**Manhattan\_platform编译系统使用指南**

|  |
| --- |
|  |

**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 5](#_Toc3800)**

[1.1 简介 5](#_Toc10294)

[1.2 目录结构介绍 5](#_Toc25529)

[1.3 特别注意 6](#_Toc22086)

**[2 工程环境 7](#_Toc7056)**

[2.1 开发环境要求 7](#_Toc4179)

[2.2 工程环境搭建 7](#_Toc20954)

**[3 编译工程 7](#_Toc9410)**

[3.1 编译整个工程 7](#_Toc16577)

[3.2 Make clean工程 8](#_Toc23819)

[3.3 编译工程单个模块 8](#_Toc27662)

[3.3.1 mma命令 8](#_Toc13654)

[3.3.2 Target端模块编译 8](#_Toc20667)

[3.3.3 Host端模块编译 9](#_Toc19010)

[3.3.4 Target端模块clean 9](#_Toc9783)

[3.3.5 Host端模块clean 9](#_Toc30122)

[3.4 单独编译文件系统镜像 9](#_Toc26256)

**[4 基于工程的开发 10](#_Toc6802)**

[4.1 创建板级 10](#_Toc15646)

[4.1.1 创建板级目录 10](#_Toc25013)

[4.1.2 创建板级目录下的存储介质目录 10](#_Toc14187)

[4.1.3 创建不同的model 10](#_Toc29512)

[4.1.4 编辑板级device.mk 10](#_Toc12901)

[4.1.5 新建并编辑板级\_base.mk文件 12](#_Toc24544)

[4.2 添加自己的lunch操作 13](#_Toc3619)

[4.2.1 Lunch选项详解 13](#_Toc14767)

[4.2.2 Lunch选项的命名要求 14](#_Toc17598)

[4.2.3 命名方式与板级目录的关系 14](#_Toc10345)

[4.2.4 添加自己板级lunch选项方法 14](#_Toc9488)

[4.3 制作自己的文件系统patch 15](#_Toc21906)

[4.3.1 Patch制作机制 15](#_Toc14427)

[4.3.2 制作normal文件系统patch 16](#_Toc15366)

[4.3.3 制作扩展功能文件系统patch 16](#_Toc21022)

[4.4 手动创建Build.mk 17](#_Toc31996)

[4.4.1 添加第三方模块 17](#_Toc934)

[4.4.2 添加自己的模块 18](#_Toc11724)

[4.4.3 添加复制模块——prebuild 21](#_Toc14623)

[4.5 通过命令自动生成Build.mk 22](#_Toc27970)

**[5 SDK的制作与使用 25](#_Toc13361)**

[5.1 SDK的制作 25](#_Toc17486)

[5.1.1 Linux SDK的制作 25](#_Toc30822)

[5.1.2 Windows SDK的制作 25](#_Toc13635)

[5.2 SDK的使用 25](#_Toc8176)

[5.2.1 SDK目录结构 25](#_Toc13833)

[5.2.2 SDK中编译package 26](#_Toc913)

**[6 API文档生成及使用 27](#_Toc29935)**

[6.1 DOC目录 27](#_Toc13990)

[6.2 html目录 27](#_Toc1280)

[6.2.1 添加自己的html帮助文档 27](#_Toc8358)

[6.3 latex目录 27](#_Toc13830)

[6.4 ModuleDescription目录 28](#_Toc1949)

# 绪论

## 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。

## 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

## 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。

# 工程环境

## 开发环境要求

系统：Ubuntu14.04 64bit

系统可用空间：50G以上

## 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）*

# 编译工程

## 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。

## Make 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。

## 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。

### 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)*

### 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*

### 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*

### 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*

### 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*

## 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）。

# 基于工程的开发

## 创建板级

### 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*

### 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*

### 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

### 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* |

### 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](#_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。

## 添加自己的lunch操作

### 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：

1. lunch 选择编译的模式
   1. eng 开发者模式编译
   2. user 用户模式编译
   3. 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）
2. 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。
3. 模块的build.mk中LOCAL\_MODULE\_TAGS指定当前模块所属的开发编译模式。
   1. 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
   2. 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
   3. 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

1. 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。

### 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 |

### 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.

### 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*** |

## 制作自己的文件系统patch

### Patch制作机制

该工程下的文件系统生成原理如下图所示：

Common system

Device system patch

Device system

**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。

### 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s ystem\_patch.sh文件。如：device/phoenix/norflash/model1/ 下的system.patch 目录及system\_patch.sh文件。

### 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。

## 手动创建Build.mk

### 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➀

**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：➀ 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****)* |

### 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）

1. 生成动态库

**BUILD\_SHARED\_LIBRARY** （device）

**BUILD\_HOST\_SHARED\_LIBRARY** （host）

1. 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➀

**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：➀ 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****)* |

### 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****)* |

## 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)* |

# SDK的制作与使用

## SDK的制作

### 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

### 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。

## SDK的使用

### 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

### 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。

# 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。

## 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。

## 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。

### 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。

## 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。

## 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。