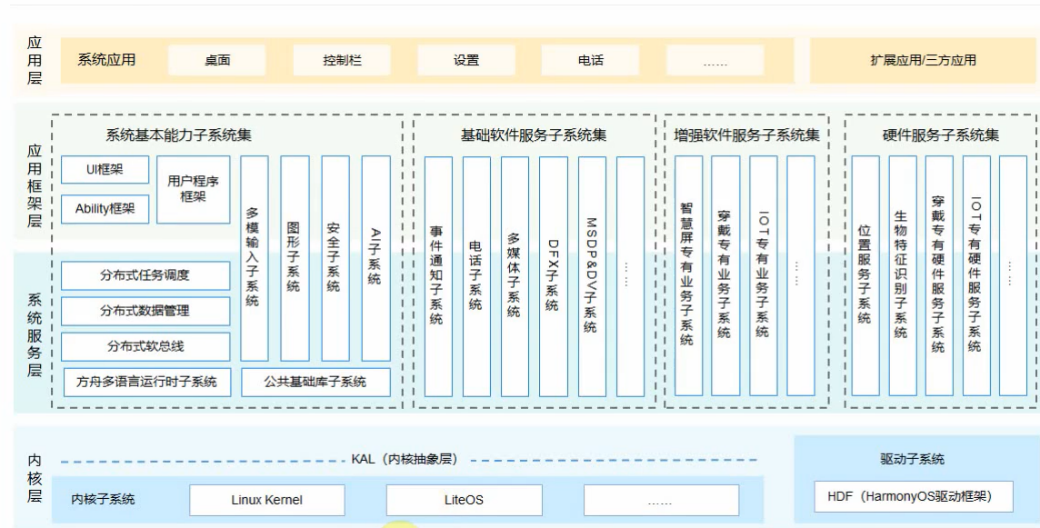
# 实验一环境搭建与鸿蒙LiteOS-a内核体验

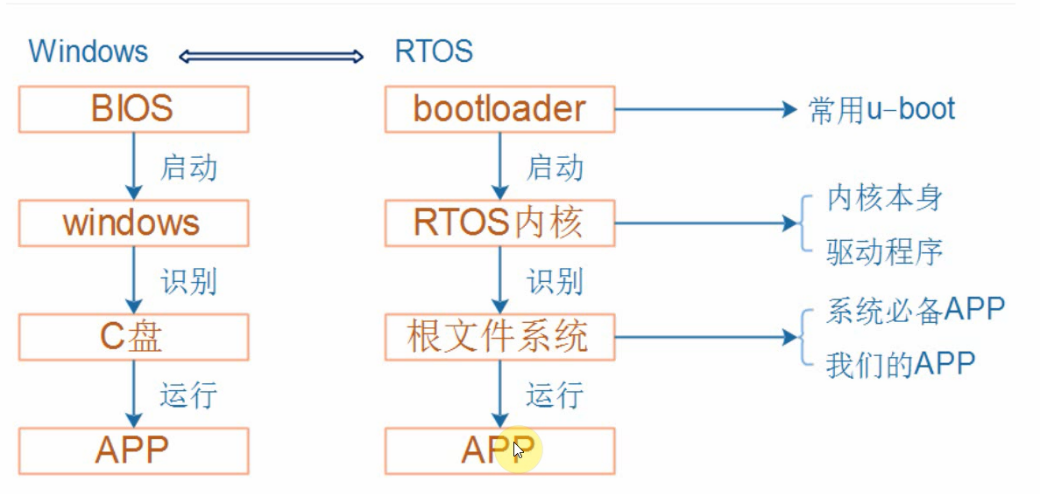
1. 实验目的
2. 实验内容

# 1.框架



鸿蒙是一套完整的、普通人可以直接使用的操作系统，跟Windows、安卓、IOS类似。  
 常见的错误观点是把鸿蒙跟Linux放在一起来对比，这不对：

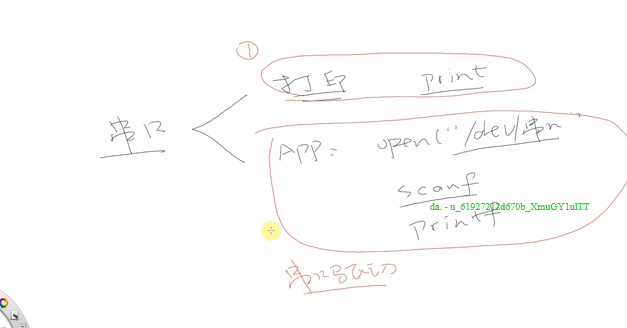
* Linux只是一个内核，普通人无法使用
* 还需要在Linux之上安装各类程序
* 比如Ubuntu等发行版，它们在Linux内核之上，还有桌面、各类办公软件
* 鸿蒙支持多种内核：Linux、Liteos(又分为Liteos-a、Liteos-m)
* 在内核之上，鸿蒙还有各种子系统，在子系统之上又有桌面等软件



我们可以把内核之上的软件，简单称为APP(实际上还可以细分，比如各类子系统、桌面等APP)。  
 启动内核，并不复杂，使用U-boot即可。

**串口相关**

* 打印(只是打印调试信息)
* 串口驱动(可发可收，APP执行printf时可以从串口打印，所以需要**驱动**)



* MMU(Memory Management Unit，内存管理单元)的设置：虚拟地址与物理地址

**完善中断子系统**

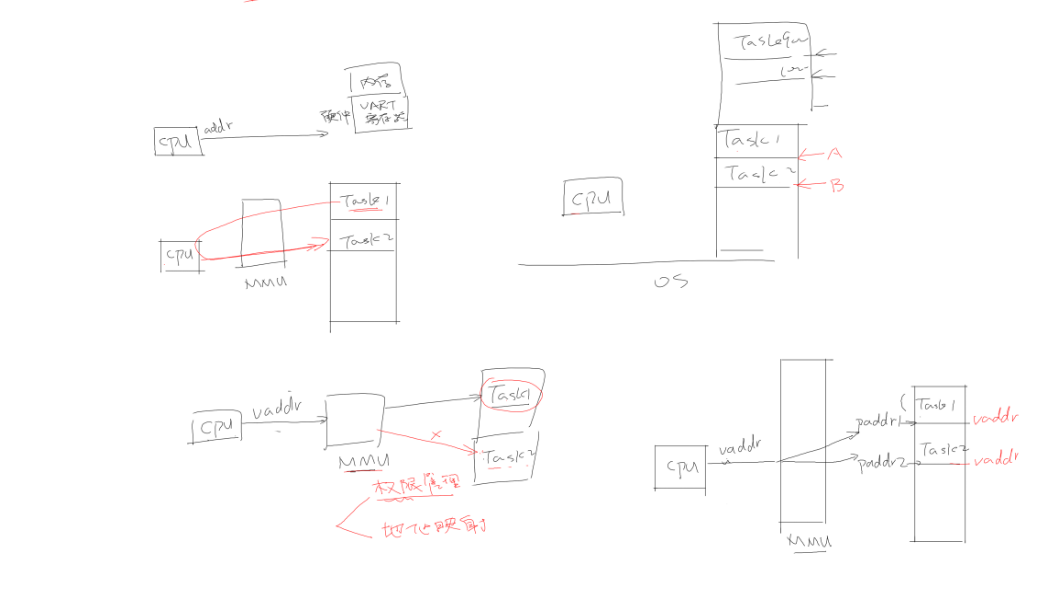
* 提供系统tick时钟
* 为串口驱动实现基于中断的读取字符函数

* 实现存储设备驱动程序
* 在存储设备上烧录文件系统

## 3.1 串口相关

与Linux的串口驱动相比，鸿蒙的串口驱动极大简化了。  
 对于输出：不使用中断，直接使用查询方式输出。  
 对于输入：使用中断，我们只需要提供底层硬件相关的代码。  
 要注意：使用的是虚拟地址。

MMU有2大功能：



### 3.2.1 权限管理

* 比如可以把进程A、B的地址空间完全隔离开，它们互不影响
* 写得差的进程、有恶意的进程，不能影响到其他进程
* 用户程序、内核地址空间完全隔离开：不允许用户直接访问硬件

### 3.2.2 地址映射

* 使能MMU后，CPU发出的地址被称为"虚拟地址"，它不是直接发送给硬件，而是发给MMU
* MMU根据页表

* 进行权限判定
* 转换为物理地址，发给外设

运行app1时，CPU发出的addr，通过MMU映射到paddr1；  
 运行app2时，CPU发出的同一个addr，通过MMU映射到paddr2；  
 虽然app1、app2使用的地址相同，但是对应的内存不同

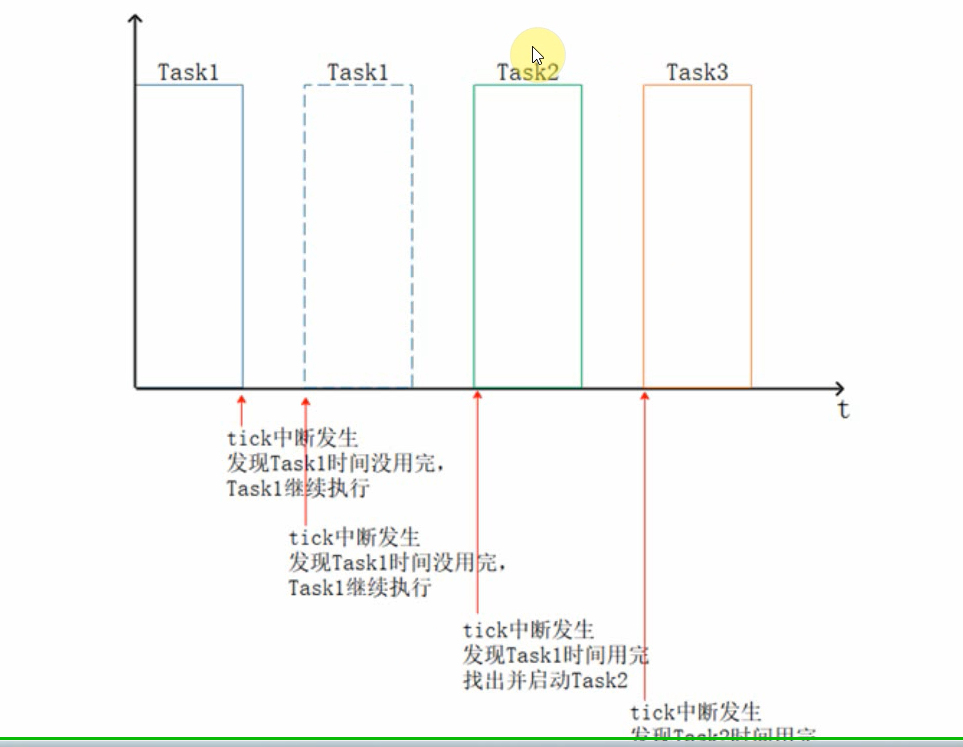
* 在移植过程中，我们不需要关注“权限”，只需要关注“地址映射”

## 3.3 中断子系统

操作系统跟单片机程序最大的区别，就是多任务，也就是**同时**运行多个程序。  
 **同时**，对人类来说是这样的，多个程序好像可以同时运行，实际上它们是**轮流**运行。

### 3.3.1 操作系统"同时"运行多个任务

轮流运行：



### 3.3.2 串口接收数据的中断

使用串口接收数据时，如果使用"查询"方式，低效并且费电。 一般都使用中断方式。

## 3.4 存储设备的驱动程序

板子上一般都有EMMC、SD/TF卡、Nor Flash、Nand Flash等存储设备。 Nor Flash、Nand Flash的驱动程序相对简单，但是这些设备比较少见了。 而EMMC、SD/TF卡的驱动程序又太复杂，足够出一个专题了。  
 我们聚焦在**最小系统**的移植，先把流程走通：用内存来模拟Flash。

## 3.5 根文件系统

光有存储设备还不行，上面需要有文件：这就是根文件系统。  
 一个程序要能运行，出了你写出的程序本身，还需要其他库，比如printf就不是你写的，它在库文件里。 根文件系统里会有这些内容：

* 程序
* 库
* 配置文件
* 用户数据(可选)
* 驱动程序(可选)

# 4. 想做更多

* 为有更好的人机交互可以移植LCD、触摸屏驱动
* 为了方便开发，移植EMMC驱动、网卡驱动
* 要接各类外设，还需要I2C、SPI、GPIO、UART驱动
* 摄像头、声卡驱动

# 5. 基础知识

移植内核对技术的要求比较全面、比较细致。

## 5.1 单片机相关的知识

* 栈的作用
* 加载地址、链接地址
* 重定位
* 几个简单的硬件知识 \* 串口 \* 定时器
* 中断的概念

## 5.2 Linux操作相关的知识

* Linux常用命令
* 简单的脚本：脚本就是把命令写在一个文件里
* GCC编译命令
* Kconfig和Makefile

## 5.3 芯片相关知识

* 能阅读芯片手册(英文)
* 移植最小系统时，涉及的手册内容不多
* 能看懂硬件原理图
* 移植最小系统时，涉及的原理图内容不多

# 6. 驱动程序知识

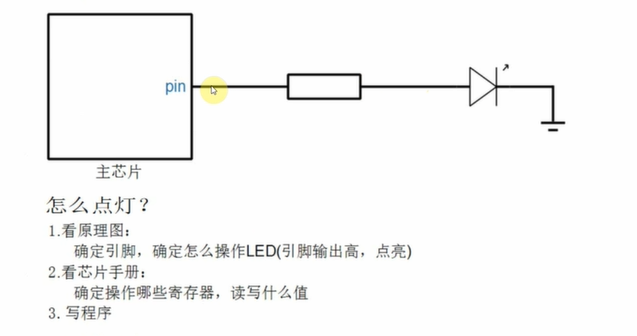
对于只有单片机知识的人来说，怎么去操作硬件？

* 直接读写寄存器
* 使用库函数

在RTOS中，本质也是去读写寄存器，但是需要有**统一**的驱动程序框架。 所以：RTOS驱动 = 驱动框架 + 硬件操作

## 6.1 以点灯为例

### 6.1.1 硬件原理

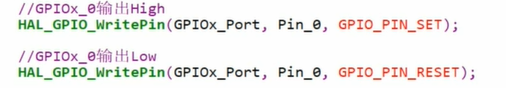


### 6.1.2 单片机点灯

* 方法1：直接读写寄存器

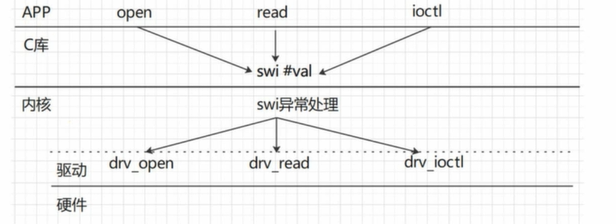
在这里插入图片描述

* 方法2：使用厂家的HAL库



### 6.1.3 Liteos-a/Linux怎么点灯

使用MMU时，一般APP与内核是相互隔离的。APP通过标准的open/read/write等文件操作函数去调用驱动程序。  
 如下图所示：



为何要多此一举？

* 它们支持MMU(内存管理单元)
* 用户程序跟内核是分隔开的，用户程序不能直接读写寄存器
* 用户程序通过标准接口访问驱动程序
* 基于这些内核的软件一般都比单片机软件复杂，术业有专攻
* 不应该让写APP的人去看原理图、写驱动、写寄存器
* 软件和硬件隔离，硬件再怎么变化，只需要改驱动，APP不需要改

## 6.2 怎么编写驱动程序

### 6.2.1 驱动程序的核心

Linux和Liteos-a的驱动程序时类似的，Liteos-a的更加精简。  
 既然APP使用驱动是调用open/read/write等接口，那么写驱动程序是最简单的方法就是提供对应的drv\_open/drv\_read/drv\_write等函数。  
 这些函数放在一个结构体里：Linux对应file\_operations结构体，Liteos-a对应file\_operations\_vfs结构体。

#### *1. Linux*

Linux中是定义一个file\_operations结构体，如下:

struct file\_operations {  
        struct module \*owner;  
        loff\_t (\*llseek) (struct file \*, loff\_t, int);  
        ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);  
        ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);  
        ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);  
        ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);  
        int (\*readdir) (struct file \*, void \*, filldir\_t);  
        unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  
        long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);  
        long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);  
        int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  
        int (\*open) (struct inode \*, struct file \*);  
        int (\*flush) (struct file \*, fl\_owner\_t id);  
        int (\*release) (struct inode \*, struct file \*);  
        int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);  
        int (\*aio\_fsync) (struct kiocb \*, int datasync);  
        int (\*fasync) (int, struct file \*, int);  
        int (\*lock) (struct file \*, int, struct file\_lock \*);  
        ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  
        unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  
        int (\*check\_flags)(int);  
        int (\*flock) (struct file \*, int, struct file\_lock \*);  
        ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);  
        ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);  
        int (\*setlease)(struct file \*, long, struct file\_lock \*\*);  
        long (\*fallocate)(struct file \*file, int mode, loff\_t offset,  
                         loff\_t len);  
        int (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);  
};

#### *2. Liteos-a*

Liteos-a中定义了一个file\_operations\_vfs结构体，如下：

struct file\_operations\_vfs  
{  
 /\* The device driver open method differs from the mountpoint open method \*/  
  
 int (\*open)(FAR struct file \*filep);  
  
 /\* The following methods must be identical in signature and position because  
 \* the struct file\_operations and struct mountp\_operations are treated like  
 \* unions.  
 \*/  
  
 int (\*close)(FAR struct file \*filep);  
 ssize\_t (\*read)(FAR struct file \*filep, FAR char \*buffer, size\_t buflen);  
 ssize\_t (\*write)(FAR struct file \*filep, FAR const char \*buffer, size\_t buflen);  
 off\_t (\*seek)(FAR struct file \*filep, off\_t offset, int whence);  
 int (\*ioctl)(FAR struct file \*filep, int cmd, unsigned long arg);  
 int (\*mmap)(FAR struct file\* filep, struct VmMapRegion \*region);  
 /\* The two structures need not be common after this point \*/  
  
#ifndef CONFIG\_DISABLE\_POLL  
 int (\*poll)(FAR struct file \*filep, poll\_table \*fds);  
#endif  
 int (\*unlink)(FAR struct inode \*inode);  
};

## 6.3 注册驱动程序

### 1. Linux

static struct file\_operations hello\_drv = {  
        .owner         = THIS\_MODULE,  
        .open = hello\_drv\_open,  
        .read = hello\_drv\_read,  
        .write = hello\_drv\_write,  
        .release = hello\_drv\_close,  
};  
  
int major = register\_chrdev(0, "hello", &hello\_drv); /\* /dev/hello \*/  
static struct class \*hello\_class = class\_create(THIS\_MODULE, "hello\_class");  
device\_create(hello\_class, NULL, MKDEV(major, 0), NULL, "hello"); /\* /dev/hello \*/

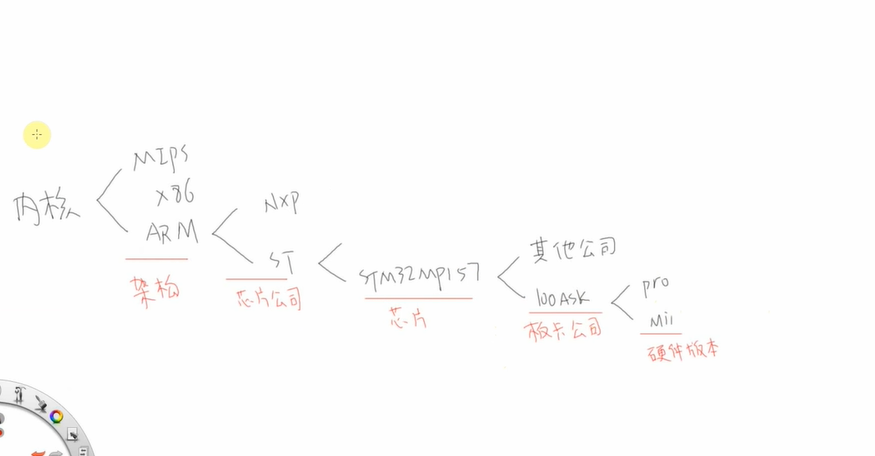
### 2. Liteos-a

static const struct file\_operations\_vfs g\_helloDevOps = {  
 .open = hello\_open,  
 .close = hello\_close,  
 .read = hello\_read,  
 .write = NULL,  
 .seek = NULL,  
 .ioctl = NULL,  
 .mmap = NULL,  
 .unlink = NULL,  
};  
  
int ret = register\_driver("/dev/hello", &g\_helloDevOps, 0666, NULL);

## 6.4 APP如何使用

Linux和Liteos-a在APP层面都一样：

int main(int argc, char \*\*argv)  
{  
        int fd;  
        char buf[1024];  
        int len;  
          
        /\* 1. 判断参数 \*/  
        if (argc < 2)   
        {  
                printf("Usage: %s -w <string>\n", argv[0]);  
                printf(" %s -r\n", argv[0]);  
                return -1;  
        }  
  
        /\* 2. 打开文件 \*/  
        fd = open("/dev/hello", O\_RDWR);  
        if (fd == -1)  
        {  
                printf("can not open file /dev/hello\n");  
                return -1;  
        }  
  
        /\* 3. 写文件或读文件 \*/  
        if ((0 == strcmp(argv[1], "-w")) && (argc == 3))  
        {  
                len = strlen(argv[2]) + 1;  
                len = len < 1024 ? len : 1024;  
                write(fd, argv[2], len);  
        }  
        else  
        {  
                len = read(fd, buf, 1024);                  
                buf[1023] = '\0';  
                printf("APP read : %s\n", buf);  
        }  
          
        close(fd);  
          
        return 0;  
}



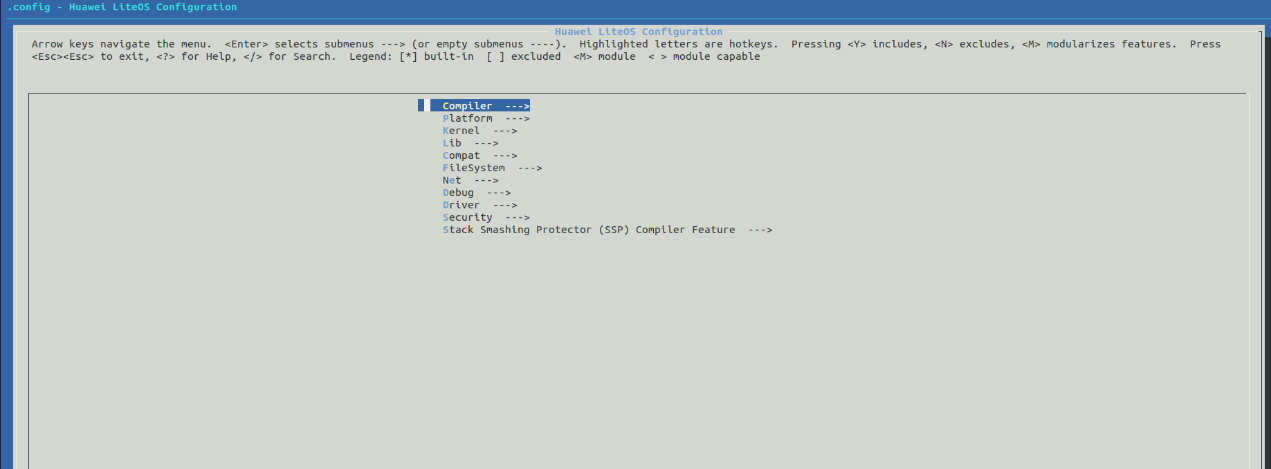
参考文档：

任一个Linux内核的Documentation\kbuild\kconfig-language.rst  
  
<https://www.rt-thread.org/document/site/programming-manual/kconfig/kconfig/>

* 1
* 2
* 3

对于各类内核，只要支持menuconfig配置界面，都是使用Kconfig。  
 在配置界面中，可以选择、设置选项，这些设置会保存在.config文件里。  
 Makefile会包含.config，根据里面的值决定编译哪些文件、怎么编译文件。

## 1.1 配置界面示例



问题：

* 这个界面里，各个配置项来自哪里
* 这个界面里，这些配置项是怎么组织的
* 这个界面里，我们的选择、设置，结果保存在哪里

## 1.2 配置结果的保存

### 1.2.1 示例

在配置界面中操作的结果保存在.config文件中，示例如下：

# LOSCFG\_COMPILER\_HIMIX\_32 is not set  
LOSCFG\_COMPILER\_CLANG\_LLVM=y  
  
#  
# Platform  
#  
LOSCFG\_PLATFORM="stm32mp157"  
# LOSCFG\_PLATFORM\_HI3516DV300 is not set  
# LOSCFG\_PLATFORM\_HI3518EV300 is not set  
LOSCFG\_PLATFORM\_STM32MP157=y  
# LOSCFG\_PLATFORM\_IMX6ULL is not set  
LOSCFG\_PLATFORM\_BSP\_GIC\_V2=y  
LOSCFG\_ARCH\_ARM=y  
LOSCFG\_ARCH\_ARM\_AARCH32=y  
LOSCFG\_ARCH\_ARM\_V7A=y  
LOSCFG\_ARCH\_ARM\_VER="armv7-a"  
LOSCFG\_ARCH\_FPU\_VFP\_V4=y  
LOSCFG\_ARCH\_FPU\_VFP\_D32=y  
LOSCFG\_ARCH\_FPU\_VFP\_NEON=y  
LOSCFG\_ARCH\_FPU="neon-vfpv4"  
LOSCFG\_ARCH\_CORTEX\_A7=y  
LOSCFG\_ARCH\_CPU="cortex-a7"



* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

Makefile会包含.config文件，它会根据里面的变量比如LOSCFG\_PLATFORM\_STM32MP157选择单板相关的文件。

### 1.2.2 配置项的前缀

在Kconfig文件中，假设配置项的名字是XXX，在.config文件中：

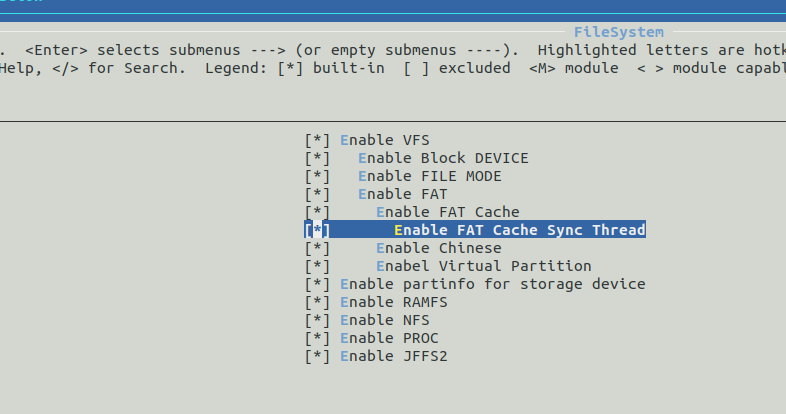
* 默认情况下，它对应的变量名为CONFIG\_XXX
* 如果设置了环境变量CONFIG\_=ABC，则对应的变量名为ABC\_XXX
* 在Liteos-a中的Makefile中export CONFIG\_=LOSCFG\_，所以对应的变量名为LOSCFG\_XXX



## 1.3 描述单个配置项config

### 1.3.1 示例

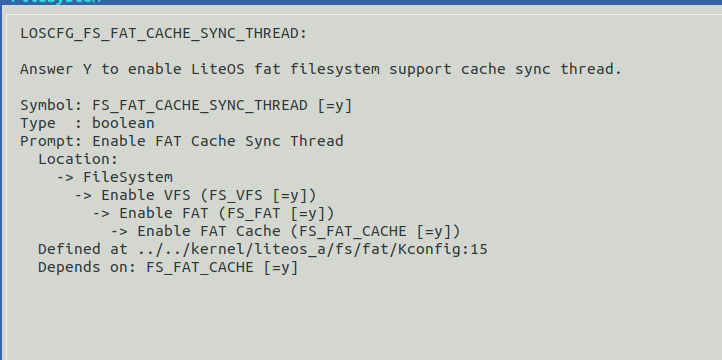
在make menuconfig界面，可以看到这个配置项：



在配置界面，使用方向箭头游走到Enable FAT Cache Sync Thread后，可以：

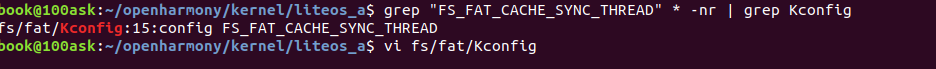
* 输入Y，选择配置项，在.config中对应LOSCFG\_FS\_FAT\_CACHE\_SYNC\_THREAD=y
* 输入N，不选择配置项，在.config中对应# LOSCFG\_FS\_FAT\_CACHE\_SYNC\_THREAD is not set

上图中的配置项怎么实现的？ (下面是帮助信息。我们可以通过这个找到kconfig文件)

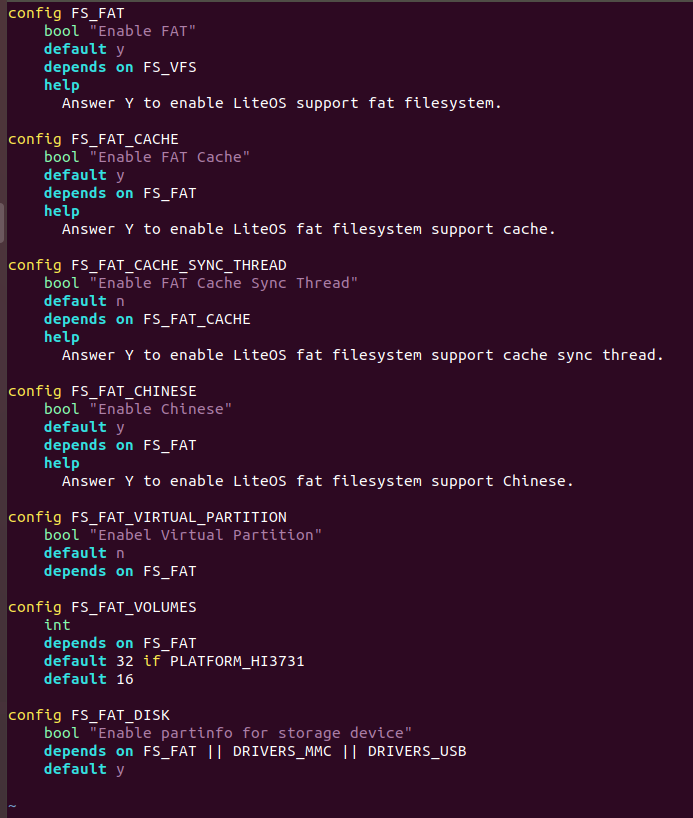


grep "FS\_FAT\_CACHE\_SYNC\_THREAD" \* -nr | grep Kconfig

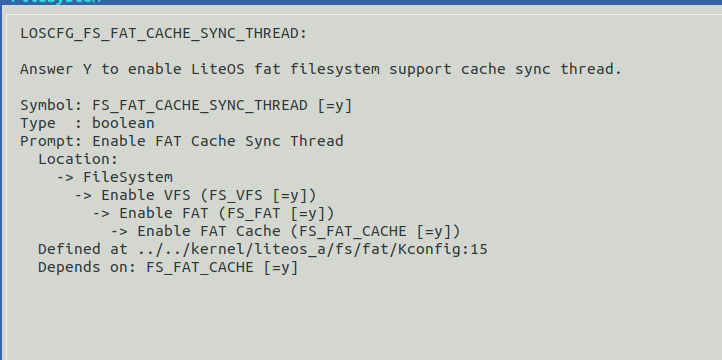
* 1



在Kconfig文件中，它对应下列代码：



### 1.3.2 语法



解释如下：

* config 表示config option，这是Kconfig的基本entry；其他entry是用来管理config的。 config 表示一个配置选项的开始，紧跟着的 FS\_FAT\_CACHE\_SYNC\_THREAD 是配置选项的名称。

config 下面几行定义了该配置选项的属性。

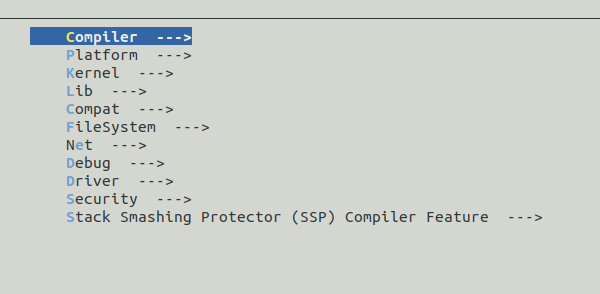
属性可以是该配置选项的：类型、输入提示、依赖关系、默认值、帮助信息。

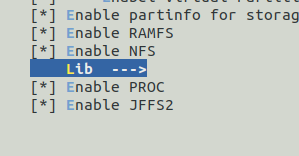
* bool 表示配置选项的类型，每个 config 菜单项都要有类型定义，变量有5种类型

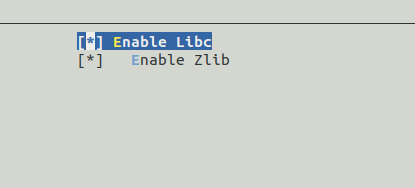
* bool 布尔类型
* tristate 三态类型
* string 字符串
* hex 十六进制
* int 整型
* "Enable FAT Cache Sync Thread"：提示信息
* depends on：表示依赖关系，只有FS\_FAT\_CACHE被选中，才可以选择FS\_FAT\_CACHE\_SYNC\_THREAD
* select XXX：表示反向依赖关系，即当前配置选项被选中后，XXX选项就会被选中。
* default 表示配置选项的默认值，bool 类型的默认值可以是 y/n。
* help 帮助信息，在make menuconfig界面输入H键时，就会提示帮助信息。

## 1.4 实现菜单menu/endmenu

### 1.4.1 示例







在Kconfig中，代码如下：

menu "Lib"  
config LIB\_LIBC  
 bool "Enable Libc"  
 default y  
 help  
 Answer Y to enable libc for full code.  
  
config LIB\_ZLIB  
 bool "Enable Zlib"  
 default y  
 depends on LIB\_LIBC  
 help  
 Answer Y to enable LiteOS support compress file library.  
endmenu

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

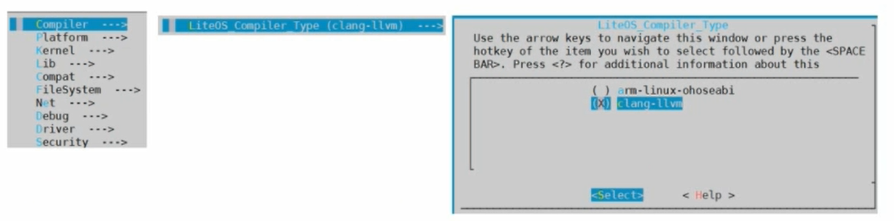
### 1.4.2 语法

解释如下：

* menu "xxx"表示一个菜单，菜单名是"xxx"
* menu和endmenu之间的entry都是"xxx"菜单的选项
* 在上面的例子中子菜单有2个选项：“Enable Libc”、“Enable Zlib”
* 由于第二个菜单项依赖于第一个菜单项，所以第二个菜单项缩进一格

## 1.5 实现单选choice/endchoice

### 1.5.1 示例



在上述界面中，对于LiteOS\_Compiler\_Type，有2个选择：arm-linux-ohoseabi、clang-llvm。  
 在Kconfig文件中怎么描述？如下：

menu "Compiler"  
choice  
 prompt "LiteOS\_Compiler\_Type"  
 default COMPILER\_CLANG\_LLVM  
 help  
 Enable arm-himix100 or aarch64-himix100 or compiler.  
  
config COMPILER\_HIMIX\_32  
 bool "arm-linux-ohoseabi"  
 depends on PLATFORM\_HI3518EV300 || PLATFORM\_HI3516DV300 || PLATFORM\_IMX6ULL || PLATFORM\_STM32MP157  
  
config COMPILER\_CLANG\_LLVM  
 bool "clang-llvm"  
 depends on PLATFORM\_HI3518EV300 || PLATFORM\_HI3516DV300 || PLATFORM\_IMX6ULL || PLATFORM\_STM32MP157  
  
endchoice  
endmenu



* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

### 1.5.2 语法

解释如下：

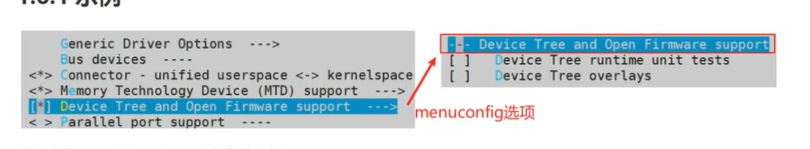
* choice表示"选择"
* choice和endchoice之间的entry是可以选择的项目
* 它们之间，只能有一个被设置为"y"：表示编进内核
* 它们之间，可以设置多个为"m"：表示编译为模块
* 比如一个硬件有多个驱动程序

* 同一时间只能有一个驱动能编进内核
* 但是多个驱动都可以单独编译为模块

## 1.6 menuconfig

menuconfig XXX和config XXX类似， 唯一不同的是该选项除了能设置y/m/n外，还可以实现菜单效果(**能回车进入该项内部**)。

### 1.6.1 示例



对于上述界面，Kconfig文件中代码如下：

menuconfig OF  
 bool "Device Tree and Open Firmware support"  
 help  
 This option enables the device tree infrastructure.  
 It is automatically selected by platforms that need it or can  
 be enabled manually for unittests, overlays or  
 compile-coverage.  
  
if OF  
  
config OF\_UNITTEST  
 bool "Device Tree runtime unit tests"  
 depends on OF\_IRQ  
 select OF\_EARLY\_FLATTREE  
 select OF\_RESOLVE  
 help  
 This option builds in test cases for the device tree infrastructure  
 that are executed once at boot time, and the results dumped to the  
 console.  
  
 If unsure, say N here, but this option is safe to enable.  
  
config OF\_OVERLAY  
 bool "Device Tree overlays"  
 select OF\_DYNAMIC  
 select OF\_RESOLVE  
 help  
 Overlays are a method to dynamically modify part of the kernel's  
 device tree with dynamically loaded data.  
 While this option is selected automatically when needed, you can  
 enable it manually to improve device tree unit test coverage.  
  
endif # OF



* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33

### 1.6.2 语法

menuconfig常用格式有2种：

menuconfig M  
 if M  
 config C1  
 config C2  
 endif

* 1
* 2
* 3
* 4
* 5

或：

menuconfig M  
 config C1  
 depends on M  
 config C2  
 depends on M

* 1
* 2
* 3
* 4
* 5

第1项menuconfig M跟config M语法是一样的， 不同之处在于menuocnfig M后面可以跟着好几个依赖于M的config C1、config C2等子配置项。

## 1.7 if/endif

### 1.7.1 语法

在上面的menuconfig中就有if/endif的使用，它的语法如下：

"if" <expr>  
<if block>  
"endif"

* 1
* 2
* 3

### 1.7.2 示例

示例如下，只有定义的OF项，OF\_UNITTEST和OF\_OVERLAY才会显示出来：

if OF  
  
config OF\_UNITTEST  
 bool "Device Tree runtime unit tests"  
 depends on OF\_IRQ  
 select OF\_EARLY\_FLATTREE  
 select OF\_RESOLVE  
 help  
 This option builds in test cases for the device tree infrastructure  
 that are executed once at boot time, and the results dumped to the  
 console.  
  
 If unsure, say N here, but this option is safe to enable.  
  
config OF\_OVERLAY  
 bool "Device Tree overlays"  
 select OF\_DYNAMIC  
 select OF\_RESOLVE  
 help  
 Overlays are a method to dynamically modify part of the kernel's  
 device tree with dynamically loaded data.  
 While this option is selected automatically when needed, you can  
 enable it manually to improve device tree unit test coverage.  
  
endif # OF



* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

## 1.8 source

source 语句用于读取另一个文件中的 Kconfig 文件，如：

source "../../kernel/liteos\_a/platform/Kconfig"

* 1

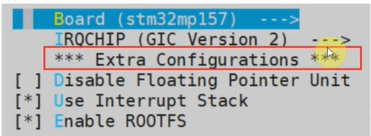
## 1.9 comment

comment 语句出现在界面的第一行，用于定义一些提示信息，如：

config ARCH\_ARM  
 bool  
  
source "arch/arm/Kconfig"  
  
comment "Extra Configurations"  
  
config ARCH\_FPU\_DISABLE  
 bool "Disable Floating Pointer Unit"  
 default n  
 help  
 This option will bypass floating procedure in system.

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

界面如下：



# 2.准备工作

安装软件在GIT仓库：

doc\_and\_source\_for\_openharmony\工具\  
 codeblocks-20.03mingw-setup.exe  
 make-3.81.exe

* 1
* 2
* 3

* **arm-linux-gcc**
* 给ARM芯片编译程序
* **gcc**
* 在x86编译程序
* 用法基本一样
* 为方便演示，我们使用gcc
* 为了方便在windows下演示，我们使用Code::Blocks
* 它的安装程序自带gcc

## 2.2. Code::Blocks

它是一款基于GCC的windows IDE，可以用来开发C/C++/Fortran。  
 官网地址：http://www.codeblocks.org/

在我们提供的GIT仓库里也有：git clone <https://e.coding.net/weidongshan/openharmony/doc_and_source_for_openharmony.git>  
 下载GIT后，在工具目录下。

### 2.2.1 安装

按提示安装。

### 2.2.2 设置windows环境变量

在Path环境变量中添加：C:\Program Files\CodeBlocks\MinGW\bin

### 2.2.3 命令行示例

启动Git Bash，编译程序hello.c:

#include <stdio.h>  
  
int main(void)  
{  
        printf("hello, world!\n");  
        return 0;  
}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

编译、运行命令如下：

gcc -o hello hello.c  
./hello.exe

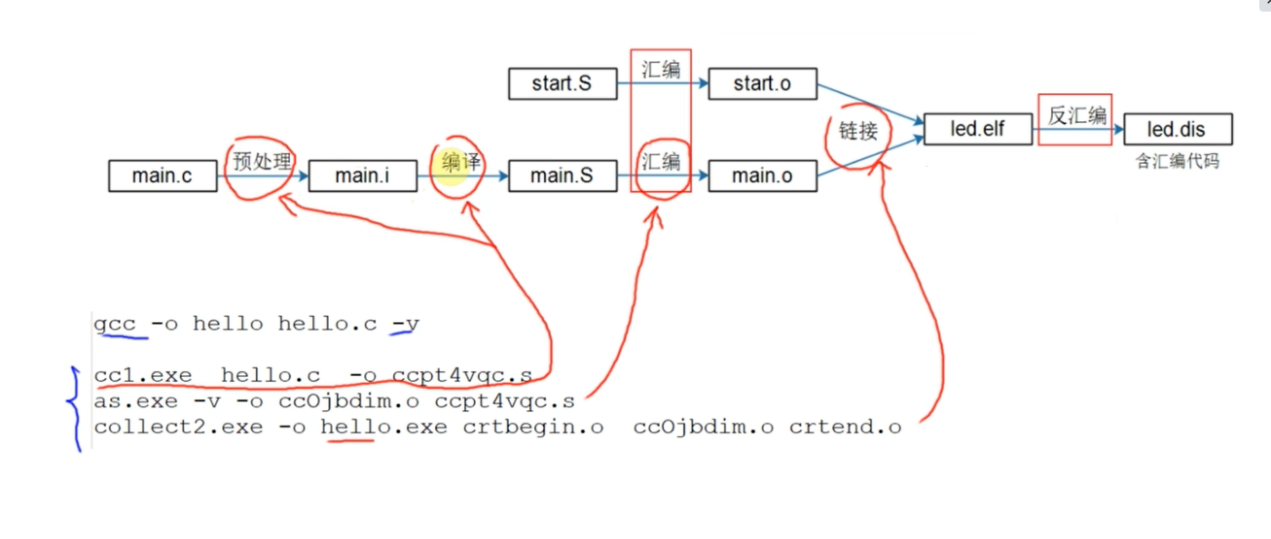
* 1
* 2

## 2.3. Make

安装make-3.81.exe后，在Path环境变量中添加：C:\Program Files (x86)\GnuWin32\bin

配套源码在GIT仓库里各个板子的source\02\_录制视频时编写的源码\01\_gcc\_options里。

## 3.1. 程序编译4步骤



我们经常使用“编译”泛指上面的4个步骤之一，甚至有时候会囊括这四个步骤。

## 3.2. gcc的使用方法

gcc [选项] 文件名

* 1

### 3.2.1 gcc使用示例

gcc hello.c // 输出一个名为a.out的可执行程序，然后可以执行./a.out  
gcc -o hello hello.c // 输出名为hello的可执行程序，然后可以执行./hello  
gcc -o hello hello.c -static // 静态链接  
  
gcc -c -o hello.o hello.c // 先编译(不链接)  
gcc -o hello hello.o // 再链接

* 1
* 2
* 3
* 4
* 5
* 6

### 3.2.2 gcc常用选项

#### *3.2.2.1 手工控制编译过程*

选项功能

|  |  |
| --- | --- |
|  |  |
| -v | 查看gcc编译器的版本，显示gcc执行时的详细过程 |
| -o | 指定输出文件名为file，这个名称不能跟源文件名同名 |
| -E | 只预处理，不会编译、汇编、链接t |
| -S | 只编译，不会汇编、链接 |
| -c | 编译和汇编，不会链接 |

一个c/c++文件要经过预处理、编译、汇编和链接才能变成可执行文件。  
 （1）预处理  
 C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个“.i”文件中等待进一步处理。  
 （2）编译  
 编译就是把C/C++代码(比如上述的“.i”文件)“翻译”成汇编代码。  
 （3）汇编  
 汇编就是将第二步输出的汇编代码翻译成符合一定格式的机器代码，在Linux系统上一般表现为ELF目标文件(OBJ文件)。“反汇编”是指将机器代码转换为汇编代码，这在调试程序时常常用到。  
 （4）链接  
 链接就是将上步生成的OBJ文件和系统库的OBJ文件、库文件链接起来，最终生成了可以在特定平台运行的可执行文件。

hello.c(预处理)->hello.i(编译)->hello.s(汇编)->hello.o(链接)->hello 详细的每一步命令如下：

gcc -E -o hello.i hello.c  
gcc -S -o hello.s hello.i  
gcc -c -o hello.o hello.s  
gcc -o hello hello.o

* 1
* 2
* 3
* 4

上面一连串命令比较麻烦，gcc会对.c文件默认进行预处理操作，使用-c再来指明了编译、汇编，从而得到.o文件,再将.o文件进行链接，得到可执行应用程序。简化如下：

gcc -c -o hello.o hello.c//前三步骤  
gcc -o hello hello.o

* 1
* 2

#### *3.2.2.2 使用后缀名决定编译过程*

参考《嵌入式Linux应用开发完全手册》：

* 总结 \* 输入文件的后缀名和选项共同决定gcc到底执行那些操作
* 在编译过程中，最后的步骤都是链接
* 除非使用了-E、-S、-c选项
* 或者编译出错阻止了完整的编译过程

#### *3.2.2.3 指定头文件目录*

头文件在哪里？

* 系统目录 \* 系统目录在哪？工具链里的某个include目录

* 怎么确定？

```  
echo 'main(){}'| gcc -E -v - // 它会列出头文件目录、库目录(LIBRARY\_PATH)  
```

* 1
* 2
* 3

* 可以不使用系统include目录吗？可以，编译时指定参数-nostdinc
* 可以自己指定头文件目录

-I <头文件目录>

* 1

#### *3.2.2.4 指定库文件*

库文件在哪里？

* 系统目录 \* 系统目录在哪？工具链里的某个lib目录

* 怎么确定？

```  
echo 'main(){}'| gcc -E -v - // 它会列出头文件目录、库目录(LIBRARY\_PATH)  
```

* 1
* 2
* 3

* 可以不使用系统lib目录吗？可以，编译时指定参数-nostdlib
* 可以自己指定库文件目录

-L <库文件目录>

* 1

* 指定库文件

-l <abc> // 链接 libabc.so 或 lib.a

* 1

## 3.3 开发板程序编译示例

最后链接时，使用arm-linux-ld而不是使用arm-linux-gcc

* 前者可以完全自己指定所连接的文件
* 后者会链接一些默认的启动文件

1.1 怎么编译子目录

以kernel/liteos\_a/fs/fat/Makefile为例：

# Copyright (c) 2013-2019, Huawei Technologies Co., Ltd. All rights reserved.

# Copyright (c) 2020, Huawei Device Co., Ltd. All rights reserved.

#

# Redistribution and use in source and binary forms, with or without modification,

# are permitted provided that the following conditions are met:

#

# 1. Redistributions of source code must retain the above copyright notice, this list of

# conditions and the following disclaimer.

#

# 2. Redistributions in binary form must reproduce the above copyright notice, this list

# of conditions and he following disclaimer in the documentation and/or other materials

# provided with the distribution.

#

# 3. Neither the name of the copyright holder nor the names of its contributors may be used

# to endorse or promote products derived from this software without specific prior written

# permission.

#

# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS

# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,

# THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

# PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR

# CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

# EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;

# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,

# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR

# OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF

# ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

include $(LITEOSTOPDIR)/config.mk

MODULE\_NAME := $(notdir $(shell pwd))//当前目录fat

LOCAL\_SRCS := $(wildcard os\_adapt/\*.c)//

LOCAL\_SRCS += $(wildcard $(LITEOSTHIRDPARTY)/FatFs/source/\*.c)

LOCAL\_INCLUDE := \

-I $(LITEOSTHIRDPARTY)/FatFs/source \

-I $(LITEOSTOPDIR)/fs/fat/virpart/include

LOCAL\_FLAGS := $(LOCAL\_INCLUDE) $(LITEOS\_GCOV\_OPTS)

include $(MODULE)

~

~

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

第1行包含config.mk

这是包含一些预先定义的变量，比如默认的编译选项等。

下面的代码定义了LOCAL\_SRCS

等于一系列C文件，这就是要编译的源文件。

定义了LOCAL\_INCLUDE

这是头文件的目录

定义了LOCAL\_FLAGS

这是编译选项

定义了MODULE\_NAME

一般等于当前目录的名字，比如fat，以后就编译得到libfat.a

怎么编译？看最后一行

include $(MODULE)

1

MODULE就是：

MODULE = $(MK\_PATH)/module.mk # kernel/liteos\_a/tools/build/mk/module.mk

1

分析module.mk：

# 找到第1个目标

all : $(LIB)

# LIB是什么, 如果没定义LOCAL\_SO，LIB就是 lib$(MODULE\_NAME).a, 比如 libfat.a

ifeq ($(LOCAL\_SO), y)

LIBSO := $(OUT)/lib/lib$(MODULE\_NAME).so

LIBA := $(OUT)/lib/lib$(MODULE\_NAME).a

else

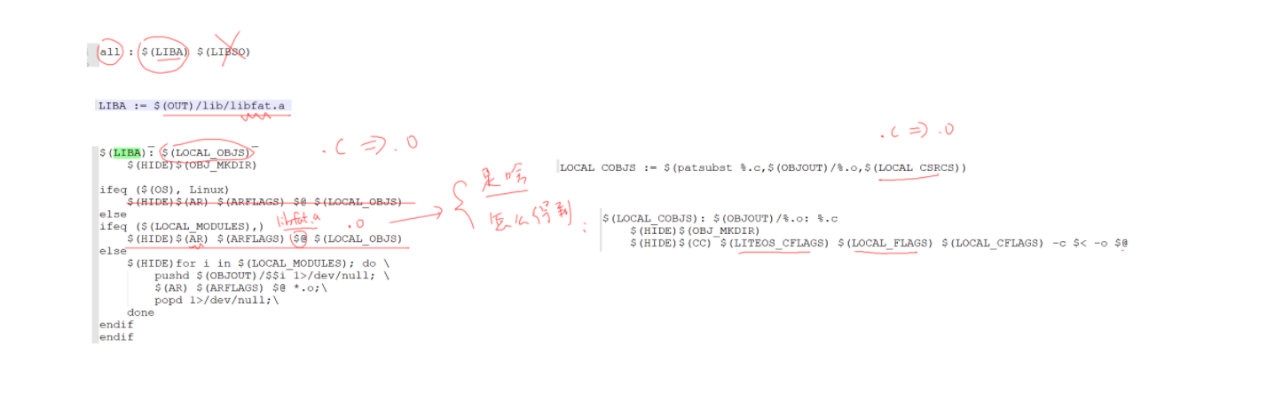
LIBSO :=

LIBA := $(OUT)/lib/lib$(MODULE\_NAME).a

endif

LIB := $(LIBA) $(LIBSO)

# 怎么编译 LIBA ? 看下图

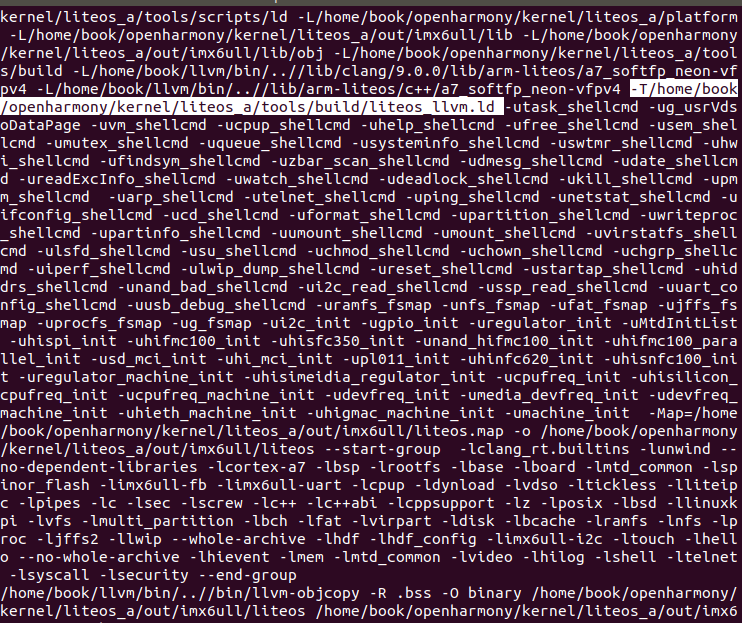


1.2 编译哪些子目录

1.2.1 从链接命令看内核的组成

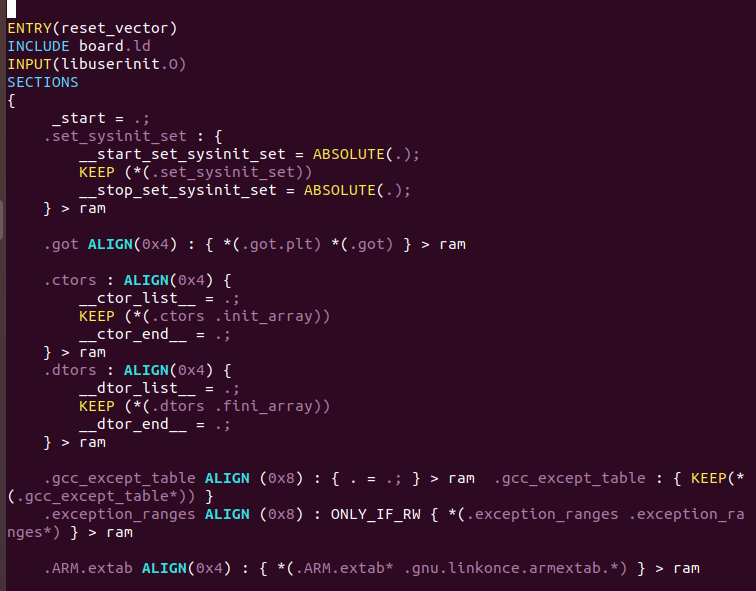
链接命令如下

liteos-a由一系列的库文件组成，reset\_vector是它的入口。



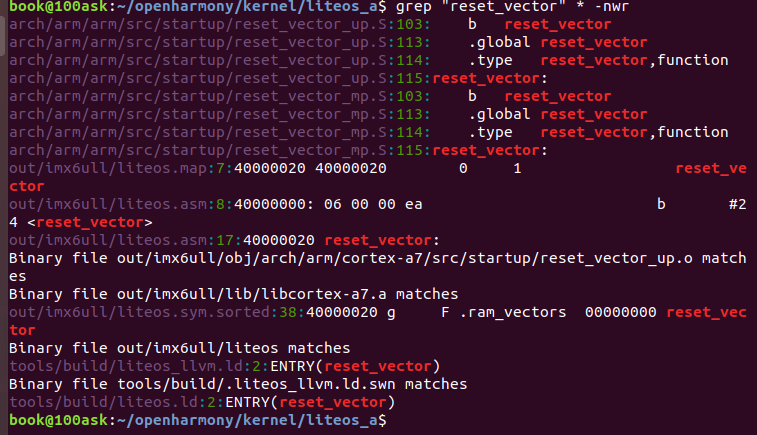
查看链接脚本

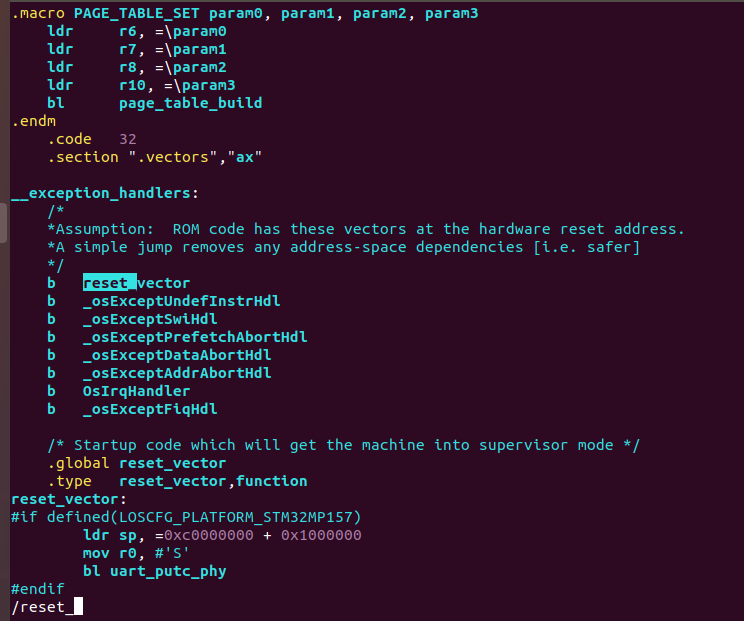
vi /home/book/openharmony/kernel/liteos\_a/tools/build/liteos\_llvm.ld



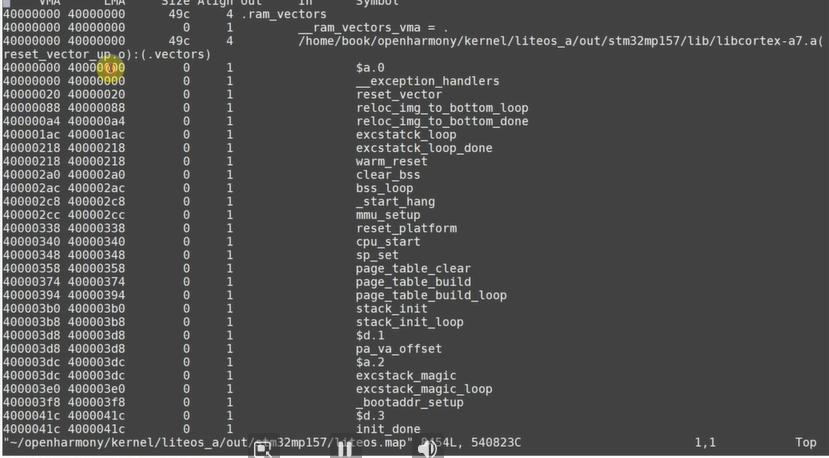
查看入口：

grep "reset\_vector" \* -nwr





查看liteos.map文件



1.2.2 从Makefile开始分析

从链接命令查看内核的组成，是一个取巧的方法。

本质的方法应该是从kernel/liteos\_a/Makefile开始分析。

1.3 顶层Makefile分析

指的是kernel/liteos\_a/Makefile。

1.3.1 Makfile中常用变量

1. LITEOSTOPDIR // kernel/liteos\_a

2. LITEOSTHIRDPARTY // third\_party

3. LITEOS\_MK\_PATH // kernel/liteos\_a/tools/build/mk

4. MK\_PATH = $(LITEOSTOPDIR)/tools/build/mk // kernel/liteos\_a/tools/build/mk

1.3.2 包含的文件

Makefile里使用include命令包含了很多文件，这些文件又包含了其他文件。

1. STM32MP157

Makefile

-include $(LITEOSTOPDIR)/tools/build/config.mk

-include $(LITEOSTOPDIR)/tools/build/mk/los\_config.mk

-include $(LITEOSTOPDIR)/.config

include $(LITEOSTOPDIR)/arch/cpu.mk

-include $(LITEOSTOPDIR)/arch/arm/arm.mk

include $(LITEOSTOPDIR)/platform/bsp.mk

include $(LITEOSTOPDIR)/../../vendor/st/stm32mp157/stm32mp157.mk

include $(LITEOSTOPDIR)/../../drivers/hdf/lite/hdf\_lite.mk

include $(LITEOSTOPDIR)/../../vendor/st/hdf/hdf\_vendor.mk

-include $(LITEOSTOPDIR)/3rdParty/3rdParty.mk

-include $(LITEOS\_MK\_PATH)/liteos\_tables\_ldflags.mk

-include $(LITEOS\_MK\_PATH)/dynload.mk

1

2

3

4

5

6

7

8

9

10

11

12

13

2. IMX6ULL

Makefile

-include $(LITEOSTOPDIR)/tools/build/config.mk

-include $(LITEOSTOPDIR)/tools/build/mk/los\_config.mk

-include $(LITEOSTOPDIR)/.config

include $(LITEOSTOPDIR)/arch/cpu.mk

-include $(LITEOSTOPDIR)/arch/arm/arm.mk

include $(LITEOSTOPDIR)/platform/bsp.mk

include $(LITEOSTOPDIR)/../../vendor/nxp/imx6ull/imx6ull.mk

include $(LITEOSTOPDIR)/../../drivers/hdf/lite/hdf\_lite.mk

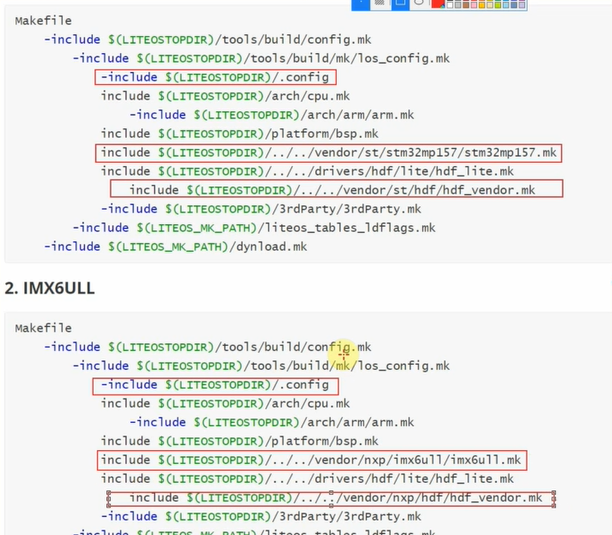
include $(LITEOSTOPDIR)/../../vendor/nxp/hdf/hdf\_vendor.mk

-include $(LITEOSTOPDIR)/3rdParty/3rdParty.mk

-include $(LITEOS\_MK\_PATH)/liteos\_tables\_ldflags.mk

-include $(LITEOS\_MK\_PATH)/dynload.mk

两个板子的区别：



3. 展开Makefile

把Makefile里被包含的文件都展开，这样便于分析。

我们事先对STM32MP157、IMX6ULL展开了它们的Makefile，结果分别保存在这两个文件里：

Makefile\_all\_stm32mp157.txt

Makefile\_all\_imx6ull.txt

注意：-include表示有文件就包含，没有文件就不包含。

1.3.3 分析make过程

1. 第1个目标

all: $(OUT) $(BUILD) $(LITEOS\_TARGET) $(APPS)

1

2. 目标: OUT

OUT目标：

创建了目录：kernel/liteos\_a/imx6ull/lib

把board.ld.S编译成了board.ld，这是链接文件。

# .config文件中， LOSCFG\_PLATFORM="imx6ull"

OUT = $(LITEOSTOPDIR)/out/$(LITEOS\_PLATFORM)

$(OUT): $(LITEOS\_MENUCONFIG\_H)

$(HIDE)mkdir -p $(OUT)/lib

$(HIDE)$(CC) -I$(LITEOS\_PLATFORM\_BASE)/include -I$(BOARD\_INCLUDE\_DIR) \

-E $(LITEOS\_PLATFORM\_BASE)/board.ld.S \

-o $(LITEOS\_PLATFORM\_BASE)/board.ld -P

1

2

3

4

5

6

7

8

OUT的依赖：LITEOS\_MENUCONFIG\_H

配置内核，生成头文件autoconf.h。

LITEOS\_MENUCONFIG\_H = $(LITEOSTOPDIR)/include/generated/autoconf.h

KCONFIG\_FILE\_PATH = $(LITEOSTOPDIR)/Kconfig

$(LITEOS\_MENUCONFIG\_H):

ifneq ($(LITEOS\_PLATFORM\_MENUCONFIG\_H), $(wildcard $(LITEOS\_PLATFORM\_MENUCONFIG\_H)))

$(HIDE)$(MAKE) genconfig

endif

genconfig:$(MENUCONFIG\_PATH)/conf

$(HIDE)mkdir -p include/config include/generated

$< --silentoldconfig $(KCONFIG\_FILE\_PATH)

-mv -f $(LITEOS\_MENUCONFIG\_H) $(LITEOS\_PLATFORM\_MENUCONFIG\_H)

1

2

3

4

5

6

7

8

9

10

11

12

13

3. 目标: BUILD

创建目录 kernel/liteos\_a/out/stm32mp157/obj或kernel/liteos\_a/out/imx6ull/obj

OUT = $(LITEOSTOPDIR)/out/$(LITEOS\_PLATFORM)

BUILD = $(OUT)/obj

$(BUILD):

$(HIDE)mkdir -p $(BUILD)

1

2

3

4

4. 目标: LITEOS\_TARGET

这是核心，进入子目录执行make，把子目录中的文件链接为一个库。

最后，把这些库链接为liteos内核。

LITEOS\_TARGET = liteos

$(LITEOS\_TARGET): $(\_\_LIBS)

$(HIDE)touch $(LOSCFG\_ENTRY\_SRC)

$(HIDE)for dir in $(LITEOS\_SUBDIRS); \

do $(MAKE) -C $$dir all || exit 1; \

done

$(LD) $(LITEOS\_LDFLAGS) $(LITEOS\_TABLES\_LDFLAGS) $(LITEOS\_DYNLDFLAGS) -Map=$(OUT)/$@.map -o $(OUT)/$@ --start-group $(LITEOS\_LIBDEP) --end-group

# $(SIZE) -t --common $(OUT)/lib/\*.a >$(OUT)/$@.objsize

$(OBJCOPY) -O binary $(OUT)/$@ $(LITEOS\_TARGET\_DIR)/$@.bin

$(OBJDUMP) -t $(OUT)/$@ |sort >$(OUT)/$@.sym.sorted

$(OBJDUMP) -d $(OUT)/$@ >$(OUT)/$@.asm

# $(NM) -S --size-sort $(OUT)/$@ >$(OUT)/$@.size

1

2

3

4

5

6

7

8

9

10

11

12

13

14

目标：\_\_LIBS

# 没做什么事

\_\_LIBS = libs

$(\_\_LIBS): $(OUT) $(CXX\_INCLUDE)

1

2

3

命令：$(HIDE)touch $(LOSCFG\_ENTRY\_SRC)

每次都要编译los\_config.c，touch一下

LOSCFG\_ENTRY\_SRC = $(LITEOSTOPDIR)/kernel/common/los\_config.c

1

命令：进入每个LITEOS\_SUBDIRS，执行make

# LIB\_SUBDIRS 等于一系列的目录

LIB\_SUBDIRS :=

LIB\_SUBDIRS += arch/arm/$(LITEOS\_ARCH\_ARM) # 就是arch/arm/arm

LIB\_SUBDIRS += $(PLATFORM\_BSP\_HISI\_BASE)

LIB\_SUBDIRS += $(LITEOSTOPDIR)/kernel/common

LIB\_SUBDIRS += kernel/base

LIB\_SUBDIRS += $(IMX6ULL\_BASE\_DIR)/board

LIB\_SUBDIRS += $(IMX6ULL\_BASE\_DIR)/driver/mtd/common

LIB\_SUBDIRS += $(IMX6ULL\_BASE\_DIR)/driver/mtd/spi\_nor

LIB\_SUBDIRS += $(IMX6ULL\_BASE\_DIR)/driver/imx6ull-fb

LIB\_SUBDIRS += $(IMX6ULL\_BASE\_DIR)/driver/imx6ull-uart

LIB\_SUBDIRS += kernel/extended/cpup

LIB\_SUBDIRS += lib/libc

LIB\_SUBDIRS += lib/libsec

LIB\_SUBDIRS += lib/libscrew

LIB\_SUBDIRS += fs/fat

LIB\_SUBDIRS += fs/jffs2

LITEOS\_SUBDIRS = $(LIB\_SUBDIRS)

$(HIDE)for dir in $(LITEOS\_SUBDIRS); \

do $(MAKE) -C $$dir all || exit 1; \

done

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

链接及各类处理

$(LD) $(LITEOS\_LDFLAGS) $(LITEOS\_TABLES\_LDFLAGS) $(LITEOS\_DYNLDFLAGS) -Map=$(OUT)/$@.map -o $(OUT)/$@ --start-group $(LITEOS\_LIBDEP) --end-group

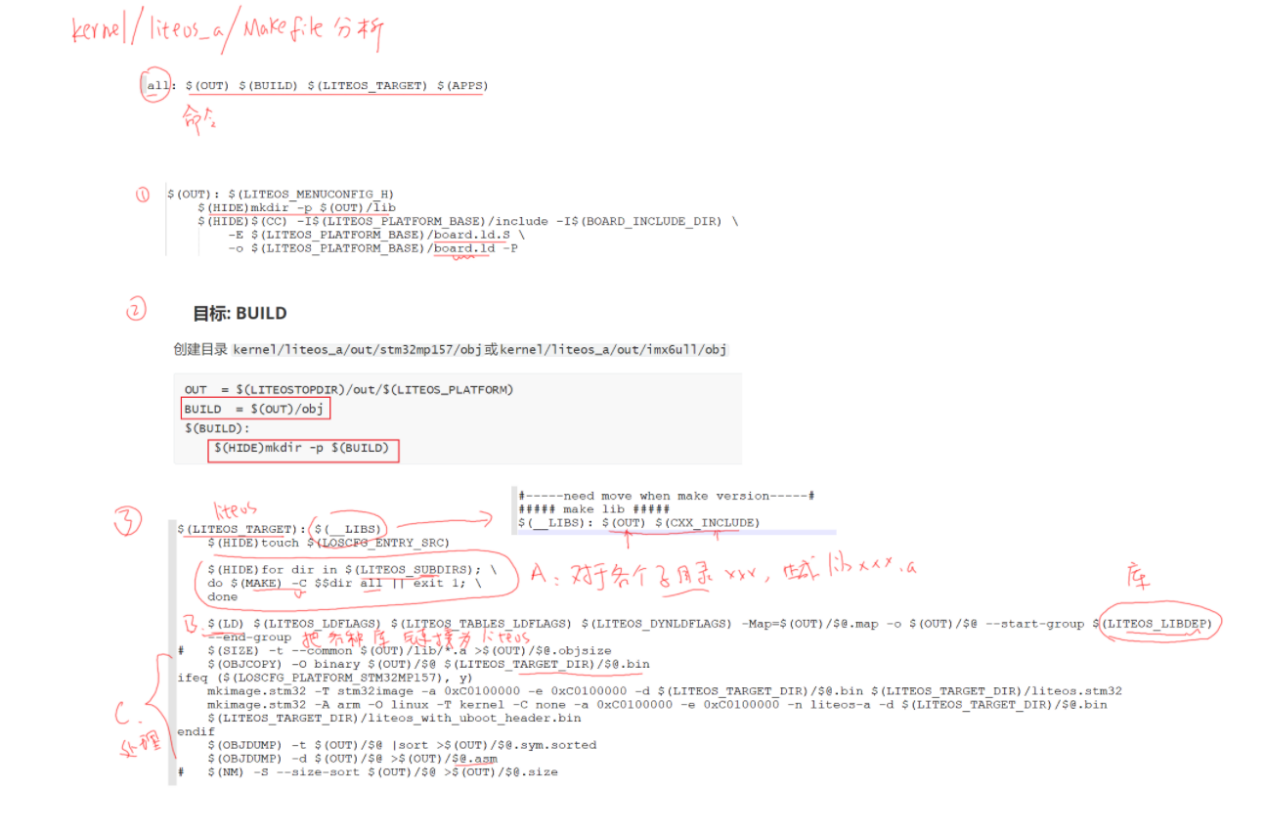
# $(SIZE) -t --common $(OUT)/lib/\*.a >$(OUT)/$@.objsize

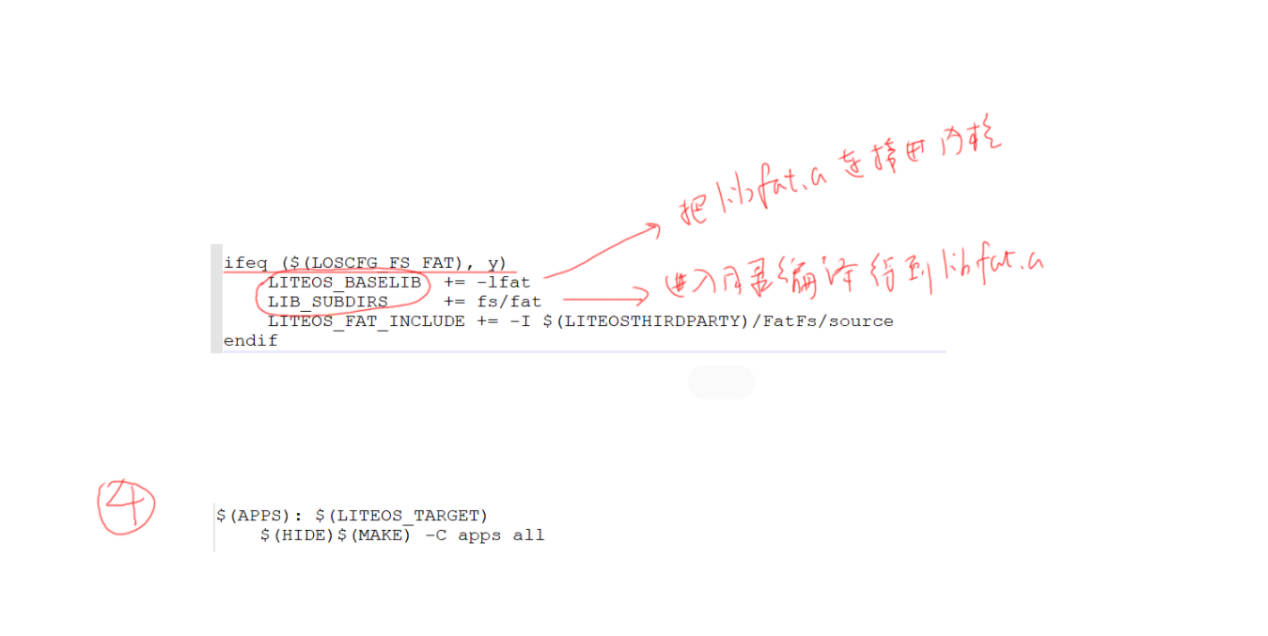
$(OBJCOPY) -O binary $(OUT)/$@ $(LITEOS\_TARGET\_DIR)/$@.bin

$(OBJDUMP) -t $(OUT)/$@ |sort >$(OUT)/$@.sym.sorted

$(OBJDUMP) -d $(OUT)/$@ >$(OUT)/$@.asm

# $(NM) -S --size-sort $(OUT)/$@ >$(OUT)[/$@.size](mailto:/$@.size)





1.4 核心总结

1.4.1 生成内核的过程

配置内核时，配置信息放在.config里，比如：LOSCFG\_FS\_FAT=y

kernel/liteos\_a/Makefile会包含很多文件，比如kernel/liteos\_a/tools/build/mk/los\_config.mk

它会看到LIB\_SUBDIRS和LITEOS\_BASELIB这两个变量

LIB\_SUBDIRS表示要进入那个子目录编译

LITEOS\_BASELIB表示要把哪些库链接进内核

比如：

```shell

ifeq ($(LOSCFG\_FS\_FAT), y)

LITEOS\_BASELIB += -lfat

LIB\_SUBDIRS += fs/fat

LITEOS\_FAT\_INCLUDE += -I $(LITEOSTHIRDPARTY)/FatFs/source

endif

```

1

2

3

4

5

6

7

编译子目录：进入LIB\_SUBDIRS指定的子目录，执行make all

对于子目录xxx的编译，一般是生成libxxx.a

链接内核 \* 把 LITEOS\_BASELIB指定的库都链接进内核

1.4.2 增加模块

在kernel/liteos\_a/Makefile能包含的文件中，增加这2个变量：

LIB\_SUBDIRS += 源码目录/xxx

LITEOS\_BASELIB += -lxxx

在源码目录/xxx下仿照fs/fat/Makefile增加Makefile

# 添加一个单板

按照编译内核的流程，添加代码。

补丁文件openharmony\_100ask\_v1.2.patch，已经添加了2个单板：STM32MP157、IMX6ULL。

目标：仿照它们，添加一个新的单板，以后修改源码让它再次支持STM32MP157或IMX6ULL。

新加的单板：

* 芯片公司：DemoCom
* 芯片名称：DemoChip

本章节做的修改会制作为补丁文件：01\_openharmony\_add\_demo\_board.patch，  
 先打补丁：openharmony\_100ask\_v1.2.patch  
 再打补丁：01\_openharmony\_add\_demo\_board.patch

假设目录openharmony中是未修改的代码，从没打过补丁；  
 假设补丁文件放在openharmony的同级目录； 打补丁方法如下：

$ cd openharmony  
$ patch -p1 < ../openharmony\_100ask\_v1.2.patch  
$ patch -p1 < ../01\_openharmony\_add\_demo\_board.patch

* 1
* 2
* 3

打上补丁后，可以如此编译：

$ cd kernel/liteos\_a  
$ cp tools/build/config/debug/demochip\_clang.config .config  
$ make clean  
$ make

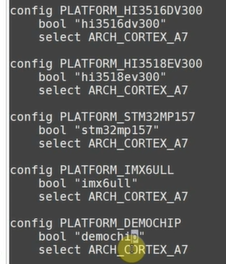
* 1
* 2
* 3
* 4

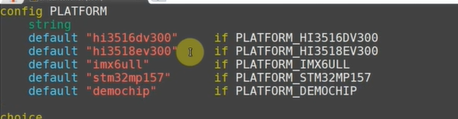
## 1.1 配置界面里添加单板

kernel/liteos\_a/platform/Kconfig

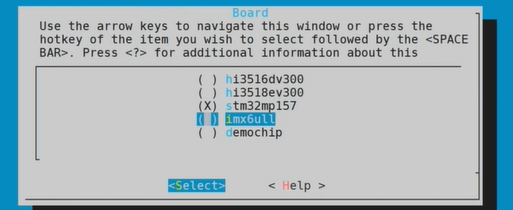
在这里插入图片描述

仿照代码添加一个新的板子：





打开menuconfig，就有了一个新的单板的配置



打开.config文件如下

在这里插入图片描述

LOSCFG\_PLATFORM="demochip"  
# LOSCFG\_PLATFORM\_HI3516DV300 is not set  
# LOSCFG\_PLATFORM\_HI3518EV300 is not set  
# LOSCFG\_PLATFORM\_STM32MP157 is not set  
# LOSCFG\_PLATFORM\_IMX6ULL is not set  
LOSCFG\_PLATFORM\_DEMOCHIP=y

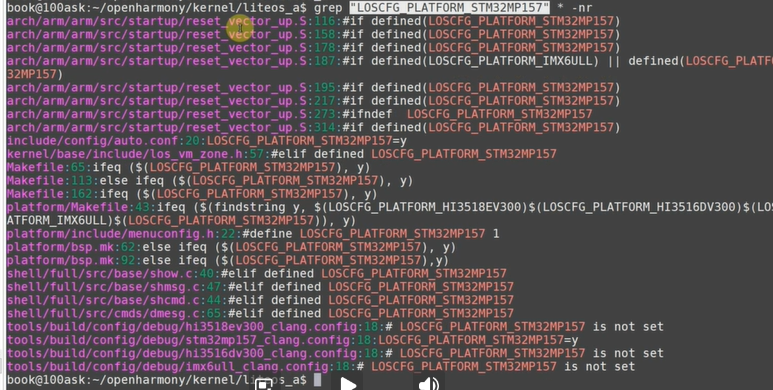
* 1
* 2
* 3
* 4
* 5
* 6

## 1.2 添加源码

刚刚配置完了config,现在要修改makefile。用一个取巧的办法就是搜索宏。参照STM32是怎么写的。

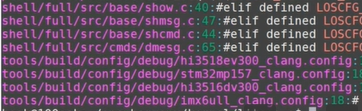
grep "LOSCFG\_PLATFORM\_STM32MP157" \* -nr

* 1

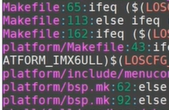


这部分是源码,这部分先不管：





先修改makefile:



在vendor目录下创建：DemoCom/DemoChip目录，里面放置文件。

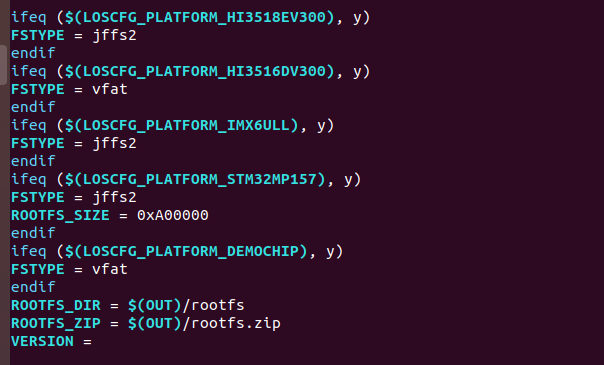
## 1.3 Makefile中根据配置项修改

Makefile  
platform/Makefile  
platform/bsp.mk

* 1
* 2
* 3

/openharmony/kernel/liteos\_a/Makefile

**第一处**

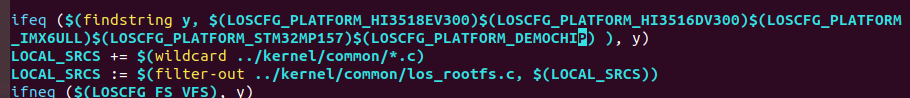


**第二处**



/openharmony/kernel/liteos\_a/platform/Makefile

**第三处**

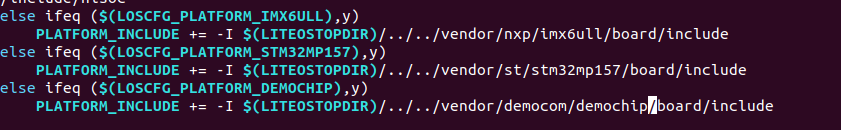


/openharmony/kernel/liteos\_a/platform/platform/bsp.mk

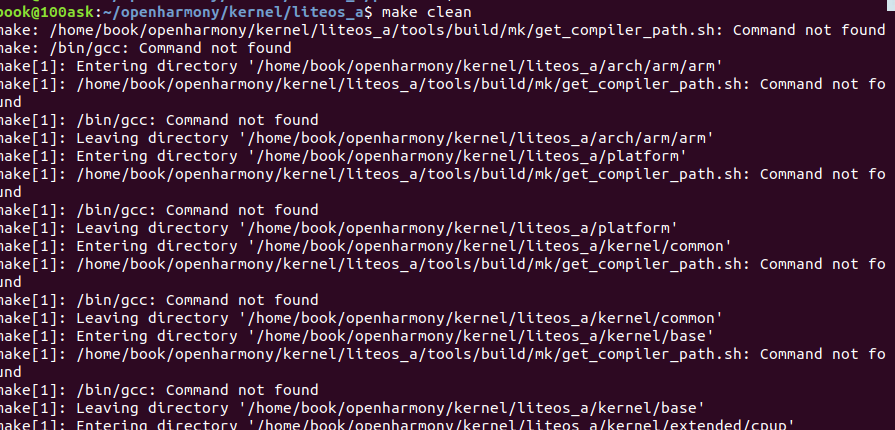
**第四处**



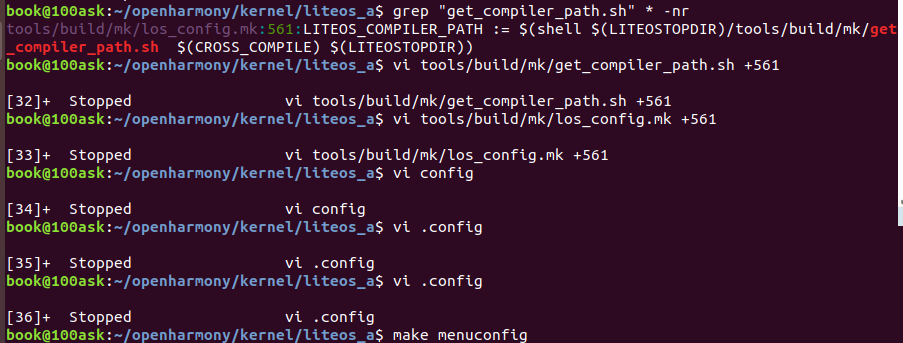
**第五处**

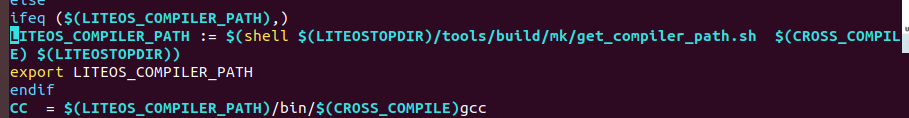


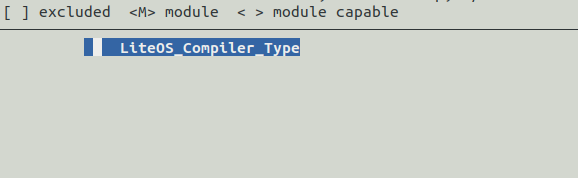
## 1.4 能否编译通过

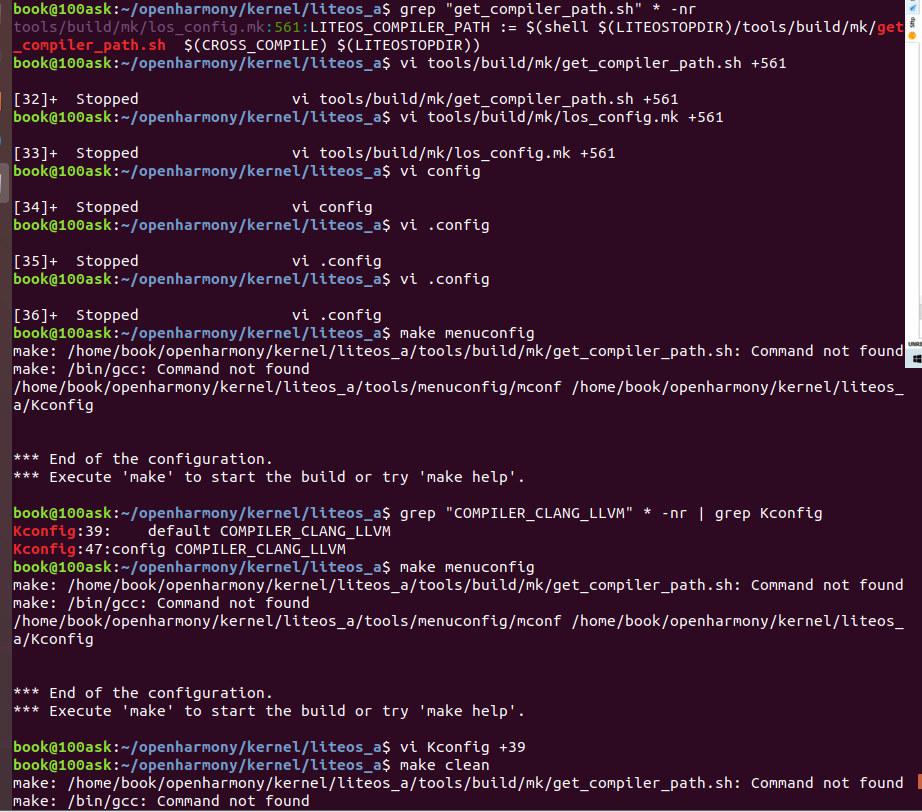


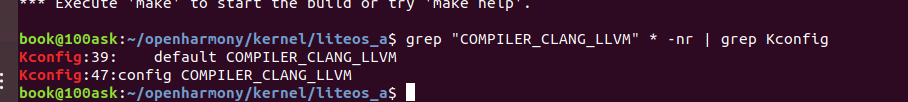
未通过：

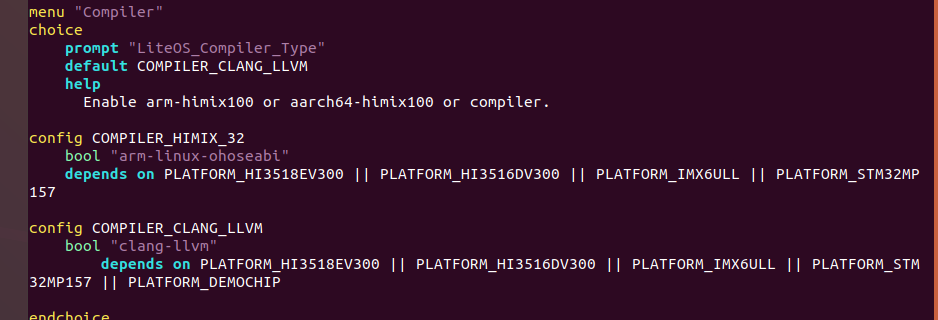












shell/full/src/base/show.c

shell/full/src/cmds/dmesg.c

shell/full/src/base/shcmd.c

shell/full/src/base/shmsg.c

## 1.5 解决链接错误

tools/build/mk/los\_config.mk

mv vendor/st/[stm32mp157](https://so.csdn.net/so/search?q=stm32mp157&spm=1001.2101.3001.7020)/stm32mp157.mk vendor/st/stm32mp157/demochip.mk

vendor/democom/hdf/hdf\_vendor.mk

drivers/hdf/lite/hdf\_lite.mk

## 1.6 内核启动流程

内核启动流程可以分为4步骤(非官方)：

* 启动

* 使用汇编代码编写，涉及非常底层的设置，比如CPU设置、代码重定位等等
* 地址映射也在这里实现
* 它最终会调用main函数
* main函数

* 以后的代码，基本都是使用C语言编写了
* 主要工作是：调用OsMain进行各类初始化、最终会启动用户程序
* OsMain函数

* 进行操作系统层面的初始化，比如异常初始化、任务初始化、IPC初始化
* 调用SystemInit
* SystemInit

* 偏向于应用程序的初始化
* 挂载根文件系统
* 启动第一个用户进程

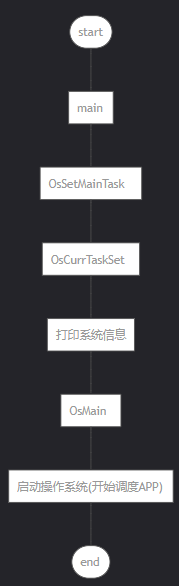
### 1.6.1 启动文件分析

从kernel\liteos\_a\arch\arm\arm\src\startup\reset\_vector\_up.S开始阅读代码，  
 流程如下：



### 1.6.2 main函数分析

main函数在这个文件里kernel\liteos\_a\platform\main.c，  
 流程如下：



### 1.6.3 OsMain函数分析

OsMain函数在这个文件里kernel\liteos\_a\kernel\common\los\_config.c，  
 流程如下：



### 1.6.4 SystemInit函数分析

SystemInit函数在这个文件里vendor\st\stm32mp157\board\board.c，  
 流程如下：

