2 Lunch 选项的命名要求

目前的定义格式如下：

$ 板级_存储介质_扩展功能-开发模式

如：*phoenix_norflash_ota-eng*

如若没有特殊扩展功能则扩展功能项可以省略，如：phoenix_norflash-eng

eng	开发者模式编译 adb 默认是可以使用的
user	用户模式编译 默认是没有带 adb 的
userdebug	调试模式（此模式下会编译所有属于 eng、user、及 userdebug 模式的模块，且被编译模块可使用 gdb 进行调试，一些调试操作可在此模式下实现） adb 默认是可以使用的

4.2.3 命名方式与板级目录的关系

Lunch 选项的命名跟板级的目录结构存在一定的关系，还以 phoenix 板级为例，device 下 phoenix 的目录结构如下：

```
Phoenix                          //板级名称（与 lunch 选项保持一致）
    ├──── device.mk              //板级编译的配置文件
    ├──── norflash               // 存储介质（与 lunch 选项保持一致）
    │   ├──── model1             //产品形态
    │   ├──── model2
    │   └──── Readme
    ├──── phoenix_base.mk        //板级模块选择性编译的配置文件
    └──── spinand                // 存储介质（与 lunch 选项保持一致）
        ├──── model1             //产品形态
        └──── vendorsetup.sh //lunch 选项的定义文件
```

Lunch 选项解析后所对应的各个配置的定义在板级的 device.mk 里，如：device/phoenix/device.mk.

4.2.4 添加自己板级 lunch 选项方法

编辑目录板级/存储介质/model/的 vendorsetup.sh，在里面用 add_lunch_combo 关键字添加自己的板级相关的 lunch 选项名称，如 add_lunch_combo phoenix_norflash-eng 就是添加了 phoenix_norflash-eng lunch 选项。还是以板级

phoenix，存储为 norflash 为例。

➢ 示例一：

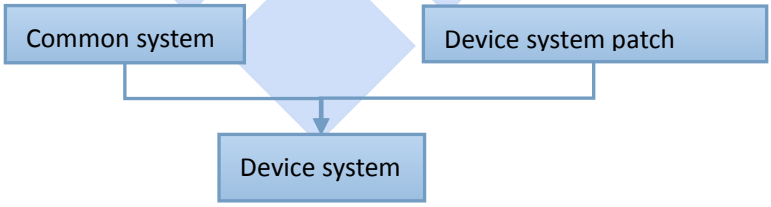
本示例为工程 device/phoenix/norflash/model1/vendorsetup.sh

```
#!/bin/sh

# This file is executed by build/envsetup.sh, and can use anything
# defined in envsetup.sh.
#
# In particular, you can add lunch options with the add_lunch_combo
# function: add_lunch_combo generic-nand-eng

#note:
#          eng              --->      Developer mode
#          userdebug        --->      release mode

add_lunch_combo phoenix_norflash-eng
add_lunch_combo phoenix_norflash-user
add_lunch_combo phoenix_norflash-userdebug
```

4.3 制作自己的文件系统 patch

4.3.1 Patch 制作机制

该工程下的文件系统生成原理如下图所示：



1.model1 模型文件系统的制作机制：

Common system + device system patch ------>> device system

还以 phoenix 板级为例，及工程在编译的过程中利用 device/common/system 下的 common system 及 device/phoenix/norflash/model1/system.patch/ 下的 phoenix model1 产品模型的文件系统 patch 相结合，最终生成 phoenix model1 产品模型自己的文件系统。

2.model2 模型文件系统的制作机制：

还以 phoenix 板级为例，及工程在编译的过程中利用 device/common/system

下 的 common system 及 device/phoenix/norflash/model2/system.patch/ 和 device/phoenix/norflash/model2/system.patch/ 下的 phoenix model2 产品模型的文件系统 patch 相结合，最终生成 phoenix model2 产品模型自己的文件系统。

4.3.2 制作 normal 文件系统 patch

1、首先你必须已经利用 buildroot 或者 busybox 等一些方法制作出了自己的文件系统。我们暂时将这个文件系统的目录定义为 device_system.

2、工程目录下执行 *source build/envsetup.sh；lunch* 选择自己所需的板级及对应的存储选项

3、将 device_system 拷贝到工程的目录下（推荐）。

执行 *$ patch_system* 命令，会出现如下输入提示：

device_system(the system must in current directory --> relative path):

此处输入的为 device_system 的路径（基于当前路径下的相对路径），此项输入后，回车确定，出现另外一条如下输入提示：

which model (the value of 'MODEL' in device.mk) ?:

此项输入为所选择的产品模型，如：model1 或者 model2，此项输入回车确认，即可在对应的“板级”/“存储”/“模型”/下生成 system.patch 目录及 system_patch.sh 文件。如：device/phoenix/norflash/model1/ 下的 system.patch 目录及 system_patch.sh 文件。

4.3.3 制作扩展功能文件系统 patch

1、首先你必须已经利用 buildroot 或者 busybox 等一些方法制作出了自己的两个文件系统，normal 文件系统以及带扩展功能的文件。扩展功能以 ota 为例。我们暂时将扩展功能（ota）文件系统的目录定义为 device_ota_system。

2、工程目录下执行 *source build/envsetup.sh；lunch* 选择自己所需的板级及对应的存储选项

3、将 device_ota_system 拷贝到工程的目录下（推荐）。

device_ext_system(the system which support ext function must in current directory --> relative path):

此处输入的为 device_ota_system 的路径（基于当前路径下的相对路径），此项输入后，回车确定，出现另外一条如下输入提示

which model (the value of 'MODEL' in device.mk) ?:

此项输入为所选择的产品模型，如：model1 或者 model2，此项输入回车确认，即可在对应的“板级”/“存储”/“模型”/下生成 system.patch 目录及 system_patch.sh 文件。如：device/phoenix/norflash/model1/ 下的 system.patch 目录及 system_patch.sh 文件。

4.4 手动创建 Build.mk

4.4.1 添加第三方模块

在添加第三方模块时，首先要确保模块能够正常编译通过，再添加到 platform 上，在添加放置第三方功能模块时，建议放置到 external 目录下，并参照 external 目录下各个的 Build.mk 文件编写所添模块的 Build.mk 文件，对所添模块进行指导性编译，如下几个宏对第三方的编译提供支持。

- 核心编译宏：使用 include 引入如下两个宏中的一个（实为.mk 文件）完成模块的编译工作。

BUILD_HOST_THIRDPART #负责 HOST 端第三方模块的编译，*参见示例一*
BUILD_THIRDPART #负责 DEVICES 端第三方模块的编译，*参见示例二*

- 接口宏：

LOCAL_MODULE	#模块名
LOCAL_MODULE_TAGS	#模块所属开发模式
LOCAL_MODULE_GEN_BINRARY_FILES	#模块产生的可执行程序（需含路径）
LOCAL_MODULE_GEN_STATIC_FILES	#模块产生的静态库文件 （含路径）
LOCAL_MODULE_GEN_SHARED_FILES	#模块产生的动态库文件 （含路径）
LOCAL_MODULE_PATH	#指定目标文件的 copy 路径，不定义此宏目标文件将 copy 到默认路径下①
LOCAL_MODULE_CONFIG_FILES	#模块配置后生成的文件
LOCAL_MODULE_CONFIG	#模块的配置方法
LOCAL_MODULE_COMPILE	#模块的编译方法
LOCAL_MODULE_COMPILE_CLEAN	#模块的 clean 方法
LOCAL_EXPORT_C_INCLUDE_FILES	#模块对外导出的文件，所导出文件被其他模块使用
LOCAL_EXPORT_C_INCLUDE_DIRS	#模块对外导出的目录，所导出目录下的文件被其他模块使用
LOCAL_DEPANNER_MODULES	#本模块的编译依赖的其他模块（生成的库，头文件等），此处为所依赖模块的 module 名，即 **LOCAL_MODULE** 的值。

注：① target 端静态库的路径默认存放在 system 的 usr/lib 下，可执行程序默认存放在 system 的 usr/bin 下。
Host 端默认路径在 out/host 下
其他的参见工程下 build/core/envsetup.mk 文件或者上述的 out 目录介绍。

➢ 示例一：

本示例为工程 external/e2fsprogs/Build.mk

```
LOCAL_PATH := $(my-dir)
include $(CLEAR_VARS)
#vversion e2fsprogs-1.42.9
LOCAL_MODULE=e2fsprogs
LOCAL_MODULE_TAGS:=optional

LOCAL_MODULE_GEN_BINRARY_FILES=e2fsck/e2fsck \
                               misc/tune2fs \
LOCAL_MODULE_CONFIG_FILES:=Makefile
LOCAL_MODULE_CONFIG=./configure
LOCAL_MODULE_COMPILE=make -j$(MAKE_JLEVEL)
LOCAL_MODULE_COMPILE_CLEAN=make distclean
include $(BUILD_HOST_THIRDPART)
```

4.4.2 添加自己的模块

Manhattan 平台支持如下几种类型文件的编译：

- 可执行 bin 文件（host，device）
- 动态库（host，device）
- 静态库（host，device）

各类型模块分别由如下几个宏负责编译：

- 核心编译宏：使用 include 引入如下宏中的一个（实为.mk 文件）完成模块的编译工作

1. 生成可执行 bin 文件

   BUILD_EXECUTABLE （device）

   BUILD_HOST_EXECUTABLE （host）

2. 生成动态库

   BUILD_SHARED_LIBRARY （device）

   BUILD_HOST_SHARED_LIBRARY （host）

3. 生成静态库

   BUILD_STATIC_LIBRARY （device）

   BUILD_HOST_STATIC_LIBRARY （host）

注：可执行文件的编译请参考 packages/example/Sample/下的模块

库的编译请参考 development/service/下的模块

- 接口宏：

LOCAL_MODULE	#模块名
LOCAL_MODULE_TAGS	#模块所属开发模式
LOCAL_SRC_FILES	#源文件
LOCAL_LDLIBS	#链接参数
LOCAL_CFLAGS	#C 编译参数
LOCAL_CPPFLAGS	#CPP 编译参数
LOCAL_DOC_FILES	#指定导出 api 的文件
LOCAL_C_INCLUDES	#编译所需头文件
LOCAL_MODULE_PATH	#指定目标文件的 copy 路径，不定义此宏目标文件 copy 到默认路径下①
LOCAL_SHARED_LIBRARIES	#本模块编译所依赖的其他动态库，此动态库为工程中其他模块产生。
LOCAL_STATIC_LIBRARIES	#本模块编译所依赖的其他静态库，此静态库为工程中其他模块产生。
LOCAL_DEPANNER_MODULES	#本模块的编译依赖的其他模块（生成的库，头文件等），此处为所依赖模块的 module 名，即 **LOCAL_MODULE** 的值。

注：① target 端静态库的路径默认存放在 system 的 usr/lib 下，可执行程序默认存放在 system 的 usr/bin 下。Host 端默认路径在 out/host 下

其他的参见工程下 build/core/envsetup.mk 文件或者上述的 out 目录介绍。

➢ 示例二：

本示例为工程 packages/example/Sample/grab/Build.mk

```
LOCAL_PATH := $(my-dir)
#=================================================
#build grub
include $(CLEAR_VARS)
LOCAL_MODULE=grab
LOCAL_MODULE_TAGS:=optional
LOCAL_SRC_FILES:= camera.c   \
                  grab.c   \
                  savejpeg.c   \
                  v4l2uvc.c
LOCAL_CFLAGS := -Wall -O --static -DVERSION=\"0.1.4\"
LOCAL_LDLIBS := -lc
LOCAL_MODULE_PATH:=$(TARGET_FS_BUILD)/$(TARGET_TESTSUIT_DIR)/$(LOCAL_
MODULE)
```

```
LOCAL_STATIC_LIBRARIES := libjpeg.a
LOCAL_DEPANNER_MODULES:= libjpeg
#depend on the file (basename)
include $(BUILD_EXECUTABLE)
```

➢ 示例三：

本示例为工程 development/source/jpg_api/Build.mk

```
LOCAL_PATH := $(my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE=libjpeg-hw
LOCAL_MODULE_TAGS:=optional

LOCAL_SRC_FILES:= genhead.c    \
                  genyuv.c \
                  hard_en.c    \
                  jpeg_enc.c    \
                  jpeg_encode.c    \
                  jpge_private.c    \
                  jz_mem.c    \
                  jzm_jpeg_enc.c    \
                  soft_en.c    \
                  vpu_common.c
LOCAL_EXPORT_C_INCLUDE_FILES:= include/head.h    \
                               include/ht.h    \
                               include/jpeg.h    \
                               include/jpeg_private.h    \
                               include/jz_mem.h    \
                               include/jzm_jpeg_enc.h \
                               include/qt.h \
                               include/vpu_common.h #export for others to
use
LOCAL_C_INCLUDES := include
LOCAL_CFLAGS := -Wa,-mips32r2 -O2 -G 0 -Wall -fPIC -shared
include $(BUILD_SHARED_LIBRARY)
```

4.4.3 添加复制模块——prebuild

在开发过程中经常需要将一些库和头文件不加任何修改的从一个路径下 copy 到另外一个目标路径下，为此编译系统添加了 prebuild 机制。

- 核心编译宏：使用 include 引入如下宏中的一个（实为.mk 文件）完成模块的 prebuild 工作

BUILD_PREBUILT *//一次 copy 一个文件*
BUILD_MULTI_PREBUILT *//一次 copy 多个文件*

- 接口宏：

LOCAL_MODULE #模块名
LOCAL_MODULE_TAGS #模块所属开发模式
LOCAL_MODULE_CLASS #模块所属哪种 prebuild 模式
LOCAL_MODULE_PATH #文件（夹）copy 的目标路径
LOCAL_COPY_FILES #源文件
LOCAL_MODULE_DIR #源文件夹

Prebuild 的使用，可参考如下示例：

➢ 示例一：

本示例为工程 external/minigui/resource/Build.mk

```
LOCAL_PATH := $(my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := minigui_resource
LOCAL_MODULE_TAGS :=optional
LOCAL_MODULE_CLASS:=DIR
LOCAL_MODULE_PATH := $(OUT_DEVICE_LOCAL_DIR)
LOCAL_MODULE_DIR := share

include $(BUILD_MULTI_PREBUILT)
######################copy Minigui.cfg
include $(CLEAR_VARS)
LOCAL_MODULE := MiniGUI.cfg
LOCAL_MODULE_TAGS :=optional
LOCAL_MODULE_PATH :=$(TARGET_FS_BUILD)/etc
LOCAL_COPY_FILES := MiniGUI.cfg
include $(BUILD_PREBUILT)
```

BUILD_PREBUILT 一次 copy 文件的规则：

LOCAL_COPY_FILES *:=out1:src1*

BUILD_MULTI_PREBUILT 对文件支持一种一次 copy 多个文件的规则：

L***OCAL_COPY_FILES*** *:=out1:src1 out2:src2 out3:src3*

- 示例二：

external/alsa-lib/Build.mk

```
#=================================================
# copy the a alsa.conf
# #
include $(CLEAR_VARS)
LOCAL_MODULE := alsa.conf
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_PATH := $(TARGET_FS_BUILD)/usr/share
LOCAL_COPY_FILES := alsa.conf:./src/conf/alsa.conf
include $(BUILD_PREBUILT)
```

- 示例三：

packages/example/Sample/speech/Build.mk

```
#=================================================
# copy the lib
include $(CLEAR_VARS)
LOCAL_MODULE := aiengine-lib
LOCAL_MODULE_TAGS :=optional
LOCAL_MODULE_PATH :=$(OUT_DEVICE_SHARED_DIR)
LOCAL_PREBUILT_LIBS := libaiengine.so:./lib/libaiengine.so
libecho_wakeup.so:./lib/libecho_wakeup.so libula.so:./lib/x1000-4mic/libula.so
include $(BUILD_MULTI_PREBUILT)
```

4.5 通过命令自动生成 Build.mk

创建好模块目录以后，在该目录下运行 autotouchmk，选择所要添加的模块是 host 端还是 device 端的，然后选择所要添加的模块类型（包括可执行模块、第三方模块、复制模块、静态库模块、动态库模块）。

运行 autotouchmk 命令以后，会有以下提示：

Host module or device module(A:host B:device)：创建 host 端输入 A；device 端输

入 B，不区分大小写，输入其他字符创建失败。
Host 端模块：
然后会弹出如下输入提示：
Please select host target type you want to build (A:execute B:thirdpart C:static library D:shared library):（可输入字符为 A、B、C、D，不区分大小写，输入其他字符创建失败）
Host 端可执行模块：直接输入 A；
Host 端第三方模块：直接输入 B；
Host 端静态库模块：直接输入 C；
Host 端动态库模块：直接输入 D；

Device 端模块：
然后会弹出如下输入提示：
Please select device target type you want to build (A:execute B:thirdpart C:prebuilt D:static library E:shared library):（可输入字符为 A、B、C、D、E、F，不区分大小写，输入其他字符创建失败）
Device 端可执行模块：直接输入 A；
Device 端第三方模块：直接输入 B；
Device 端复制模块：直接输入 C；
Device 端静态库模块：直接输入 D；
Device 端动态库模块：直接输入 E；
Device 端多项复制模块：直接输入 F；

然后会在当前目录下生成一个 Build.mk 文件，自动生成的 Build.mk 会将基本会用的宏都列出来，根据需要对其中的宏进行赋值，不需要的可以不进行修改。自动生成的 Build.mk 文件对于大多数宏都是没有赋值的，需要用户根据自己的需要进行赋值。

```
LOCAL_PATH := $(my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE :=                          #编译生成的模块名称，需要自己填写
LOCAL_MODULE_TAGS := optional            #开发模式默认为 optional，可以修改
LOCAL_SRC_FILES :=                       #要编译的源文件，需要自己添加哪些要
                                         编译
LOCAL_MODULE_PATH := $(TARGET_FS_BUILD)/    #目标文件生成目录
LOCAL_CFLAGS :=                          #编译 C 文件所需编译参数
LOCAL_LDFLAGS :=                         #链接参数
LOCAL_LDLIBS :=                          #链接库

LOCAL_C_INCLUDES :=                      #C 文件所需头文件
LOCAL_CPP_INCLUDES:=                     #CPP 文件所需头文件

LOCAL_DEPANNER_MODULES :=                #该模块编译时依赖的模块名称
LOCAL_COPY_FILES :=                      #prebuild 下需要拷贝的源文件
```

LOCAL_MODULE_DIR :=	#prebuild 下需要拷贝的源目录
LOCAL_MODULE_GEN_SHARED_FILES=	#模块产生的动态库文件 （含路径）
LOCAL_MODULE_GEN_STATIC_FILES =	#模块产生的静态库文件 （含路径）
LOCAL_MODULE_GEN_BINRARY_FILES =	#模块产生的二进制文件 （含路径）
LOCAL_MODULE_CONFIG_FILES :=	#第三方模块的 configure 文件
LOCAL_MODULE_CONFIG :=	#第三方模块的 configure 命令
LOCAL_MODULE_COMPILE =	#第三方模块的编译命令
LOCAL_MODULE_COMPILE_CLEAN =	#第三方模块的清除命令
LOCAL_EXPORT_C_INCLUDE_FILES :=	#模块对外导出的文件，所导出文件被其他模块使用
LOCAL_EXPORT_C_INCLUDE_DIRS :=	#模块对外导出的目录，所导出文件被其他模块使用
include $(BUILD_PREBUILT)	#根据选择自动生成的

注： 该命令仅用于用于生成几种常用模式的模板，具体内容需要用户进行编辑修改。宏的各种含义可参考该文章的该章节前面的内容。

生成的 Build.mk 可参考如下示例：

➢ 示例一：

本示例为为创建 device 端的复制（prebuild）类型

This command is used to generate the commonly used template, specific content need to edit

Host module or device module(A:host B:device):

b

Please select device target type you want to build(A:execute B:thirdpart C:prebuilt D:static library E:shared library F:multi prebuilt):

c

```
LOCAL_PATH := $(my-dir)
include $(CLEAR_VARS)
#name of the module:
LOCAL_MODULE =
LOCAL_MODULE_TAGS := optional                    #Development mode of
module ,userdebug,eng,optional
#which directory the target file copy
LOCAL_MODULE_PATH := $(TARGET_FS_BUILD)/
LOCAL_DEPANNER_MODULES :=                              #depend on module
name
LOCAL_COPY_FILES :=                                    #src copy files

include $(BUILD_PREBUILT)
```

5 SDK 的制作与使用

5.1 SDK 的制作

5.1.1 Linux SDK 的制作

$ source build/envsetup.sh （初始化编译环境）

$ lunch <lunch_name> (选择要编译的板级及模式)

$ make sdk

生成的 SDK 在 out/host/sdk/manhattan-sdk_$(USER)_linux 目录下

5.1.2 Windows SDK 的制作

$ source build/envsetup.sh （初始化编译环境）

$ lunch <lunch_name> (选择要编译的板级及模式)

$ make win_sdk

生成的 SDK 在 out/host/sdk/manhattan-sdk_$(USER)_windows 目录下。

SDK 中安装的库和 Sample 是根据源码中选择的模块进行选择性安装。

5.2 SDK 的使用

5.2.1 SDK 目录结构

```
├─── build                  //工程编译系统所在目录
├─── docs                   //工程中提供的 API 文档
├─── Makefile
├─── platform               //sysroot 目录
├─── prebuilts              //工具链所在目录
├─── Sample                 //实例目录
├─── sdk-build              //编译脚本
└─── source                 //工程开放源码目录
```

5.2.2 SDK 中编译 package

$ sdk-build <package_path>

<package_path>是要编译的 package 在 sdk 中的相对路径。

➢ 示例：

$ sdk-build Sample/speech_tinyalsa

编译后生成的程序在 obj/<package_path>目录下。

6 API 文档生成及使用

在工程中有一个专门存放文档的目录，docs，开发人员可以讲自己写的文档放到 docs/doc 目录下。也可以通过 make doc 命令生成相关 API 帮助文档。

6.1 DOC 目录

Doc 目录下存放的是手动写好的开发类文档，如开发指南，说明文档等。

6.2 html 目录

工程中格式为 html 的 API 帮助文档为动态生成，生成命令为：

$make doc

运行完该命令会在 docs 目录下生成 html 目录供上层开发人员使用。在使用时直接将 docs/html 目录拷贝给上层开发人员即可。阅读方式为用浏览器打开 index.html。

6.2.1 添加自己的 html 帮助文档

编辑需要添加帮助文档的模块 Build.mk，在宏 LOCAL_DOC_FILES 后面添加自己要写成文档的文件。 下面以 development/source/gpio/Build.mk 为例进行说明。该示例中是将 gpio_device.h 的内容写到了 html 帮助文档中。

- 如下示例： development/source/gpio/Build.mk

```
LOCAL_PATH := $(my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := libgpio
LOCAL_MODULE_TAGS := eng
LOCAL_SRC_FILES := gpio_device.cpp
LOCAL_DOC_FILES := gpio_device.h
LOCAL_EXPORT_C_INCLUDE_FILES := gpio_device.h
LOCAL_LDLIBS := -lc -lstdc++
LOCAL_CPPFLAGS := -fPIC
include $(BUILD_SHARED_LIBRARY)
```

注：该方法是基于 docxgen 开发的功能，docxgen 已经在环境安装命令 autoenvsetup 中默认安装。要写到 html 中的文件是有固定格式的，详细的语法以及用法，可以上网参考 docxgen 用法。

6.3 latex 目录

该目录同样是 make doc 命令生成的目录，其内容与 html 一致，只是格式不同。

6.4 ModuleDescription 目录

该目录是在编译（make）时自动生成的，存放用于描述工程中各个模块的功能、添加方式以及使用方式等的文档，目录下另有 html 和 latex 两个目录，分别存放内容相同但格式不同的两种描述文档。

并且在生成该目录的同时在工程的根目录下会生成一个名为 ModuleDescription.html 的链接，可以直接用浏览器打开该链接浏览此文档。