**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 4](#_Toc1714636915)

[1.1. 简介 4](#_Toc1957747793)

[1.2. 目录结构介绍 4](#_Toc424238335)

[2. 工程环境 5](#_Toc719885386)

[2.1. 开发环境要求 5](#_Toc1649760492)

[2.2. 工程环境搭建 5](#_Toc596516649)

[3. 编译工程 6](#_Toc1189641421)

[3.1. 编译整个工程 6](#_Toc1025202362)

[3.1.1. 编译工程 6](#_Toc1350490027)

[3.1.2. clean工程 7](#_Toc783368690)

[3.2. 编译单模块 7](#_Toc1102520059)

[3.2.1. 编译单模块 7](#_Toc2044897763)

[3.2.2. 编译uboot 8](#_Toc1967513926)

[3.2.3. 编译kernel 9](#_Toc1365180540)

[3.2.4. 编译buildroot 9](#_Toc1540383426)

[3.2.5. 编译host端工具 10](#_Toc304089172)

[3.3. 编译文件系统镜像 10](#_Toc1303455736)

[4. 新板级创建 11](#_Toc35005211)

[4.1. lunch操作 11](#_Toc521595368)

[4.1.1. lunch选项的命名要求 12](#_Toc294702567)

[4.1.2. lunch选项详解 12](#_Toc1726956429)

[4.2. 创建板级配置文件 13](#_Toc336465782)

[4.3. 命名方式与板级目录的关系 16](#_Toc861021530)

[5. 新模块添加 16](#_Toc278722862)

[5.1. Makefile编译的模块 16](#_Toc233665123)

[5.2. 第三方自定义编译的模块 19](#_Toc2145174067)

[5.3. cmake编译的模块 21](#_Toc468703135)

[5.4. qmake编译的模块 22](#_Toc1101513929)

[5.5. 复制模块 25](#_Toc1801979802)

[5.6. 自动生成Build.mk 27](#_Toc1315634022)

[6. API文档生成及使用 30](#_Toc635723058)

[6.1. 文档目录介绍 30](#_Toc1369133069)

[6.2. 添加html帮助文档 31](#_Toc1125898167)

# 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

Manhattan平台目录结构和内容分布如下：

.

├── 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

│   ├── assets

│   ├── core //包括uboot、kernel、buildroot、cmake、qmake以及第三方库等编译规则

│   └── doc

├── buildroot //buildroot代码，制作基础文件系统

├── development //包含一些开发中需要的库（本地），服务，以及工具等

│   ├── libmcu-bare

│   ├── libosd

│   ├── libsynchronous-frames

│   └── mcu-host

├── device //板级存放目录

│   └── halley5

├── docs //工程说明文档

│   └── doc

├── external //第三方模块，包括：库和工具

│   ├── android

│   ├── pytorch

│   └── tinyalsa

├── frameworks

│   └── ai

├── kernel-4.4.94 //kernel代码

├── kernel-5.10 //kernel代码

├── out //编译生成目录

│   └── product

│ └── halley5.v10\_nand\_4.4.94-eng

│ ├── host //编译生成的host端库以及可执行程序

│ ├── image

│ ├── obj

│ ├── sysroot

│ └── system //编译生成的device端库以及可执行程序

├── packages //Sample、App及测试用例

│   ├── example

│   └── updater

├── prebuilts //烧录工具及编译工具链

│   ├── burnertools //烧录工具

│   ├── sd-burntools

│   └── toolchains //编译工具链

└── u-boot //u-boot代码

注：该文档所有内容均以X2000平台下的halley5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

### 编译工程

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或$ make -j32

编译完成后，生成out目录，结构如下：

.

└── product

└── halley5.v10\_nand\_4.4.94-eng

├── host //host端生成文件的存放路径

├── image // 生成的目标文件（uboot、kernel、文件系统）

│   ├── kernel

│   ├── system.ubifs

│   └── uboot

├── obj //模块编译过程中生成的中间文件及目标文件（strip前）

│   ├── buildroot-intermediate

│   ├── DEPANNER

│   ├── development

│   ├── kernel-intermediate

│   ├── packages

│   └── uboot-intermediate

├── sysroot

└── 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。

编译后在工程*out/product/开发板名/image/目录下生成*u-boot-with-spl.bin、kernel和system.ubifs三个镜像，使用烧录工具“cloner”将下面三个文件烧入开发板中。

### clean工程

执行如下命令，可clean整个工程：

$ make clean

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。

## 编译单模块

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。

### 编译单模块

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，另外一种是目标编译方法。

* mma命令

此命令为通用单模块编译方法，使用方法如下：

进入到所要编译模块目录下，此模块下含有Build.mk文件。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模块编译

在工程的顶层目录下执行如下命令，实现Target单个模块编译：

$ 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

在工程的顶层目录下执行如下命令，实现Target端模块clean：

$ 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

### 编译uboot

在工程的顶层目录下执行如下命令，实现uboot编译：

$ make uboot或make uboot -j32

在工程的顶层目录下执行如下命令，实现uboot clean：

$ make uboot-clean

### 编译kernel

在工程的顶层目录下执行如下命令，可实现kernel编译：

$ make kernel或make kernel -j32

在工程的顶层目录下执行如下命令，可实现kernel clean：

$ make kernel-clean

kernel默认的编译配置在device/halley5/kernel-4.4.94.mk中指定，查看和修改kernel编译配置规则，可在工程的顶层目录下执行如下命令：

$ make kernel-menuconfig

$ make kernel或make kernel -j32

### 编译buildroot

工程用buildroot制作基础文件系统，在工程的顶层目录下执行如下命令，可实现buildroot编译：

$ make buildroot或make buildroot -j32

在工程的顶层目录下执行如下命令，实现buildroot clean：

$ make buildroot-clean

buildroot默认的编译配置在device/halley5/device.mk中指定，在工程的顶层目录下执行如下命令，可查看和修改buildroot编译配置：

$ make buildroot-menuconfig

$ make buildroot或$ make buildroot -j32

注：编译完buildroot后，需要重新执行make命令将其他模块统一编译到文件系统中。

### 编译host端工具

host端模块编译有两种方法，一种是通过配置buildroot进行编译，在工程的顶层目录下执行如下命令修改buildroot编译配置，选择需要编译的host端工具：

$ make buildroot-menuconfig

Host utilities --->

选择后，执行如下命令，重新编译文件系统。

$ make buildroot

$make

一般情况下，buildroot提供的host端工具可以满足文件系统要求，如果需要添加，参考buildroot编译规则添加。

另一种host端编译不依赖于buildroot，在工程的顶层目录下执行如下命令，实现host端单模块编译:

$ make HOST-$(LOCAL\_MODULE)

在工程的顶层目录下执行如下命令，实现host端单模块clean:

$ make HOST-$(LOCAL\_MODULE)-clean

## 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方法之中，工程目录下执行如下命令，可编译文件系统：

$ make systemimage

执行上述命令后，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）。

# 新板级创建

manhattan板级相关所有信息都在device下一个目录统一进行管理，首先需要进入工程device目录，创建以板级名称命名的目录，以halley5为例。

$cd device

$ mkdir halley5

## lunch操作

进入板级目录下，新建vendorsetup.sh文件，添加板级信息，如示例一。

$ cd halley5

$ vim vendorsetup.sh

示例一：

本示例为工程device/halley5/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 halley5.v10\_nand\_4.4.94-eng***  ***add\_lunch\_combo halley5.v10\_nand\_4.4.94-user***  ***add\_lunch\_combo halley5.v10\_nand\_4.4.94-userdebug*** |

注：*add\_lunch\_combo halley5.v10\_nand\_4.4.94-eng*命名规则有要求，执行lunch命令后编译系统会根据上面的板级信息给编译的环境变量赋值（[具体命名规则详情请参考lunch选项的命名要求](#_lunch选项的命名要求)）。

### lunch选项的命名要求

目前的定义格式如下：

$ 板级.版本\_存储介质\_扩展功能-开发模式

如：halley5.v10\_nand\_4.4.94-eng

### lunch选项详解

本编译系统提供如下几种模式实现模块的选择性编译，具体使用哪种模式通过上面的lunch设定：

|  |  |
| --- | --- |
| 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 |

每个模块所属的开发编译模式在模块的build.mk中LOCAL\_MODULE\_TAGS选择指定。模块选择性编译规则如下：

|  |  |
| --- | --- |
| LOCAL\_MODULE\_TAGS | 编译 |
| 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 |
| 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 |
| 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“板级\_base.mk”中被包含了，此模块才会被编译。  *注：板级配置文件device/板级目录/“板级\_base.mk”如： device/halley5/halley5\_base.mk，定义该产品必须编译的模块。* |

各个开发模式的关系可概括为如下逻辑：

|  |  |
| --- | --- |
| lunch选择的编译模式 | 编译模块集合 |
| eng | 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 |
| user | 所有LOCAL\_MODULE\_TAGS 为user的模块+ 板级\_base.mk包含的所有模块 |

## 创建板级配置文件

* 创建device.mk

在板级目录下新建device.mk文件。device.mk文件中要指定板级相关信息，以下对其中主要的宏进行解释。

* 编译变量设置相关

|  |  |
| --- | --- |
| PRODUCT | 指定产品名称，用于设置out下install路径 |
| MAKE\_JLEVEL | 默认的编译多线程配置 |
| LOCAL\_PATH | 相对于工程的当前路径 |
| DEVICE\_PATH | 板级目录（绝对路径） |

* 硬件配置相关

|  |  |
| --- | --- |
| TARGET\_PRODUCT\_BOARD | 板级名称 |
| TARGET\_BOARD\_PLATFORM | 板级所属平台 |
| TARGET\_BOARD\_ARCH | 体系结构 |
| TARGET\_ARCH\_VARIANT | 体系结构相关变量选择，目前两种xburst2设置mips32r2\_fp64，xburst设置mips32r2 |

* 文件系统相关编译配置

|  |  |
| --- | --- |
| BUILDROOT\_CONFIG | buildroot配置文件 |
| BUILDROOT\_CONFIG\_PATH | buildroot配置文件路径 |
| BUILDROOT\_PATH | buildroot路径 |
| ROOTFS\_OVERLAY\_DIR | 在该路径下存放需要添加或替换的文件系统的文件 |
| FILE\_SYSTEM\_TYPE | system镜像类型 |

* 创建uboot.mk

在板级目录下新建并编辑uboot.mk文件,以device/halley5/uboot.mk为例，对各个宏定义进行解释。

|  |  |
| --- | --- |
| LOCAL\_PATH :=$(my-dir) | 相对于工程根目录的当前位置 |
| include $(CLEAR\_VARS) | 初始化编译环境变量 |
| UBOOT\_BUILD\_CONFIG:=halley5\_xImage\_sfc\_nand | uboot配置文件 |
| UBOOT\_TARGET\_FILE:=u-boot-with-spl.bin | uboot编译目标文件 |
| LOCAL\_MODULE := uboot | 模块名称 |
| UBOOT\_PATH := $(TOP\_DIR)/u-boot | uboot源码路径 |
| LOCAL\_MODULE\_TAGS := optional | 指定当前模块所属的开发编译模式 |
| include $(BUILD\_UBOOT) | uboot编译规则 |

* 创建kernel.mk

在板级目录下新建并编辑kernel.mk文件,以device/halley5/kernel-4.4.94.mk为例，对各个宏定义进行解释。

|  |  |
| --- | --- |
| include $(CLEAR\_VARS) | 初始化编译环境变量 |
| LOCAL\_MODULE=kernel | 模块名称 |
| KERNEL\_PATH=$(TOP\_DIR)/kernel-4.4.94 | kernel源码路径 |
| KERNEL\_CONFIG\_PATH:=$(KERNEL\_PATH)/arch/mips/configs | kernel配置文件路径 |
| LOCAL\_MODULE\_TAGS :=optional | 指定当前模块所属的开发编译模式 |
| KERNEL\_TARGET\_IMAGE:=xImage | kernel编译目录镜像名称 |
| KERNEL\_IMAGE\_PATH:=arch/mips/boot/compressed/ | kernel目标镜像路径 |
| KERNEL\_BUILD\_CONFIG:=halley5\_v20\_linux\_sfc\_nand\_defconfig | kernel配置文件 |
| include $(BUILD\_KERNEL) | kernel编译规则 |

* 创建“板级\_base.mk”

“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](#_lunch选项详解)详情请参考[lunch选项详解](#_lunch选项详解)）。

在device/板级/目录，如halley5\_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halley5为例：

示例二：

本示例为工程device/halley5/halley5\_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。

## 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halley5板级为例，device下halley5的目录结构如下：

.

└── halley5 //板级名称（与lunch选项保持一致）

├── Build.mk

├── device.mk //板级编译的配置文件

├── halley5\_base.mk //板级模块选择性编译的配置文件（板级名称与lunch选项保持一致）

├── kernel-4.4.94.mk //kernel-4.4.94编译的配置文件

├── rootfs-overlay //文件系统需要覆盖或添加的文件目录（该目录路径在device.mk中设置）

├── uboot.mk //uboot编译的配置文件

└── vendorsetup.sh //lunch 选项的定义文件

# 新模块添加

## Makefile编译的模块

Manhattan平台支持编译文件的类型和对应的核心编译宏如下：

|  |  |
| --- | --- |
| 可执行bin文件 | BUILD\_EXECUTABLE （device）  BUILD\_HOST\_EXECUTABLE （host） |
| 动态库 | BUILD\_SHARED\_LIBRARY （device）  BUILD\_HOST\_SHARED\_LIBRARY （host） |
| 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。

模块Build.mk对应的接口宏解释如下：

|  |  |
| --- | --- |
| 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 |
| 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，此处为所依赖模块的module名，即LOCAL\_MODULE的值 |

注：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/product/板级/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。

示例三：

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****)* |

示例四：

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****)* |

## 第三方自定义编译的模块

在添加第三方自定义编译的模块时，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 |
| BUILD\_THIRDPART | 负责DEVICES端第三方模块的编译 |

模块Build.mk对应的接口宏解释如下：

|  |  |
| --- | --- |
| 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 |
| 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，此处为所依赖模块的module名，即LOCAL\_MODULE的值 |

注：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/product/板级/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。

示例五：

本示例为工程external/minigui/minigui-gpl/Build.mk

|  |
| --- |
| ***LOCAL\_PATH*** *:= $(my-dir)*  *include $(CLEAR\_VARS)*  ***LOCAL\_MODULE****=minigui\_ths*  ***LOCAL\_MODULE\_TAGS*** *:=optional*  ***LOCAL\_MODULE\_GEN\_SHARED\_FILES****= src/.libs/libminigui\_ths.so \*  *src/.libs/libminigui\_ths-3.0.so.12 \*  *src/.libs/libminigui\_ths-3.0.so.12.0.0*  ***LOCAL\_MODULE\_CONFIG\_FILES****:= config.log*  ***LOCAL\_MODULE\_CONFIG****:=./configure --enable-shared --host=$(DEVICE\_COMPILER\_PREFIX) \*  *LDFLAGS=-L$(ABS\_DEVICE\_SHARED\_DIR) \*  *CFLAGS="-I$(ABS\_DEVICE\_INCLUDE\_DIR) -std=gnu89" \*  *LIBS="-ljpeg -lpng -lz"*  ***LOCAL\_MODULE\_COMPILE****=make -j$(MAKE\_JLEVEL)*  ***LOCAL\_MODULE\_COMPILE\_CLEAN****=make distclean*  ***LOCAL\_DEPANNER\_MODULES****:=libjpeg libpng*  ***LOCAL\_EXPORT\_C\_INCLUDE\_FILES****:= include/common.h \*  *include/minigui.h \*  *include/gdi.h \*  *include/window.h \*  *include/endianrw.h \*  *mgconfig.h*  *include $(****BUILD\_THIRDPART****)* |

## cmake编译的模块

在添加cmake编译的模块时，首先要确保模块能够正常编译通过，再添加到platform上，该模块建议放置到packages目录下，并创建Build.mk文件，对所添模块进行指导性编译，如下几个宏对cmake译提供支持。

核心编译宏：使用include引入如下宏完成模块的编译工作。

|  |  |
| --- | --- |
| BUILD\_CMAKE\_DEVICE | 负责DEVICES端cmake模块的编译（实为build\_cmake.mk文件） |

模块Build.mk对应的接口宏解释如下：

|  |  |
| --- | --- |
| LOCAL\_MODULE | 模块名 |
| CMAKE\_PATH | 模块位置路径 |
| CMAKE\_CONF\_OPTS | 添加cmake option |
| 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 |
| LOCAL\_MODULE\_TAGS | 模块所属开发模式 |

示例六：

本示例为工程packages/apps/network/Build.mk

|  |
| --- |
| ***LOCAL\_PATH*** *:= $(my-dir)*  *include $(CLEAR\_VARS)*  ***CMAKE\_PATH****=$(LOCAL\_PATH)/network-manager-app*  ***LOCAL\_MODULE****:=network\_manager*  ***CMAKE\_CONF\_OPTS****:=-DBUILD\_libNetworkConnectionManager=ON*  ***LOCAL\_DEPANNER\_MODULES****=debug\_syslog systemserver*  ***LOCAL\_MODULE\_TAGS*** *:=optional*  *include $(****BUILD\_CMAKE\_DEVICE****)* |

## qmake编译的模块

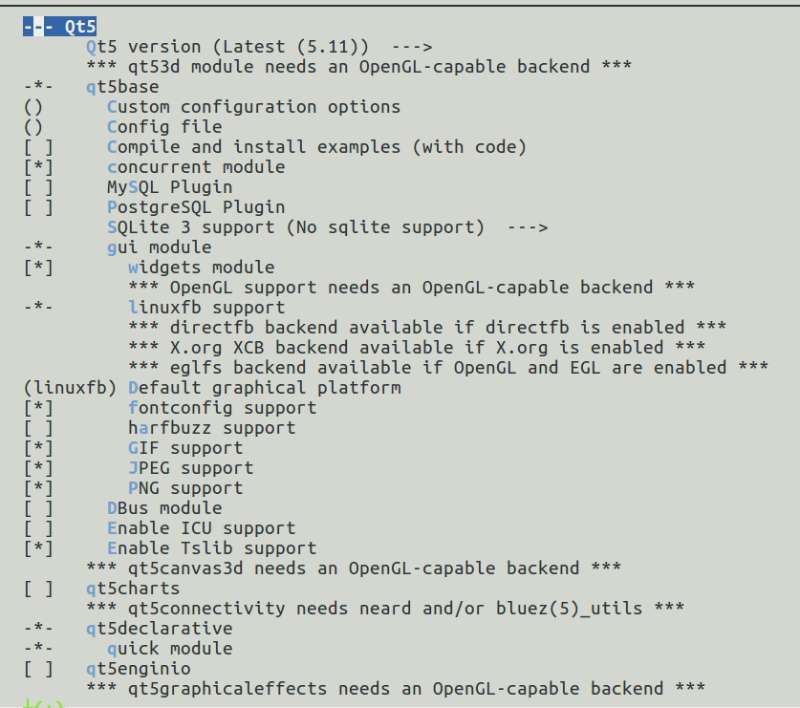
要加入qmake模块，首先需要搭建qt环境，执行下面命令编译qt相关库和工具：

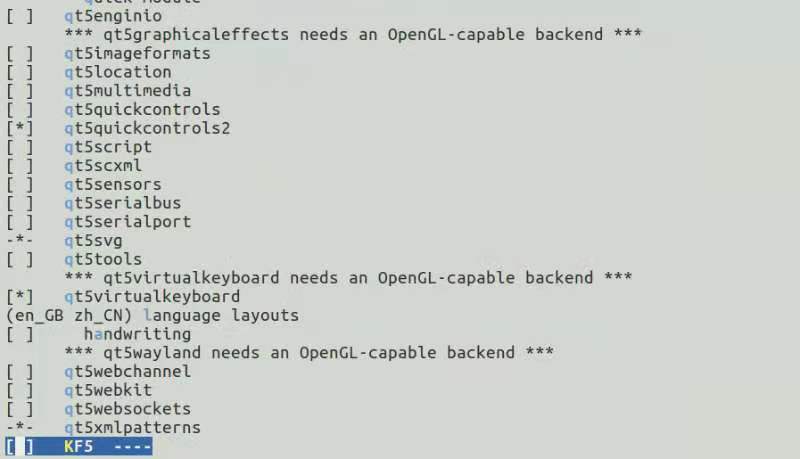
$ make buildroot-menuconfig（配置buildroot）

选择

|  |
| --- |
| Target packages --->  Graphic libraries and applications (graphic/text) --->  [\*] Qt5 ---> |

根据需求配置qt5相关选项。下面是配置示例：





然后要确保qmake模块在pc端用qtcreator能够正常编译通过，再添加到platform上。该模块建议放置到packages目录下，修改对应的.pro文件，修改生成镜像的位置如下：

unix:!android: target.path = $$[QT\_INSTALL\_EXAMPLES]/gui

创建Build.mk文件，对所添模块进行指导性编译，如下几个宏对qmake编译提供支持。

核心编译宏：使用include引入如下宏完成模块的编译工作。

|  |  |
| --- | --- |
| BUILD\_QMAKE\_DEVICE | 负责DEVICES端qmake模块的编译（实为build\_qmake.mk文件） |

模块Build.mk对应的接口宏解释如下：

|  |  |
| --- | --- |
| LOCAL\_MODULE | 模块名 |
| QMAKE\_PATH | 模块位置路径 |
| 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 |
| LOCAL\_MODULE\_TAGS | 模块所属开发模式 |
| QTARGET\_COPYS | 生成的目标镜像拷贝到system的位置（要输入目标镜像和拷贝的位置，目前只支持一个镜像的拷贝） |

注：目标镜像在工程中的位置设定在.pro文件中，其中QT\_INSTALL\_EXAMPLES为out/product/板级/sysroot/usr/lib/qt/examples。

示例七：

本示例为工程packages/kunpeng/testqt/Build.mk

|  |
| --- |
| *LOCAL\_PATH := $(my-dir)*  *include $(CLEAR\_VARS)*  ***QMAKE\_PATH****=$(LOCAL\_PATH)/qtmain*  ***LOCAL\_MODULE****:= qtmain*  ***LOCAL\_MODULE\_TAGS*** *:=optional*  ***QTARGET\_COPYS****=usr/lib/qt/examples/gui/qtmain:usr/bin*  *include $(****BUILD\_QMAKE\_DEVICE****)* |

## 复制模块

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 |

模块Build.mk对应的接口宏解释如下：

|  |  |
| --- | --- |
| 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：

示例八：

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****)* |

注：

**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*

示例九：

本示例为工程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****)* |

示例十：

本示例为工程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****)* |

## 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):

Input Module :

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。

* 选择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。各选项解释如下：

|  |  |
| --- | --- |
| A | Host端可执行模块 |
| B | Host端第三方模块 |
| C | Host端静态库模块 |
| D | Host端动态库模块 |

* 选择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:static library

D:shared library

E:multi prebuilt

F:multi copy

Input module :

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。各选项解释如下：

|  |  |
| --- | --- |
| A | Device端可执行模块 |
| B | Device端第三方自定义编译模块 |
| C | Device端静态库模块 |
| D | Device端动态库模块 |
| E | Device端复制模块 |
| F | Device端多项复制模块 |

执行完以上操作，会在当前目录下生成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。

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。

生成的Build.mk可参考如下示例：

示例十一：

本示例为创建device端的编译可执行程序，创建/packages/example/App/test目录，在该目录下执行如下命令：

$ autotouchmk

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

Input Module :B

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Please select device target type you want to build

A:execute

B:thirdpart

C:static library

D:shared library

E:multi prebuilt

F:multi copy

Input module :A

生成的Build.mk如下：

|  |
| --- |
| *LOCAL\_PATH := $(my-dir)*  *include $(CLEAR\_VARS)*  *LOCAL\_MODULE = #name of the module:*  *LOCAL\_MODULE\_TAGS := optional #Development mode of module ,userdebug,eng,optional*  *LOCAL\_MODULE\_PATH := $(TARGET\_FS\_BUILD)/ #which directory the target file copy*  *LOCAL\_SRC\_FILES := #compiled src files*  *LOCAL\_CFLAGS := #C compile parameters*  *LOCAL\_LDFLAGS := #Link parameters*  *LOCAL\_LDLIBS := #Link parameters*  *LOCAL\_C\_INCLUDES := #include files*  *LOCAL\_CPP\_INCLUDES:= #cpp include files*  *LOCAL\_MODULE\_GEN\_BINRARY\_FILES := #The binrary file module*  *LOCAL\_EXPORT\_C\_INCLUDE\_FILES := #export included files*  *LOCAL\_EXPORT\_C\_INCLUDE\_DIRS := #export included dir*  *LOCAL\_SHARE\_LIBRARIES:= #depend shared library*  *LOCAL\_DEPANNER\_MODULES := #depend on module name*  *include $(BUILD\_EXECUTABLE)* |

# API文档生成及使用

## 文档目录介绍

在工程中有一个专门存放文档的目录docs，开发人员可以将自己写的文档放到docs/doc目录下。也可以通过make doc命令生成相关API帮助文档（该命令生成文档需要依赖代码中固定的注释格式，目前库里代码有待完善代码注释）。

* 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。

* 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。

## 添加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。

示例十二：

本示例为工程 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。