嵌入式Linux学习笔记

开发板：正点原子阿尔法I.MX6ULL（EMMC版）

参考手册：《I.MX6ULL参考手册》

**目录**

[**第一部分** **开发环境搭建** 2](#_Toc156776413)

[**1. 安装VI/VIM编辑器** 2](#_Toc156776414)

[**2. Ubuntu 和 Windows 文件互传** 2](#_Toc156776415)

[**3. 开启SSH服务** 3](#_Toc156776416)

[**4. 解压缩指令** 3](#_Toc156776417)

[**5. 文件操作指令** 3](#_Toc156776418)

[**6. 安装GCC交叉编译器** 4](#_Toc156776419)

[**第二部分** **裸机开发** 4](#_Toc156776420)

[**1. 学习Cortex-A汇编的目的** 4](#_Toc156776421)

[**2. IO初始化流程** 4](#_Toc156776422)

[**3. 程序编译步骤** 5](#_Toc156776423)

[**4. 烧写代码** 5](#_Toc156776424)

[**5. Ubuntu断网解决方法**（无网络图标） 5](#_Toc156776425)

[**6. 启动方式选择** 5](#_Toc156776426)

[**7. 启动设备选择** 6](#_Toc156776427)

[**8. BOOT ROM的作用** 6](#_Toc156776428)

[**9. 镜像文件介绍** 6](#_Toc156776429)

[**10. C语言运行环境构建** 6](#_Toc156776430)

[**11. 三极管基础** 7](#_Toc156776431)

[**12. 链接脚本** 8](#_Toc156776432)

[**13. Makefile基础** 9](#_Toc156776433)

[**14. 时钟系统** 12](#_Toc156776434)

[**15. 中断** 16](#_Toc156776435)

[**16. Ubuntu网络IP改为静态** 22](#_Toc156776436)

[**17. EPIT定时器** 23](#_Toc156776437)

[**18. GPT定时器** 23](#_Toc156776438)

[**19. UART串口** 24](#_Toc156776439)

[**20. DDR3** 25](#_Toc156776440)

[**21. RGBLCD** 28](#_Toc156776441)

[**22. RTC实时时钟** 32](#_Toc156776442)

[**23. NXP官方SDK移植** 33](#_Toc156776443)

[**24. I2C** 33](#_Toc156776444)

[**25. SPI** 37](#_Toc156776445)

[**26. 电容触摸屏** 41](#_Toc156776446)

[**27. IO属性配置** 42](#_Toc156776447)

[**28. 上下拉电阻** 44](#_Toc156776448)

[**29. PWM** 45](#_Toc156776449)

[**第三部分 LINUX系统移植** 47](#_Toc156776450)

[**1. Linux系统构成**（bootloader、Linux内核和 根文件系统） 47](#_Toc156776451)

[**2. 固化系统**（阿尔法EMMC开发板） 47](#_Toc156776452)

[**3. U-Boot简介** 48](#_Toc156776453)

[**4. 编译U-Boot** 49](#_Toc156776454)

[**5. U-Boot命令** 50](#_Toc156776455)

**第一部分 开发环境搭建**

1. **安装VI/VIM编辑器**

在Ubuntu下可以使用VI/VIM编辑器对文本进行编辑或修改。Ubuntu下自带了VI编辑器，但VI编辑器不能使用键盘上的上下左右键调整光标位置，所以推荐使用VIM编辑器。VIM是一种基于指令式的编辑器，是VI的升级版。VIM不需要使用鼠标，也没有菜单，仅仅使用键盘来完成所有的编辑工作，终端输入sudo apt-get install vim即可安装。

**编辑指令**： ①进入编辑模式：a

②退出编辑：ESC按键

③保存并退出：wq

④退出但不保存：q!

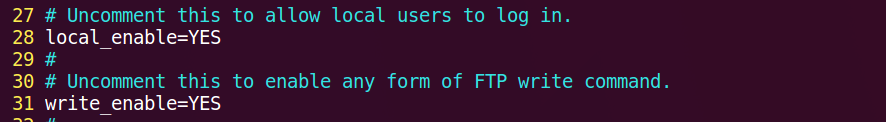
1. **Ubuntu 和 Windows 文件互传**

使用FileZilla 进行Ubuntu和Windows的文件互传，FileZilla为FTP客户端，使用前需要在Ubuntu中开启FTP服务。**开启Ubuntu的FTP服务**：

①终端输入sudo apt-get install vsftpd，安装FTP服务。

②输入sudo vi /etc/vsftpd.conf，打开配置文件。

③在打开的文件中找到write\_enable=YES，取消注释后保存并退出。



④输入sudo /etc/init.d/vsftpd restart重启 FTP 服务

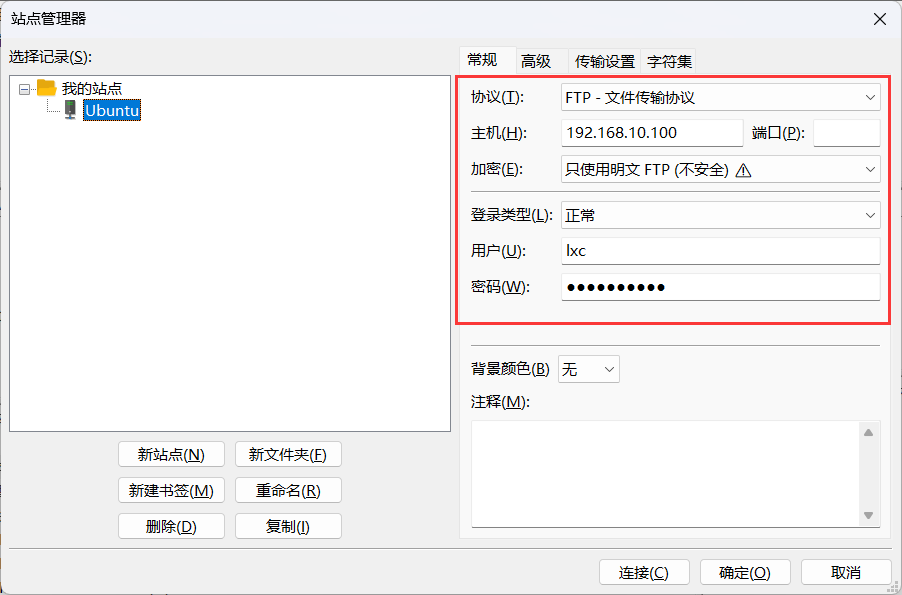
**FileZilla设置步骤**：

①官网下载FileZilla客户端，网址为：[下载 - FileZilla中文网](https://www.filezilla.cn/download)。



②进入FileZilla，点击菜单栏中文件->站点管理器，按下图配置新建站点，主机设置为Ubuntu网络地址，字符集

中设置强制UTF-8。



③保存设置，点击链接即可。

1. **开启SSH服务**

开启 Ubuntu 的 SSH 服务以后，我们就可以在 Windwos 下使用终端软件登陆到 Ubuntu，比如使用 SecureCRT，Ubuntu 下使用命令sudo apt-get install openssh-server开启 SSH 服务，ssh 的配置文件为/etc/ssh/sshd\_config，使用默认配置即可。

1. **解压缩指令**

Linux下最常用的压缩格式为.bz2 和.gz，tar指令可对两种形式的压缩包进行压缩和解压缩，命令格式如下。

tar [参数] [压缩文件名] [被压缩文件名]

**示例**： ①tar -vcjf test1.tar.bz2 test1 //将test1压缩为.bz2格式

②tar -vczf test1.tar.gz test1 //将test1压缩为.gz格式

③tar -vxjf test1.tar.bz2 //对.bz2格式压缩包进行解压

④tar -vxzf test2.tar.gz //对.gz格式压缩包进行解压

**注**： **-v**：显示指令执行过程  **-j**：用 tar 生成压缩文件，然后用 bzip2 进行压缩

**-f**：< 备份文件> 指定压缩文件  **-z**：用 tar 生成压缩文件，用 gzip 压缩

**-x**：从备份文件中释放文件，也就是解压缩文件  **-c**：创建新的压缩文件

1. **文件操作指令**

①**touch**（创建文件）

touch指令用于创建文件。除了创建文本文件外，还可创建其他类型文件，如创建test文件：touch test。

②**mkdir**（创建文件夹）

mkdir指令用于创建文件夹。如创建文件夹dirtest：mkdir dirtest。

③**rm**（删除文件或文件夹）

rm用于删除文件或文件夹。如删除文件test：rm test ，删除文件夹时要加参数-rf，如删除文件夹dirtest：

rm dirtest -rf。

④**rmdir**（删除文件夹）

rmdir指令用于删除文件夹，且不需要加参数，如删除文件夹test：rmdir test。

⑤**cp**（文件复制）

cp指令用于复制文件。如文件夹test1下有1个文件a.c，现复制a.c到本文件夹，并命名为b.c，即cp a.c b.c；现将a.c和b.c复制到上级目录中的文件夹test2中：cp \*.c ../test2；复制文件夹时要加参数-rf，如复制复制文件夹test2到当前路径，并命名为test3：cp -rf test2/ test3/。

⑥**mv**（移动文件/重新命名）

mv指令用于移动、重命名文件或文件夹。如文件夹test1中有文件a.c，现将a.c改名为b.c：mv a.c b.c；现将b.c移动到上级目录中的文件夹test2中：mv b.c ../test2。

**注**： -**r** ：递归删除，指定文件夹(目录)下所有文件和子文件夹全部删除；

**-f ：**强制删除文件和文件夹(目录)

1. **安装GCC交叉编译器**

ARM 裸机、Uboot 移植、Linux 移植都需要在Ubuntu下进行编译，Ubuntu自带的GCC编译器是针对X86架构的，而我们要编译的是ARM架构的代码，所以我们需要使GCC能够在X86的PC上编译ARM结构的代码，即GCC交叉编译器。使用Linaro出品的交叉编译器：Linaro GCC，安装步骤如下。

①可去Linaro官网下载Linaro GCC，具体操作看相关教程，我直接使用正点原子提供的安装包，路径如下。

05.开发工具\01.交叉编译器gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar

②使用FileZilla将安装包发送到Ubuntu。

③在路径usr/local下进入终端，输入sudo mkdir arm创建arm文件夹。

④进入安装包所在文件夹，打开终端。

⑤输入sudo cp gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar.xz /usr/local/arm/将安装包复制到arm文件夹中。

⑤进入arm文件夹，输入sudo tar -vxf gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf.tar.xz解压安装包。

⑥可看到生成了gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf文件夹。

⑦终端输入sudo vi /etc/profile修改环境变量，最下方输入如下指令，修改完成后保存退出，输入reboot重启系统。

export PATH=$PATH:/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf/bin

⑧终端输入arm-linux-gnueabihf-gcc -v，若出现版本信息，则说明安装成功。

**第二部分 裸机开发**

1. **学习Cortex-A汇编的目的**

①初始化一些SOC外设

②初始化DDR（I.MX6U不需要）

③设置sp指针，一般指向DDR，设置好C语言运行的环境

**注**：开发常用汇编：存储器访问指令示例如下。

假设变量a的地址为0X20，变量b的地址为0X30，将b的值赋值给a

LDR R0, =0X30

LDR R1, [R0]

LDR R0, =0X20

STR R1, [R0]

**LDR**：从存储器读取数据 **STR**：向存储器写入数据

**R0、R1**：ARM汇编指令不能直接从RAM里读数据，所以需要借用寄存器R0、R1等通用寄存器。

1. **IO初始化流程**

**STM32的IO初始化**：

①使能GPIO时钟

②设置IO复用，默认为GPIO

③配置GPIO电气属性

④使用GPIO输出高低电平

**I.MX6U的IO初始化**：（IO属性配置信息见[IO属性配置](#IO初始化配置)）

①使能时钟，CCGR0 ~ CCGR6这7个寄存器控制所有外设时钟的使能，为方便起见，设置这七个寄存器全部 为0XFFFFFFFF，即使能所有外设时钟。（见参考手册699页）

②IO复用，点灯为例，GPIO1\_IO03为控制引脚，IOMUXC\_SW\_MUX\_CTL\_PAD\_GPIO1\_IO03为复用寄存器， 设置为0101，此时GPIO1\_IO03就复用为GPIO。（见参考手册1571页）

③配置电气属性，IOMUXC\_SW\_PAD\_CTL\_PAD\_GPIO1\_IO03为设置GPIO1\_IO03电气属性的寄存器，主要 设置：压摆率SRE、驱动能力DSE、速度SPEED、上下拉PUE、开漏ODE等。（见参考手册1793页）

④配置GPIO功能，配置GPIOx\_GDIR寄存器bit3为1，即设置GPIO1\_IO03为输出模式，配置GPIOx\_DR寄 存器bit3，设置输出电平值，为1表示高电平，为0表示低电平。（见参考手册1357页）

1. **程序编译步骤**

①将.c、.s文件编译为.o文件（使用指令：**arm-linux-guneabihf-gcc**）

如使用指令：arm-linux-guneabihf-gcc -g -c leds.s -o led.o 将led.s文件编译为led.o。

**-g**：产生调试信息 **-c**：编译源文件但不链接 **-o**：指定编译产生的文件名

②将所有的.o文件链接为elf格式的可执行文件（使用指令：**arm-linux-guneabihf-ld**）

如使用指令：arm-linux-guneabihf-ld -Ttext 0X87800000 led.o -o led.elf 将led.o文件链接到0X87800000这个地址上。

**-Ttext**：指定链接地址 **-o**：指定链接生成的elf文件名

**链接**：将众多的.o 文件链接到一个指定的链接位置，即链接起始地址。

**链接起始地址**：代码运行的起始地址。对于I.MX6ULL来说，链接起始地址指向RAM地址，RAM分为内部RAM和 外部RAM（DDR），内部RAM地址为：0X900000 ~ 0X91FFFF，正点原子的阿尔法EMMC版外部RAM地

址为： 0X0X80000000 ~ 0X9FFFFFFF。后面要学习的Uboot的链接起始地址是0X87800000，为了不易记混，所以统一使用DDR的0X87800000作为链接起始地址。

③将elf文件转为bin文件（使用指令：**arm-linux-guneabihf-objcopy**）

如使用指令：arm-linux-guneabihf-objcopy -O binary -S -g led.elf led.bin 将led.elf文件转换为led.bin 文件。

**-O**：指定以什么格式输出 **binary**：表示以二进制格式输出

**-S**：表示不复制源文件中的重定位信息和符号信息 **-g**：表示不复制源文件中的调试信息

**注**：调试C语言时，可将elf文件转为汇编，即反汇编。（使用软件：**arm-linux-gnueabihf-objdump**）

如使用指令：arm-linux-gnueabihf-objdump -D led.elf > led.dis 将elf文件反汇编为dis文件。

**-D**：表示反汇编所有的段

1. **烧写代码**

I.MX6U支持从外置的NOR Flash、NAND Flash、SD/EMMC、SPI NOR Flash和QSPI Flash这些存储介质中启动，上电以后I.MX6U的内部BOOT ROM程序会将可执行文件拷贝到链接地址处。除了SD卡以外，其他的一般都是焊接到板子上的，无法直接烧写，所以选择将代码烧写到SD卡。烧步骤如下：

①准备工具：正点原子提供的imxdownload软件

②确定要烧写的SD卡：终端输入指令ls /dev/sd\*/ -l查看，如sdb即为SD卡

③给imxdownload软件可执行权限：chmod 777 imxdownload

④开始烧写：./imxdownload led.bin /dev/sdb

⑤烧成成功后将SD卡插到开发板，将开发板上的BOOT\_CFG拨码开关调为SD卡启动（1、7上拨，其余下拨）

**注**：可发现当前文件夹下生成了load.imx文件，因为imxdownload会向led.bin添加一个头部，从而生成了load.imx文件，此 文件才是烧写到SD卡中的文件。

1. **Ubuntu断网解决方法**（无网络图标）

①打开终端

②输入：sudo service NetworkManager stop

③输入：sudo gedit /var/lib/NetworkManager/NetworkManager.state

④文件内将 NetworkingEnabled=false 修改为 NetworkingEnabled=true

⑤输入：sudo service NetworkManager start

⑥关闭终端

1. **启动方式选择**

I.MX6U 芯片上电以后，芯片会根据BOOT\_MODE[1:0]的设置来选择BOOT方式。BOOT\_MODE[1:0]的值有两种改变方式，一种是改写 eFUSE(熔丝)，一种是修改相应的GPIO高低电平。第一种修改eFUSE的方式只能修改一次，后面就不能再修改了，所以我们不使用。我们使用的是通过修改 BOOT\_MODE[1:0]对应的GPIO高低电平来选择启动方式。I.MX6U有四种BOOT 模式，00：从FUSE启动、01：串行下载、10：内部BOOT模式、11：保留。只使用第二、三种，介绍如下：

①**串行下载**： BOOT\_MODE1为0，BOOT\_MODE0为1，通过USB或者UART将代码下载到板子上的外置存储设备 中，这个下载是需要用到 NXP 提供的一个软件，一般用来在最终量产的时候将代码烧写到外置存储设备中。

②**内部BOOT模式**：BOOT\_MODE1为1，BOOT\_MODE0为0，芯片会执行内部的BOOT ROM代码进行硬件初始化

(一部分外设)，然后从BOOT设备(存放代码的设备，如SD/EMMC、NAND)中将代码拷贝出来复制到指定RAM （一般是DDR）中。

1. **启动设备选择**

启动方式设置为内部BOOT模式后，I.MX6U支持从外置的NOR Flash、NAND Flash、SD/EMMC、SPI NOR Flash和QSPI Flash这些存储介质中启动。通过配置BOOT\_CFG1[7:0]、BOOT\_CFG2[7:0]和BOOT\_CFG4[7:0]这24个IO进行设备选择，同样有两种配置方式：改写eFUSE和修改GPIO，我们选择修改GPIO。这24个IO不需要全部配置，开发板中需要配置的IO为： BOOT\_CFG2[3]、BOOT\_CFG1[3] ~ BOOT\_CFG1[7]。加上BOOT\_MODE[1:0]一共8位IO就可完成启动设备的选择，设置方式如下：

**编号**：1：BOOT\_MODE[1] 2：BOOT\_MODE[0] 3：BOOT\_CFG2[3] 4：BOOT\_CFG1[3]

5：BOOT\_CFG1[4] 6：BOOT\_CFG1[5] 7：BOOT\_CFG1[6] 8：BOOT\_CFG1[7]

**设置1 ~ 8**： 01xxxxxx：串行下载，可以通过USB烧写镜像文件

01100110：USB启动

10000010：SD卡启动

10100110：EMMC启动

10001001：NAND FLASH启动

1. **BOOT ROM的作用**

①初始化时钟，将内核时钟为396MHz，此外还设置System PLL=528Mhz，USB PLL=480MHz，AHB=132MHz，

IPG=66MHz，具体内容在[时钟系统](#时钟系统)中介绍。（见参考手册261页）

②使能内存管理单元（MMU）和缓存去提高启动速度。下载镜像时打开L1指令缓存；验证镜像时打开L1数据缓存、 L2缓存和MMU；镜像验证完成时关闭L1数据缓存、L2缓存和MMU。（见参考手册263页）

③从寄存器BOOT\_CFG设置的外部存储中读取镜像，然后做相应处理（初始化DDR、启动时必要的硬件设置等）。

1. **镜像文件介绍**

STM32可烧写编译生成的.bin文件，但I.MX6U不能直接烧写.bin文件，需要在.bin文件前面添加一些头部信息，构成

最终可烧写的文件，即.imx文件。.imx文件由四部分组成：IVT + Boot data + DCD + 用户代码及数据。其中ROM将IVT固定在启动设备中一个固定的地址，该地址根据启动设备决定，其余三部分的地址由IVT决定。

①**Image vector table（IVT）**：镜像向量表，大小为32个字节。包含镜像程序的入口点和Boot Data、DCD地址定位

的信息，BOOT ROM根据这些地址定位信息找到对应部分的存储位置。

②**Boot data**：启动数据，大小为12个字节。设置镜像数据的大小和要拷贝到的目标地址等。

③**Device configuration data（DCD）**：设备配置信息，大小不能超过1768字节。BOOT ROM使用DCD中的配置参

数来初始化DDR，配置时钟，以及其他启动时必要的硬件设置。

④**用户代码及数据**，如led.bin。

启动设备以SD卡为例，初始载荷区域大小为4K字节，即启动时BOOT ROM会拷贝SD卡中前4K字节到片内RAM中，这4K字节中必须包含IVT、Boot data和DCD，BOOT ROM就是根据这4K字节的信息对镜像文件做相应的处理。其中前1K字节固定保存SD卡的分区表信息，是不可以改变的，所以IVT、Boot data和DCD只能存储在后3K字节中。IVT固定存储在最前面，Boot data其次，DCD接在Boot data后面。（见参考手册309页）

1. **C语言运行环境构建**

实际工作中，很少用汇编写驱动，大部分情况下是先使用汇编初始化C语言环境，然后使用C语言编写驱动。汇编初始化C语言环境内容如下，具体指令可查看[start.S文件](#C语言环境配置文件)：

①**设置I.MX6ULL处于SVC运行模式下**。通过配置CPSR状态寄存器的M[4:0]位为10011，即0X13，配置状态寄存

器需要使用MRS和MSR指令。汇编指令如下：

mrs r0, cpsr /\* 读取cpsr到r0 \*/

bic r0, r0, #0x1f /\* 清除cpsr的M[4:0]位 \*/

orr r0, r0, #0x13 /\* 选择SVC模式 \*/

msr cpsr, r0 /\* 将r0写入到cpsr中，进入SVC模式\*/

BIC Rd, Rn, #immed @Rd = Rn & (~#immed) 位清除

ORR Rd, Rn, #immed @Rd = Rn | #immed 按位或

MRS R0, CPSR @将特殊寄存器CPSR里面的数据传递给R0，即 R0=CPSR

MSR CPSR, R0 @将R0中的数据复制到CPSR中，即 CPSR=R0

②**设置sp堆栈指针**。我们设置指针指向DDR，栈大小设置为0X200000 = 2MB。Cortex-A7 架构中栈的增长方式为向 下增长（从高地址向低地址增长），本开发板DDR范围为0X80000000 ~ 0X9FFFFFFF，所以设置sp指针指向 0X80200000。汇编指令为：ldr sp, =0x80200000

③**跳转到C语言main函数**。汇编指令为：b main

**注**：1）**运行模式**

I.MX6ULL使用的是Cortex-A7 架构，Cortex-A7架构的一个优点是功耗低。Cortex-A有9种运行模式，介绍如下。其中除了用户模式外，其他8种模式均为特权模式，可由软件任意切换，也可通过中断或异常进行切换；用户模式只能通过中断或异常进行模式切换，用户模式下不能访问系统的所有资源，有些资源是受限的。

a) User(USR)：用户模式

b) FIQ：中断模式，用于进入FIQ中断异常

c) IRQ：一般中断模式

d) Supervisor(SVC)：超级管理员模式，供操作系统使用

e) Monitor(MON)：监视模式，用于安全扩展

f) Abort(ABT) ：数据访问终止模式，用于虚拟存储以及存储保护

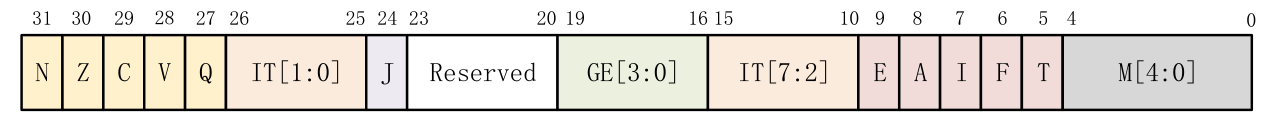
g) Hyp(HYP)：超级监视模式，用于虚拟化扩展

h) Undef(UND)：未定义指令终止模式

i) System(SYS) ：系统模式，用于运行特权级的操作系统任务

2）**CPSR寄存器**

当前程序状态寄存器，该寄存器包含了条件标志位、中断禁止位、当前处理器模式标志等一些状态位以及一些控制位。结构图如下，其中M[4:0]为处理器模式控制位，配置信息如下：



10000：User模式 10001：FIQ模式 10010：IRQ模式

10011：Supervisor(SVC)模式 10110：Monitor(MON)模式 10111：Abort(ABT)模式

11010：Hyp(HYP)模式 11011：Undef(UND)模式 11111：System(SYS)模式

3）读取特殊寄存器（如CPSR和SPSR）的数据时只能使用MRS和MSR这两个指令。

4）有的芯片在设置sp指针之前需要初始化DDR，对于本芯片，在生成的.imx文件中的头部信息中已经完成了DDR的 初始化，所以这里直接设置sp指针就可以了。

1. **三极管基础**

信号放大是对模拟信号最基本的处理，三极管是放大电路的核心元件。三极管的放大作用表现为小的基极电流可以控制大的集电极电流。如基本共射放大电路图所示，为输入电压信号,它接入基极-发射极回路，称为输入回路；放大后的信号在集电极-发射极回路,称为输出回路。由于发射极是两个回路的公共端，故称该电路为共射放大电路。因为晶体管工作在放大状态的外部条件是发射结正向偏置且集电结反向偏置,所以在输入回路应加基极电源；在输出回路应加集电极电源。和的极性应如下图所示，且大于，即晶体管的放大作用表现为小的基极电流可以控制大的集电极电流。

图示

描述已自动生成图示, 工程绘图, 示意图

描述已自动生成 图示, 示意图

描述已自动生成 图示, 示意图

描述已自动生成

**NPN型硅管的结构 NPN型管的结构示意图 NPN型和PNP型管的符号 基础共射放大电路**

NPN型三极管与PNP型三极管区别如下：

①**电流方向**

NPN：电流两进一出，以b→e电流控制c→e电流。 正常放大时，即Vc > Vb > Ve

PNP：电流一进两出，以e→b电流控制e→c电流。 正常放大时，即Ve > Vb > Vc

②**作为开关的常用方式**

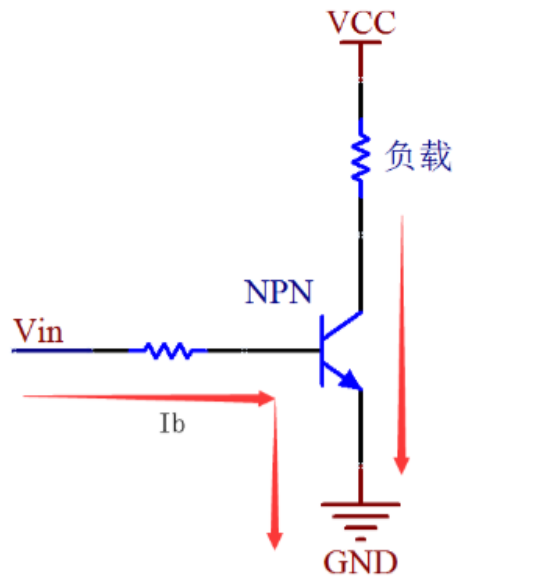
三极管做开关时，有两种工作状态：截止状态、饱和状态，一般是通过控制三极管的基极电压Ub来控制三极管

的导通与断开。三极管导通与否一般看基极电位和发射极电位的电压差，所以要保证发射极的电压的状态是确定的，具体对比如下。

NPN：基极电位高于发射极电位一定值时导通，一般发射级接GND。

PNP：基极电位低于发射极电位一定值时导通，一般发射级接VCC。

使用三极管驱动负载时，用NPN型的晶体管来控制接地的引线（作为上管），用PNP型来控制接VCC的引线（作为下管），NPN与PNP驱动电路如下。

 图示, 示意图

描述已自动生成

**NPN驱动电路 PNP驱动电路**

1. **链接脚本**

链接代码时，可使用-Text 指定链接地址，如arm-linux-gnueabihf-ld -Ttext 0X87800000 -o ledc.elf $^可将所有的文件链接到以0X87800000为起始地址的区域。但有时需要将不同文件链接到指定地址区域，此时就需要使用链接脚本。我们编译出来的代码一般都包含在text、data、bss和rodata段内。最简单的链接脚本可以只包含一个命令“SECTIONS”,我们可以在这一个“SECTIONS”里面来描述输出文件的内存布局，示例如下。

SECTIONS{

. = 0X87800000; #定位计数器，指明链接地址为0X87800000

.text : #.text为段名，注意.text与：之间要加空格

{ #括号内包含要放入text段内的文件

obj/start.o #start.o包含第一个要执行的指令，所以要链接到最前面

obj/main.o

\*(.text)

}

.rodata ALIGN(4) : {\*(.rodata\*)} #.rodata段，将所有的.rodata文件放入段内

.data ALIGN(4) : { \*(.data) } #.data段，将所有的.data文件放入段内，ALIGN(4)表示4字节对齐

. = ALIGN(4); #对\_\_bss\_start起始地址进行四字节对齐，防止删除bss段时出现BUG

\_\_bss\_start = . ; #保存定位符‘.’的地址作为bss的起始地址，注意‘=’前后要有空格

.bss ALIGN(4) : { \*(.bss) \*(COMMON) } #.bss数据是那些定义了但是没有被初始化的变量

\_\_bss\_end = . ; #保存定位符‘.’的地址作为bss的结束地址，注意‘=’前后要有空格

} #后面就可以通过\_\_bss\_start和\_bss\_end对bss段进行清零操作

将链接脚本保存为imx6ul.lds，则将上述指令改写为arm-linux-gnueabihf-ld -Timx6ul.lds -o ledc.elf $^即可将指定文件链接到指定地址区域。

1. **Makefile基础**

Makefile定义了一系列的规则来指定：哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。Makefile一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。Makefile关系到了整个工程的编译规则，会不会写Makefile从侧面说明了一个人是否具备完成大型工程的能力。

①**程序的编译和链接**

关于程序编译，源文件首先会生成中间目标文件（Object File），再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成Object File。而在链接程序时，链接器会在所有的Object File中找寻函数的实现，如果找不到，那就会报链接错误码（Linker Error）。

②**Makefile效果**

1. 如果这个工程没有编译过，那么我们的所有C文件都要编译并被链接。
2. 如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。
3. 如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。

③**示例**（第一条规则的目标文件为Makefile的最终目标文件，若第一条规则存在多个目标，则第一个目标为最终目标）

main: main.o input.o calcu.o #第一条规则，main为目标文件，main.o input.o calcu.o为依赖文件

gcc -o main main.o input.o calcu.o #若依赖文件存在，则执行指令，将依赖文件链接生成目标文件main

main.o: main.c #第二条规则，main.o为目标文件，main.c为依赖文件

gcc -c main.c #若依赖文件存在，则将依赖文件编译生成目标文件main.o

input.o: input.c #第三条规则，input.o为目标文件，input.c为依赖文件

gcc -c input.c #若依赖文件存在，则将依赖文件编译生成目标文件input.o

calcu.o: calcu.c #第四条规则，calcu.o为目标文件，calcu.c为依赖文件

gcc -c calcu.c #若依赖文件存在，则将依赖文件编译生成目标文件calcu.o

clean: #第五条规则，clean为目标，由于没有目标文件，所以其对应的命令需要使用make clean才会执行

rm \*.o #清除左右的.o文件

rm main #清除main文件

执行流程为： 1）输入make指令

2）make指令在当前目录下寻找以Makefile或makefile命名的文件

3）查看Makefile的目标文件是否存在且最新

4）若目标文件不存在或依赖文件比目标文件新，则执行相应指令，更新目标文件

GCC指令：**-c：**只编译不链接为可执行文件，编译器将输入的.c 文件编译为.o 的目标文件

-**o：**用来指定编译结束以后的输出文件名，GCC默认编译出来的可执行文件名字为a.out

-**O：**对程序进行优化编译，整个源代码在编译、链接的的时候都会进行优化，产生的可执行文件执行效率高

-**O2**：比-O幅度更大的优化，生成的可执行效率更高，但是整个编译过程会更慢

④**Makefile变量**

Makefile 中的变量都是字符串，将下面指令使用变量可修改如下：

修改前：

main: main.o input.o calcu.o

gcc -o main main.o input.o calcu.o

修改后：

objects = main.o input.o calcu.o #定义了一个变量objects，后面可用objects表示后面三个文件

main: $(objects) #使用$引用变量objects，引用方法为$(变量名)

gcc -o main $(objects) #同理，使用$引用变量objects

⑤**赋值符**

1. = ：变量的真实值取决于它所引用的变量的最后一次有效值
2. := ：变量的真实值取决于它当前赋予的有效值，不会使用后面定义的变量
3. ?= ：变量的真实值取决于之前的赋值，若之前赋值了，则使用之前的值，否则使用本次的赋值
4. += ：变量追加

示例： name = lxc #定义变量name，变量赋值为lxc

curname = $(name) #使用 = 将变量赋值给了curname，此时curname的值为lxc

name = luoxingchi #重新给name赋值，此时curname的值也改为了luoxingchi

name = lxc #定义变量name，变量赋值为lxc

curname := $(name) #使用 := 将变量赋值给了curname，此时curname的值为lxc

name = luoxingchi #重新给name赋值，此时curname的值仍使用赋值时的值，为lxc

name = lxc #定义变量name，变量赋值为lxc

name ?= LXC #由于上面已经给变量name赋值为lxc，所以此时name的值仍为lxc，

#若上面未赋值，则此时name的值为LXC

name = lxc #定义变量name，变量赋值为lxc

name += LXC #使用 += 变量追加，此时name的值为lxcLXC

⑥**自动化变量**

自动化变量把模式中所定义的一系列的文件自动的挨个取出，直至所有的符合模式的文件都取完。自动化变量只应该出现在规则的命令中。常用的自动化变量有三种：$@、$<、$^。

1. $@：规则中的目标集合，在模式规则中，如果有多个目标的话，“$@”表示匹配模式中定义的目标集合。
2. $< ：依赖文件集合中的第一个文件，如果依赖文件是以模式(即“%”)定义的，那么就是符合模式的一系列的 文件集合。
3. $^ ：所有依赖文件的集合，使用空格分开，如果在依赖文件中有多个重复的文件，“$^”会去除重复的依赖文 件，只保留一份。

**示例**： objects = main.o input.o calcu.o #定义变量，使用objects表示后面的三个.o文件

main: $(objects) #使用$引用objects作为依赖文件，即三个.o文件为依赖文件

gcc -o main $(objects) #将依赖文件链接为main文件

%.o : %.c #%.o表示所有的.o文件，%.c表示所有的.c文件

gcc -c $< #$<表示依赖文件的集合，即将对应的.c文件依次带入，编译生成对应的.o文件

⑦**例程通用Makefile**

CROSS\_COMPILE ?= arm-linux-gnueabihf- #交叉编译工具

TARGET ?= beep #要生成的目标文件名

CC := $(CROSS\_COMPILE)gcc #编译指令：arm-linux-gnueabihf-gcc

LD := $(CROSS\_COMPILE)ld #链接指令：arm-linux-gnueabihf-ld

OBJCOPY := $(CROSS\_COMPILE)objcopy #格式转换指令：arm-linux-gnueabihf-objcopy

OBJDUMP := $(CROSS\_COMPILE)objdump #反汇编指令：arm-linux-gnueabihf-objdump

INCUDIRS := imx6u \ #头文件路径

bsp/clk \

bsp/led \

bsp/beep \

bsp/delay

SRCDIRS := project \ #源码文件路径

bsp/clk \

bsp/led \

bsp/beep \

bsp/delay

INCLUDE := $(patsubst %, -I %, $(INCUDIRS)) #在头文件路径前加-I，见注①②

SFILES := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/\*.S)) #保存所有的.S文件名，见注③④

CFILES := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/\*.c)) #保存所有的.c文件名，见注③④

SFILENDIR := $(notdir $(SFILES)) #去掉SFILES中.S文件路径

CFILENDIR := $(notdir $(CFILES)) #去掉CFILES中.c文件路径

SOBJS := $(patsubst %, obj/%, $(SFILENDIR:.S=.o)) #将.S改为.o并在前面加上obj/

COBJS := $(patsubst %, obj/%, $(CFILENDIR:.c=.o)) #将.c改为.o并在前面加上obj/

OBJS := $(SOBJS)$(COBJS) #保存所有的.o文件名

VPATH := $(SRCDIRS) #源码文件搜索，见注⑤

.PHONY:clean #定义伪目标clean，见注⑥

$(TARGET).bin : $(OBJS) #目标文件依赖所有的.o文件

$(LD) -Timx6ul.lds -o $(TARGET).elf $^ #将所有的.o文件链接为.elf文件

$(OBJCOPY) -O binary -S $(TARGET).elf $@ #将.elf文件格式转为目标文件格式

$(OBJDUMP) -D -m arm $(TARGET).elf > $(TARGET).dis #将.elf文件格式反汇编为.dis文件

$(SOBJS) : obj/%.o : %.S #将SOBJS中所有的.S文件编译成.o文件

$(CC) -Wall -nostdlib -c -O2 $(INCLUDE) -o $@ $< #并存放到obj目录下，见注⑦

$(COBJS) : obj/%.o : %.c #将COBJS中所有的.c文件编译成.o文件

$(CC) -Wall -nostdlib -c -O2 $(INCLUDE) -o $@ $< #并存放到obj目录下，见注⑦

clean: #伪指令，用于清除中间文件

rm -rf $(TARGET).elf $(TARGET).bin $(TARGET).dis $(OBJS)

**注**：1）Makefile语法要求指定头文件路径时需要加上-I。

2）patsubst为模式字符串替换函数，格式为：$(patsubst <pattern>,<replacement>,<text>)，其功能是：查看text中的单词

是否符合pattern中的模式规则。如果符合，则以replacement中的内容替换。text中单词以空格、Tab、回车或换行分隔，本文件使用的是空格分隔。pattern中可以包含通配符%，若replacement中也使用通配符%，则两个%代表的字符串对应相同。所以$(patsubst %, -I %, $(INCUDIRS))的功能是将INCUDIRS中的字符串以空格拆分，并在前面加上-I。

3）wildcard为关键字，通配符在变量中展开时使用。如objects是所有.o文件名的集合，即objects := $(wildcard \*.o)，

而objects = \*.o时，objects的值就是\*.o，而不是所有.o文件名的集合。所以$(wildcard $(dir)/\*.S)的功能是表示dir目录下所有.S文件名的集合。

4）foreach是用来做循环的函数，格式为：$(foreach <var>,<list>,<text>)，其功能是把list中的单词逐一取出放到var中，

再执行text中的表达式，每次text会返回一个字符串，循环结束后将所有的字符串以空格分隔组成一个字符串作为函

数返回值。所以$(foreach dir, $(SRCDIRS), $(wildcard $(dir)/\*.S))的功能是将SRCDIRS中的源码文件路径逐一取出，

再取出对应路径下所有的.S文件，并以空格分隔组成一个字符串。SRCDIRS中包含路径和文件名，如project/start.S.

5）VPATH是用于文件搜索的变量，在一些大工程中有大量的源文件，我们通常将其分类存储在不同目录中。当没有定

义VPATH变量时，make只会在当前目录下寻找依赖文件和目标文件；当定义了VPATH后，make在当前目录下找不到对应文件时就会到VPATH变量内所指定的目录中去寻找文件。

6）伪目标不代表真正的目标名，在执行make命令时通过指定这个伪目标来执行其所在规则定义的命令，使用伪目标主

要是为了避免Makefile中定义的执行命令的目标和工作目录下的实际文件出现名字冲突，所以.PHONY:clean的功能

是定义一个名为clean的伪目标，使用指令make clean即可执行该目标下定义的命令。

7）静态模式可以更加容易地定义多目标的规则，格式为：<targets>:<target-pattern>:<prereq-pattern>，其中，targets定义

了一系列的目标文件, 是目标的一个集合；target-parrtern指明了targets的模式，也就是目标集模式；prereq-parrterns 是目标的依赖模式，它对target-parrtern形成的模式再进行一次依赖目标的定义。示例如下：

objects = foo.o bar.o

$(objects): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

上例中指明了目标文件从objects中获取，%.o表明要所有以.o结尾的文件作为目标文件，%.c表明所有的目标文件对应的依赖文件为对应的.c文件。$<表示所有的依赖文件集合，$@表示所有的目标文件集合，CFLAGS表示指定头文件（.h文件）的路径，如：CFLAGS=-I/usr/include -I/path/include。所以上例等价于：

foo.o : foo.c

$(CC) -c $(CFLAGS) foo.c -o foo.o

bar.o : bar.c

$(CC) -c $(CFLAGS) bar.c -o bar.o

所以$(SOBJS) : obj/%.o : %.S的功能是将SOBJS中所有的.S文件编译成对应的.o文件，并存放到obj目录下。

**注：-Wall：**打印出编译时所有的错误或者警告信息。

**-nostdlib：**不连接系统标准启动文件和标准库文件，只把指定的文件传递给连接器。生成一个C程序的可执行文件时

编译器通常会在我们的代码上加几个启动文件，如crt1.o、crti.o、crtend.o、crtn.o等，它们是标准库文件。这些代码设置C程序的堆栈等，然后调用main函数，它们依赖于操作系统，在裸板上无法执行。这个选项常用于编译内核、bootloader等程序，它们不需要启动文件、标准库文件。

1. **时钟系统**

I.MX6U的系统主频为528MHz，有些型号可以跑到696MHz，但默认情况下内部BOOT ROM会将I.MX6U的主频设置为396MHz。为了发挥芯片最大性能，需要将主频设置到最高频率。（具体见参考手册345、625页）

①**时钟来源**：I.MX6U-ALPHA开发板中时钟来源有两部分，32.768KHz晶振和24MHz的晶振。

**32.768KHz晶振**：RTC时钟源

**24MHz晶振**：I.MX6U内核和其它外设的时钟源

②**PLL时钟源**

I.MX6U的外设有很多，不同的外设时钟源不同，NXP将这些外设的时钟源分成了7组PLL时钟源，其来源是24MHz晶振，7组PLL时钟源介绍如下。（PLL时钟源生成图见参考手册350页）

1) **PLL1**：ARM\_PLL，倍频可调整，频率范围为650MHz~1.3GHz，供ARM内核使用。

2）**PLL2**： System\_PLL，固定22倍频，即其频率固定为24MHz×22=528MHz。PLL2还分出4路PFD时钟源，

即**PLL2\_PFD0~PLL2\_PFD3**，这五路供I.MX6U 内部系统总线使用，如DDR接口、NAND/NOR接口、内处理逻辑单元等。

3）**PLL3**：USB1\_PLL，固定20倍频，即其频率固定为24MHz×20=480MHz。PLL3还分出4路PFD时钟源，即

**PLL3\_PFD0~PLL3\_PFD3**，这五路主要供USBPHY使用，但也可作为其他外设的根时钟源。

4）**PLL4**：AUDIO\_PLL，倍频可调整，频率范围为：650MHz~1.3GHz。此路时钟源输出时可进行1/2/4/8/16分频，

供音频相关外设使用。

5）**PLL5**：VIDEO\_PLL，倍频可调整，频率范围为：650MHz~1.3GHz。此路时钟源输出时可进行1/2/4/8/16分频，

供显示相关外设使用，如LCD等。

6）**PLL6**：ENET\_PLL，固定20+(5/6)倍频，VCO频率为500MHz，即频率固定为MIN(24×(20+(5/6)),500)=500

MHz。此路用于生成网络所需时钟，如25/50/100/125MHz的网络时钟。

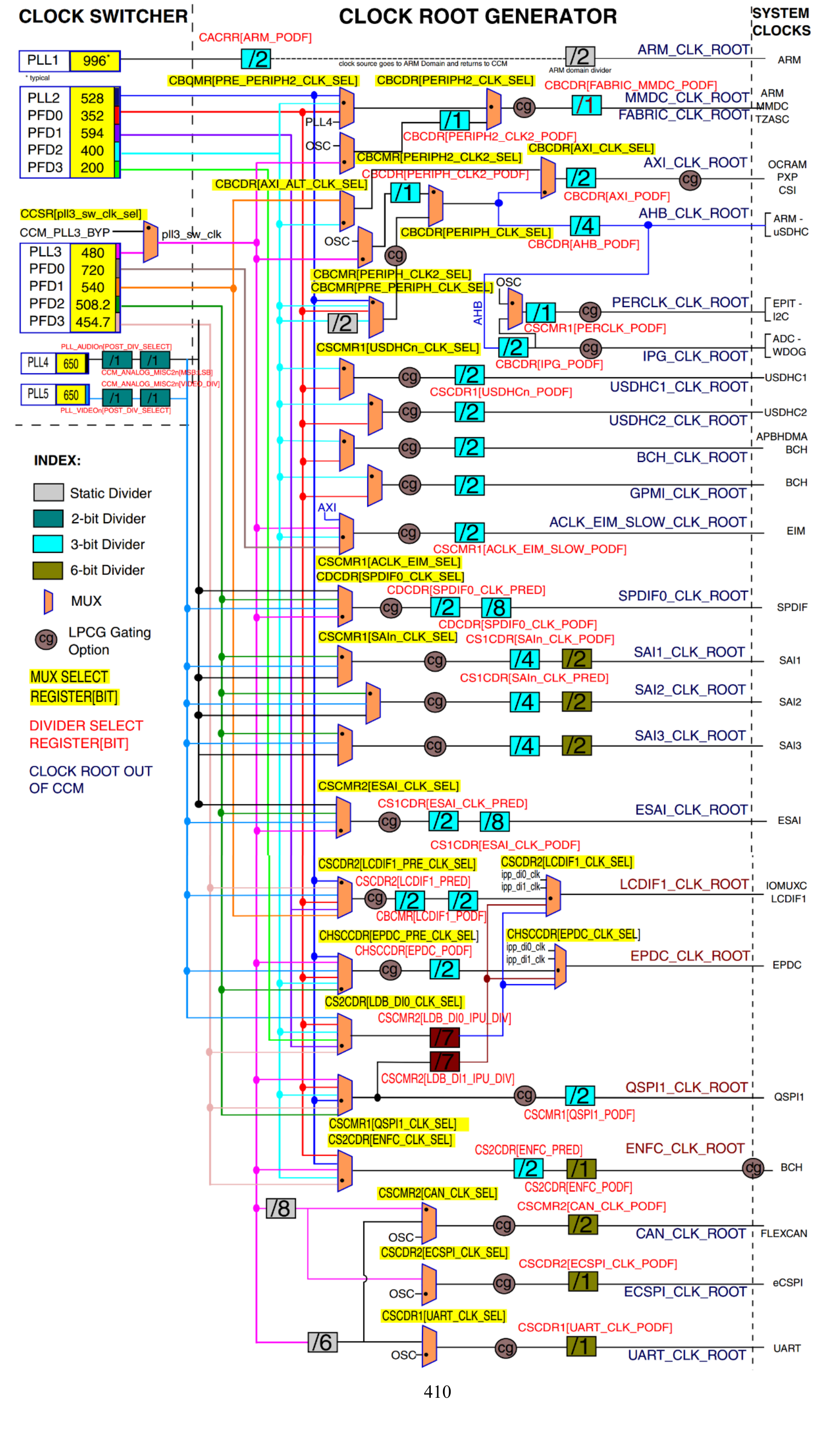
7）**PLL7**：USB2\_PLL，固定20倍频，即其频率固定为24MHz×20=480MHz，专门供USB2PHY使用。

③**时钟树**

I.MX6U的外设时钟源都来自上述的PLL1~PLL7，PLL2\_PFD0~PLL2\_PFD3和PLL3\_PFD0~PLL3\_PFD3，而将上述时钟源分配给外设时就需要参考时钟树，根据时钟树配置相应寄存器即可完成相应外设的时钟配置，时钟树图如下。

以ESAI外设为例，ESAI有四个可选时钟源：PLL4、PLL5、PLL3\_PFD2 和pll3\_sw\_clk，通过配置寄存器CCM->CSCMR2的ESAI\_CLK\_SEL位即可对这4个时钟源进行选择；配置寄存器CCM->CCM\_CS1CDR的ESAI\_CLK\_PRED和ESAI\_CLK\_PODF位即可对分频值进行选择。（具体寄存器配置见参考手册658页）

下面将分别对内核时钟、PFD时钟、AHB、IPG和PERCLK时钟进行配置，配置寄存器均参考时钟树及参考手册，PLL2、PLL3和PLL7固定为528MHz、480MHz和480MHz，而PLL4~PLL6都是针对特殊外设的，用到的时候再设置。其他外设时钟配置方式与下述时钟配置流程相同，查看时钟树和参考手册配置相应寄存器即可。



④**配置内核时钟（ARM）**

内核时钟来源为PLL1，计划将频率设置为528MHz，配置寄存器CCM\_ANALOG\_PLL\_ARMn的DIV\_SELECT位对PLL1频率进行选择（范围为54~108，配置公式：Fout = Fin \* div\_select/2.0），配置寄存器CCM\_CACRR的ARM\_PODF位对PLL1进行分频。在修改PLL1时，需为I.MX6U提供一个临时时钟，配置CCSR的STEP\_SEL位设置备用时钟，配置寄存器CCSR的PLL1\_SW\_CLK\_SEL位将备用时钟作为系统时钟，PLL1配置完成后再将系统时钟切换为PLL1。将内核时钟设置为528MHz具体步骤如下：（时钟切换图见参考手册648页）

1）设置备用时钟由24MHz晶振提供。将CCM\_CCSR的STEP\_SEL位配置为0。

2）将备用时钟作为内核时钟。将CCSR的PLL1\_SW\_CLK\_SEL位配置为1。

3）使能PLL1时钟。将CCM\_ANALOG\_PLL\_ARMn的ENABLE位配置为1。

4）设置PLL1为1056MHz。将CCM\_ANALOG\_PLL\_ARMn的DIV\_SELECT位配置为1056×2÷24=88。

5）对PLL1进行2分频。将CCM\_CACRR的ARM\_PODF位配置为001。

6）将内核时钟更改为PLL1。将CCSR的PLL1\_SW\_CLK\_SEL位配置为0。

⑤**配置PFD时钟**

下表推荐频率与时钟树中稍有不同，为正点原子测试后的推荐频率，根据推荐频率配置相应寄存器即可完成时钟频率配置，时钟推荐频率及寄存器介绍如下。

|  |  |  |  |
| --- | --- | --- | --- |
| **时钟** | **频率** | **时钟** | **频率** |
| **PLL2\_PFD0** | 352MHz | **PLL3\_PFD0** | 720MHz |
| **PLL2\_PFD1** | 594MHz | **PLL3\_PFD1** | 540MHz |
| **PLL2\_PFD2** | 396MHz | **PLL3\_PFD2** | 508.2MHz |
| **PLL2\_PFD3** | 297MHz | **PLL3\_PFD3** | 454.7MHz |

1）**设置PLL2\_PFD0~PLL2\_PFD3频率。**（见参考手册736页）

配置寄存器CCM\_ANALOG\_PFD\_528n的PFD0\_FRAC、PFD1\_FRAC、PFD2\_FRAC、PFD3\_FRAC位对

PLL2\_PFD0~PLL2\_PFD3进行频率设置；PFDx\_FRAC范围为12-35，配置公式：Fout =480\*18/PFDx\_FRAC。

经计算，将PFD0\_FRAC、PFD1\_FRAC、PFD2\_FRAC、PFD3\_FRAC分别配置为27、16、24、32。

2）**设置PLL3\_PFD0~PLL2\_PFD3频率。**（见参考手册734页）

配置寄存器CCM\_ANALOG\_PFD\_480n的PFD0\_FRAC、PFD1\_FRAC、PFD2\_FRAC、PFD3\_FRAC位对

PLL3\_PFD0~PLL3\_PFD3进行频率设置；PFDx\_FRAC范围为12-35，配置公式：Fout =528\*18/PFDx\_FRAC。

经计算，将PFD0\_FRAC、PFD1\_FRAC、PFD2\_FRAC、PFD3\_FRAC分别配置为12、16、17、19。

⑥**配置常用外设时钟**（其他时钟范围见参考手册543页）

1）**AHB\_CLK\_ROOT**（频率范围：6~132MHz，）

将寄存器CBCMR的PRE\_PERIPH\_CLK\_SEL位配置为01，将寄存器CBCDR的PERIPH\_CLK\_SEL位配置为0，即可将时钟选择为PLL2\_PFD2（396MHz）。将寄存器CBCDR的AHB\_PODF位配置为010，即可进行3分频，得到132MHz作为AHB\_CLK\_ROOT时钟。

**注**：①CBCDR的对应位配置完成后要等待寄存器CDHIPR的PERIPH\_CLK\_SEL\_BUSY和AHB\_PODF\_BUSY位

为0，即等待握手完成。

②配置寄存器CBCDR的AHB\_PODF位之前，要先禁止AHB\_CLK\_ROOT时钟输出，否则会出现不可预知的错误，如导致没有时钟输出。内部BOOT ROM已将AHB\_PODF默认设置为了3分频，因此将时钟选择为PLL2\_PFD2后，再等待握手完成，无需手动分频，即可将AHB\_CLK\_ROOT配置为132MHz**。**

2）**IPG\_CLK\_ROOT**（频率范围为3~66MHz）

AHB\_CLK\_ROOT配置完成后，将寄存器CBCDR的IPG\_PODF位配置为01，即可进行2分频，得到66MHz作为IPG\_CLK\_ROOT时钟。

3）**PERCLK\_CLK\_ROOT**（频率范围为3~66MHz）

IPG\_CLK\_ROOT配置完成后，将寄存器CSCMR1的PERCLK\_CLK\_SEL位配置为0，将寄存器CSCMR1的PERCLK\_PODF配置为0，即可得到66MHz作为PERCLK\_CLK\_ROOT时钟**。**

1. **中断**

①**STM32中断**

STM32中断系统包含3部分：中断向量表、NVIC（中断控制器）和中断服务函数。介绍如下：

1）**中断向量**

中断向量是中断服务函数的入口地址或存放中断服务函数的首地址，当某个中断触发后，程序就会自动跳转到对应中断向量所定义的地址处执行中断服务函数。中断向量表在整个程序的最前面，由半导体厂商规定。

2）**中断向量表偏移**

一般ARM处理器是从地址0X00000000开始执行指令的，则中断向量表就从该地址开始存放。但也可通过中断向量表偏移将中断向量表存放到任意地址处，在启动文件中的函数SystemInit中通过向SCB\_VTOR寄存器写入新的中断向量表首地址，即可完成中断向量表偏移。

3）**NVIC**

NVIC为中断管理机构，主要功能为：使能/关闭中断、设置中断优先级等。

②**I.MX6U中断**

I.MX6U是Cortex-A7内核，Cortex-A7中断系统也包含3部分：中断向量表、GIC（中断控制器）和中断服务函数。I.MX6U的中断向量表也存放在代码最前面，但需要用户自己定义；正点原子例程中代码起始地址为0X87800000，因此也需进行中断向量表偏移，将中断向量表存放到该地址处；与NVIC类似，GIC为Cortex-A的中断管理机构，主要功能为：使能/关闭中断、设置中断优先级等。下面具体介绍。

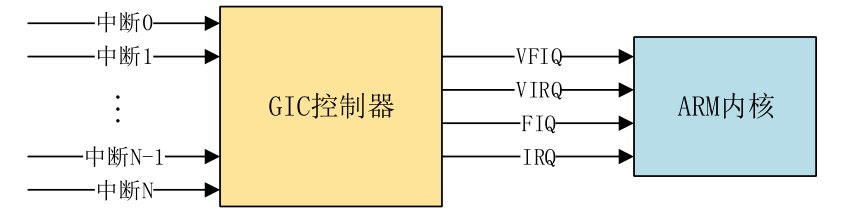
1）**中断向量表**

I.MX6U的中断向量表包含8个异常中断，其中重点关注IRQ中断，介绍如下：

|  |  |  |
| --- | --- | --- |
| **向量地址** | **中断类型** | **中断模式** |
| **0X00** | 复位中断(Rest) | 特权模式(SVC) |
| **0X04** | 未定义指令中断(Undefined Instruction) | 未定义指令中止模式(Undef) |
| **0X08** | 软中断(Software Interrupt,SWI) | 特权模式(SVC) |
| **0X0C** | 指令预取中止中断(Prefetch Abort) | 中止模式 |
| **0X10** | 数据访问中止中断(Data Abort) | 中止模式 |
| **0X14** | 未使用(Not Used) | 未使用 |
| **0X18** | IRQ中断(IRQ Interrupt) | 外部中断模式(IRQ) |
| **0X1C** | FIQ中断(FIQ Interrupt) | 快速中断模式(FIQ) |

2）**GIC**（具体介绍见《ARM Generic Interrupt Controller(ARM GIC控制器)V2.0.pdf》）

I.MX6U使用的是GIC的V2版本，GIC V2最多支持8个核。GIC接收众多的外部中断，并通过4个信号传送给ARM内核，其中主要关注IRQ信号，示意图如下：



**VFIQ**：虚拟快速FIQ **VIRQ**：虚拟外部IRQ **FIQ**：快速中断IRQ **IRQ**：外部中断IRQ

我们只关注外部中断IRQ，任何一个外部中断发生都会触发IRQ中断。GIC将众多的中断源分为三类：SPI、PPI和SGI。为了区分不同的中断源，GIC为每个中断源分配唯一的ID，即中断ID。每个CPU最多支持1020个中断ID，即中断号为：ID0~ID1019。I.MX6U共使用了160个中断ID：属于SGI的ID0~ID15、属于PPI的ID16~ID31和属于SPI的ID32~ID159。主要关注SPI的ID32~ID159，介绍如下：（具体中断编号信息见参考手册183页）

a) **SPI**(Shared Peripheral Interrupt)：共享中断，所有Core共享的中断，外部中断都属于SPI中断。（**ID32~ID1019**）

b) **PPI**(Private Peripheral Interrupt)：私有中断，GIC支持多核，每个核独有的中断叫做私有中断。（**ID16~ID31**）

c) **SGI**(Software-generated Interrupt)：软件中断，由软件触发引起的中断，通过向寄存器GICD\_SGIR写入数据来

触发，系统会使用SGI中断来完成多核之间的通信。（**ID0~ID15**）

③**编写中断向量表**（在start.S文件中编写）

a) 目前start.S文件中代码如下，下面的start.S文件完成了C语言运行环境的初始化，后续对start.S进行改写：

.global \_start /\* 定义全局标号\*/

.global \_bss\_start

.global \_bss\_end

\_bss\_start: //注意，此段代码放在这可能会导致后面设置的向量表地址偏移

.word \_\_bss\_start //发生错误，一定要保证中断向量表在代码的最前面，即在本教

\_bss\_end: //程中中断向量表的起始地址为0X87800000，所以可把这段代码

.word \_\_bss\_end //放到清除bss段代码的前面，可在.dis反汇编文件中查看地址信息

\_start: /\*程序从此函数开始执行，此函数主要功能是设置C运行环境 \*/

mrs r0, cpsr /\*这4行指令配置芯片进入SVC模式 \*/

bic r0, r0, #0x1f /\*将r0寄存器中的低5位清零，也就是cpsr的M0~M4 \*/

orr r0, r0, #0x13 /\* r0或上0x13,表示使用SVC模式 \*/

msr cpsr, r0 /\*将r0 的数据写入到cpsr\_c中 \*/

/\* 可将保存bss段起始位置的代码段放到此处 \*/

ldr r0, \_bss\_start /\* 下面7行指令用于清除bss段 \*/

ldr r1, \_bss\_end

mov r2, #0 /\* 将r2赋值为0 \*/

bss\_loop:

stmia r0!, {r2} /\* 将r2中的值赋值给r0指定的地址单元，每次赋值完成后r0+1\*/

cmp r0, r1 /\* 比较R0和R1里面的值 \*/

ble bss\_loop /\* 若R0地址小于等于R1，继续清除bss段 \*/

ldr sp,=0X80200000 /\* 设置栈指针，2M的堆栈空间 \*/

b main /\* 跳转到main函数 \*/

b) 下面就在\_start函数中编写中断向量表，I.MX6U的中断向量表包含8个异常中断，编写如下：

ldr pc, =Reset\_Handler /\* 复位中断 \*/

ldr pc, =Undefined\_Handler /\* 未定义中断 \*/

ldr pc, =SVC\_Handler /\* SVC(Supervisor)中断 \*/

ldr pc, =PrefAbort\_Handler /\* 预取终止中断 \*/

ldr pc, =DataAbort\_Handler /\* 数据终止中断 \*/

ldr pc, =NotUsed\_Handler /\* 未使用中断 \*/

ldr pc, =IRQ\_Handler /\* IRQ中断 \*/

ldr pc, =FIQ\_Handler /\* FIQ(快速中断)未定义中断 \*/

④**编写中断服务函数**（在start.S文件中编写）

下面编写中断服务函数，本教程中除了复位中断和IRQ中断外，其他6个中断没用到，所以都设置为了死循环，以SVC中断为例，设置死循环方式如下，其余5个中断服务函数按同样形式设置为死循环。下面主要关注复位中断服务函数和IRQ中断服务函数的编写：

SVC\_Handler: /\* SVC中断 \*/

ldr r0, =SVC\_Handler

bx r0

**1) 复位中断服务函数**

内容为：关闭I/D Cache和MMU、设置9种[运行模式](#运行模式)的sp指针、清除bss段、跳到main函数：

Reset\_Handler:

cpsid i /\* 关闭IRQ中断 \*/

/\* 关闭I,DCache和MMU采取读-改-写的方式，见注③ \*/

mrc p15, 0, r0, c1, c0, 0 /\* 读取CP15的C1寄存器中的值保存到寄存器R0中 \*/

bic r0, r0, #(0x1 << 12) /\* 清除C1寄存器的bit12位(I位)，关闭I Cache \*/

bic r0, r0, #(0x1 << 2) /\* 清除C1寄存器的bit2(C位)，关闭D Cache \*/

bic r0, r0, #(0x1 << 1) /\* 清除C1寄存器的bit1(A位)，关闭对齐 \*/

bic r0, r0, #(0x1 << 11) /\* 清除C1寄存器的bit11(Z位)，关闭分支预测 \*/

bic r0, r0, #0x1 /\* 清除C1寄存器的bit0(M位)，关闭MMU \*/

mcr p15, 0, r0, c1, c0, 0 /\* 将寄存器r0中的值写入到CP15的C1寄存器中 \*/

#if 0 /\* 汇编版本设置中断向量表偏移，见注④ \*/

ldr r0, =0X87800000 /\* 将偏移后的地址写入到寄存器r0中 \*/

dsb /\* 数据同步隔离 \*/

isb /\* 指令同步隔离 \*/

mcr p15, 0, r0, c12, c0, 0 /\* 将r0中的新地址写入寄存器c12中 \*/

dsb /\* 数据同步隔离 \*/

isb /\* 指令同步隔离 \*/

#endif

/\* 可将清除bss段的代码移动至此处 \*/

/\*设置各个模式下的栈指针，下面只设置了3种模式，其余模式设置方法相同 \*/

/\* IMX6UL的堆栈是向下增长的！堆栈指针地址一定要4字节地址对齐 \*/

/\* DDR范围:0X80000000~0X9FFFFFFF \*/

mrs r0, cpsr /\* 进入IRQ模式 \*/

bic r0, r0, #0x1f /\* 将r0寄存器中的低5位清零，也就是cpsr的M0~M4 \*/

orr r0, r0, #0x12 /\* r0或上0x12,表示使用IRQ模式 \*/

msr cpsr, r0 /\* 将r0的数据写入到cpsr\_c中 \*/

ldr sp, =0x80600000 /\* 设置IRQ模式下的栈首地址为0X80600000,大小为2MB \*/

mrs r0, cpsr /\* 进入SYS模式 \*/

bic r0, r0, #0x1f /\* 将r0寄存器中的低5位清零，也就是cpsr的M0~M4 \*/

orr r0, r0, #0x1f /\* r0或上0x1f,表示使用SYS模式 \*/

msr cpsr, r0 /\* 将r0的数据写入到cpsr\_c中 \*/

ldr sp, =0x80400000 /\* 设置SYS模式下的栈首地址为0X80400000,大小为2MB \*/

mrs r0, cpsr /\* 进入SVC模式 \*/

bic r0, r0, #0x1f /\* 将r0寄存器中的低5位清零，也就是cpsr的M0~M4 \*/

orr r0, r0, #0x13 /\* r0或上0x13,表示使用SVC模式 \*/

msr cpsr, r0 /\* 将r0的数据写入到cpsr\_c中 \*/

ldr sp, =0X80200000 /\* 设置SVC模式下的栈首地址为0X80200000,大小为2MB \*/

cpsie i /\* 打开IRQ中断 \*/

**注**：

1）**I/Dcache**：Icache（指令高速缓存）和Dcache（数据高速缓存）是一种内存，我们的代码分为两部分：指令和数据，

Icache用于缓存指令，Dcache用于缓存数据。CPU和主存之间存在多级高速缓存，一般分为3级，分别是L1、L2和L3。每个CPU都有两个L1 cache，即Icache、Dcache；而L2 cache和L3 cache可同时存储指令和数据。区分数据和指令可使CPU执行程序时同时获取指令和数据，进而提升性能。此外，指令一般不会修改，所以ICache在硬件设计上可以是只读的，这在一定程度上降低了硬件设计的成本。

2）**MMU**：内存管理单元，是一种负责处理CPU的内存访问请求的计算机硬件。功能包括：虚拟地址到物理地址的转换、

内存保护、CPU高速缓存的控制、在较简单的计算机体系结构中负责总线的仲裁以及存储体切换。运行linux，芯片需要MMU，STM32没有MMU，所以不能运行Linux。 ARM出品的CPU，MMU作为一个协处理器存在. 根据不同的系列有不同搭配。需要查询DATASHEET才可知道是否有MMU。如果有的话, 一定是编号为15的协处理器，可以提供32BIT共4G的地址空间。

3）关闭I,DCache和MMU需要配置CP15协处理器的SCTLR（系统控制）寄存器，此寄存器中，bit0用于打开/关闭MMU，

bit1用于检查内存是否对齐，bit2用于打开/关闭D cache，bit11用于打开/关闭分支预测，bit12用于打开/关闭I cache，只配置以上bit位即可，其余不变。使用如下指令对SCTLR进行读写：

（具体寄存器信息见《Cortex-A7 Technical ReferenceManua.pdf》第4-51页）

MRC p15, 0, <Rt>, c1, c0, 0 //读取SCTLR寄存器数据保存到Rt中。

MCR p15, 0, <Rt>, c1, c0, 0 //将Rt中的数据写到 SCTLR(c1)寄存器中。

BIC Rd, Rn, #immed //位清除，效果为Rd = Rn & (~#immed)

4）设置中断向量偏移需要配置CP15协处理器的VBAR（向量表基地址）寄存器，使用如下指令对VBAR进行读写：

（具体寄存器信息见《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》第1479页）

ldr r0, =0X87800000 //r0=0X87800000

MCR p15, 0, r0, c12, c0, 0 //将r0里面的数据写入到VBAR中，即c12=0X87800000

**DSB**：数据同步隔离，仅当它前面所有的内存访问指令都执行完毕后，才会执行在它后面的指令。

**ISB**：指令同步隔离，它会清洗流水线，保证它前面所有的指令都执行完毕后，才执行它后面的指令。

**2）IRQ中断服务函数**

IRQ\_Handler:

push {lr} /\* 保存lr地址 \*/

push {r0-r3, r12} /\* 保存r0-r3，r12寄存器 \*/

mrs r0, spsr /\* 读取spsr寄存器 \*/

push {r0} /\* 保存spsr寄存器 \*/

mrc p15, 4, r1, c15, c0, 0 /\* 从CP15的C0寄存器内的值到R1寄存器中，见注① \*/

add r1, r1, #0X2000 /\* GIC基地址加0X2000，也就是GIC的CPU接口端基地址 \*/

ldr r0, [r1, #0XC] /\* 获取中断ID号，调用对应的中断服务函数见注② \*/

push {r0, r1} /\* 保存r0和r1。r0：中断号ID，r1：GIC的CPU接口端基地址 \*/

cps #0x13 /\* 进入SVC模式，允许其他中断再次进去，见注③ \*/

push {lr} /\* 保存SVC模式的lr寄存器 \*/

ldr r2, =system\_irqhandler /\* 加载C语言中断处理函数到r2寄存器中 \*/

blx r2 /\* 运行中断处理函数，带有一个参数，即R0寄存器中的中断ID号 \*/

pop {lr} /\* 执行完C语言中断服务函数，lr出栈 \*/

cps #0x12 /\* 进入IRQ模式 \*/

pop {r0, r1}

str r0, [r1, #0X10] /\* 将中断号ID写入到EOIR，见注④ \*/

pop {r0}

msr spsr\_cxsf, r0 /\* 恢复spsr \*/

pop {r0-r3, r12} /\* r0-r3，r12出栈 \*/

pop {lr} /\* lr出栈 \*/

subs pc, lr, #4 /\* 将lr-4赋给pc，见注⑤ \*/

**注**：

1）mrc p15, 4, r1, c15, c0, 0，此条指令是读取CP15协处理器的CBAR（配置基地址）寄存器，CBAR寄存器保存了GIC控

制器的寄存器组首地址。GIC架构分为了两个逻辑块：Distributor和CPU Interface，也就是分发器端和CPU接口端。GIC寄存器组偏移地址为0x1000 - 0x1FFF的GIC块为分发器端，偏移地址为0x2000 - 0x3FFF的GIC块为CPU 接口端。CPU接口端基地址加0X0C就是GICC\_IAR寄存器，读取GICC\_IAR寄存器的bit0~9即可得到中断ID，进而可根据中断ID执行对应的中断处理函数。

（具体寄存器信息见《Cortex-A7 Technical ReferenceManua.pdf》第4-83页）

（GIC内存映射信息见《Cortex-A7 Technical ReferenceManua.pdf》第8-3页）

2）ldr r0, [r1, #0XC] //将存储器地址为r1+0XC（即GICC\_IAR寄存器）的数据读入寄存器r0

3）当前处于IRQ模式，后面要执行IRQ中断服务函数system\_irqhandler，所以要进入SVC模式，允许其他中断再次进去。

4）规定IRQ中断服务函数system\_irqhandler执行完具体的中断处理函数后，需要将对应的中断ID值写入到GICC\_EOIR中，

CPU接口端基地址加0X10就是GICC\_EOIR寄存器。

5）subs pc, lr, #4，此条指令将lr链接地址减4赋值给pc，表示中断处理完成以后重新返回到曾经被中断打断的地方继续运

行。ARM的指令是三级流水线：取指、译指、执行，pc指向的是正在取值的地址，这就是很多书上说的 pc=当前执行指令地址+8，代码示例如下。

0X2000 MOV R1, R0 //执行

0X2004 MOV R2, R3 //译指

0X2008 MOV R4, R5 //取值 <- PC

如上所示，当程序执行0X200处的指令MOV R1, R0时发生中断，此时PC保存的是取值对应的地址，即lr保存的是0X2008。所以当中断结束后要对lr-4，才能继续运行译指对应的指令。

⑤**中断驱动初始化**

编写驱动前，需要移植SDK包中的core\_ca7.h文件，该文件与中断相关。由于移植比较复杂，直接使用正点原子提供的移植后的文件。移植后的core\_ca7.h文件中主要关注10个GIC相关的API函数，介绍如下。

|  |  |
| --- | --- |
| **函数** | **描述** |
| GIC\_Init | 初始化 GIC |
| GIC\_EnableIRQ | 使能指定的外设中断 |
| GIC\_DisableIRQ | 关闭指定的外设中断 |
| GIC\_AcknowledgeIRQ | 返回中断号 |
| GIC\_DeactivateIRQ | 无效化指定中断 |
| GIC\_GetRunningPriority | 获取当前正在运行的中断优先级 |
| GIC\_SetPriorityGrouping | 设置抢占优先级位数 |
| GIC\_GetPriorityGrouping | 获取抢占优先级位数 |
| GIC\_SetPriority | 设置指定中断的优先级 |
| GIC\_GetPriority | 获取指定中断的优先级 |

下面开始编写中断驱动相关函数：

1）中断初始化

void int\_init(void)

{

GIC\_Init(); //GIC初始化

system\_irqtable\_init(); //初始化中断表

\_\_set\_VBAR(0x87800000); //若start.S文件中未设置中断向量偏移，可使用core\_ca7.h中的此函数实现

}

2）初始化中断向量表

前面在start.S中的\_start函数中编写了中断向量表，下面将具体的中断服务函数与中断向量表对应起来。I.MX6U有160个中断源，所以需要160个中断处理函数。下面将这些中断处理函数放到一个数组里，标号即对应的中断ID。

static sys\_irq\_handle\_t irqTable[NUMBER\_OF\_INT\_VECTORS]; //中断函数数组表

sys\_irq\_handle\_t为中断处理函数结构体，结构体中有两个结构体成员：中断处理函数、中断处理函数的参数。所以这是一个名为irqTable的结构体数组。NUMBER\_OF\_INT\_VECTORS表示中断服务函数的个数，因为有160个中断处理函数，所以NUMBER\_OF\_INT\_VECTORS = 160，是在文件MCIMX6Y2.h中定义的。sys\_irq\_handle\_t定义如下。

typedef struct \_sys\_irq\_handle

{

    system\_irq\_handler\_t irqHandler; /\* 中断服务函数 \*/

    void \*userParam;                 /\* 中断服务函数参数 \*/

} sys\_irq\_handle\_t;

system\_irqhandler就是中断ID对应的中断服务函数，下面介绍中断服务函数的形式，这是一个函数指针。

typedef void (\*system\_irq\_handler\_t) (unsigned int giccIar,void \*param); /\* 中断处理函数形式 \*/

如上所示，中断处理函数有两个参数：giccIar和param，giccIar为中断ID号，用于检查中断号是否符合要求，param就是函数所需要的参数。准备好中断处理函数后，下面就开始初始化中断向量表。

void system\_irqtable\_init(void)

{

    unsigned int i = 0;

    irqNesting = 0; /\* 用于中断嵌套计数，后面执行中断服务函数时会用到 \*/

    for(i = 0; i < NUMBER\_OF\_INT\_VECTORS; i++)

    { /\* 先将所有的中断服务函数设置为默认值 \*/

        system\_register\_irqhandler((IRQn\_Type)i,default\_irqhandler, NULL);

    }

}

system\_irqtable\_init函数将所有的中断函数都初始化为了默认值，其中system\_register\_irqhandler为中断服务函数注册函数，其功能是给指定的中断服务函数分配中断ID；IRQn\_Type是一个枚举类型，此枚举类型枚举出了I.MX6U的所有中断，共160个，该枚举类型也是在MCIMX6Y2.h中定义的；C语言中可以将函数作为函数参数，用到的方法是函数指针，函数名就是函数的首地址，default\_irqhandler是一个默认中断服务函数，这个函数中只有一个死循环，没定义任何内容，只是用来给160个中断服务函数进行初始化的。所以system\_irqtable\_init将160个中断服务函数功能都设置为了死循环，函数参数指针都指向了NULL，函数具体介绍如下。

/\* 中断服务函数注册函数 \*/

void system\_register\_irqhandler(IRQn\_Type irq, system\_irq\_handler\_t handler, void \*userParam)

{

    irqTable[irq].irqHandler = handler;

    irqTable[irq].userParam = userParam;

}

/\* 默认中断服务函数 \*/

void default\_irqhandler(unsigned int giccIar, void \*userParam)

{

    while(1)

    {

    }

}

至此，中断向量表初始化就完成了，当发生中断时，程序会到中断向量表中寻找中断服务函数。I.MX6U发生任何外部中断后都会跳转到IRQ中断服务函数，随后IRQ中断服务函数调用C 函数 system\_irqhandler 根据中断ID来执行具体的中断服务函数。

void system\_irqhandler(unsigned int giccIar)

{

 uint32\_t intNum = giccIar & 0x3FFUL; /\*UL表示0X3FF为无符号长整型，即unsigned long\*/

   if ((intNum == 1023) || (intNum >= NUMBER\_OF\_INT\_VECTORS))  /\*检查中断号是否符合要求\*/

        return;

   irqNesting++;    /\* 中断嵌套计数器加一\*/

 irqTable[intNum].irqHandler(intNum, irqTable[intNum].userParam); /\*根据中断ID调用中断服务函数\*/

irqNesting--;   /\* 中断执行完成，中断嵌套寄存器减一 \*/

}

以上就是中断驱动初始化的主要内容，后续就可根据用户自己的要求编写相应的中断处理函数。

⑥**GPIO中断处理函数**（以按键中断为例，对应IO为GPIO1\_IO18）

1）**GPIO配置**

I.MX6U有5组GPIO：GPIO1、GPIO2、GPIO3、GPIO4和GPIO5，其中，GPIO1有32个IO，GPIO2有22个IO,

GPIO3有29个IO，GPIO4有29个IO，GPIO5有12个IO。一共有124个GPIO，每组GPIO 8个寄存器。

配置GPIO中断需要使用3个寄存器：GPIO\_ICR1、GPIO\_ICR2、GPIO\_IMR、GPIO\_ISR。

**GPIO\_ICR1、GPIO\_ICR2**：触发模式寄存器。ICR1配置IO0~15，ICR2配置IO16~31，每两个bit位控制1个IO，配置信息为：**00：**低电平触发、**01：**高电平触发、**10：**上升沿触发**、11：**下降沿触发。此外，还可配置GPIO\_EDGE\_SEL寄存器选择触发模式，1个bit位控制1个IO，当配置为1时，表示上升沿和下降沿都触发中断。注意，GPIO\_EDGE\_SEL寄存器被用于选择触发模式时，GPIO\_ICR1、GPIO\_ICR2寄存器失效。

**GPIO\_IMR**：中断使能寄存器，为1时使能中断。

**GPIO\_ISR**：中断状态寄存器，当某个GPIO发生中断时，对应位就会被置1，当处理完中断后，必须清除标志位。

（注意，向对应位写1进行标志位清除，不是写0）

2）**GIC配置**（具体中断编号信息见参考手册183页）

GIC配置内容为：使能中断ID号、设置中断优先级、注册中断处理函数。GPIO1\_IO18对应的ID为67+32=99。67为外部中断表中的编号，32为PPI和SGI的32个中断。

以上就是中断部分笔记全部的内容，具体GPIO按键中断代码可查看正点原子对应例程。

1. **Ubuntu网络IP改为静态**

①进入网络设置

②将IPV4方式从自动（DHCP）改为手动

③新增地址：192.168.1.106 子网掩码：255.255.255.0 网关：192.168.1.1

④DNS填入：192.168.1.1

⑤终端输入：sudo ifconfig ens33 down (关闭网络)

⑥终端输入：sudo igconfig ens33 up (开启网络)

⑦更改完成

1. **EPIT定时器**

I.MX6U有两个EPIT定时器，主要用于周期性中断定时。EPIT是一个32位的向下计数器，给它一个初值，它就会从

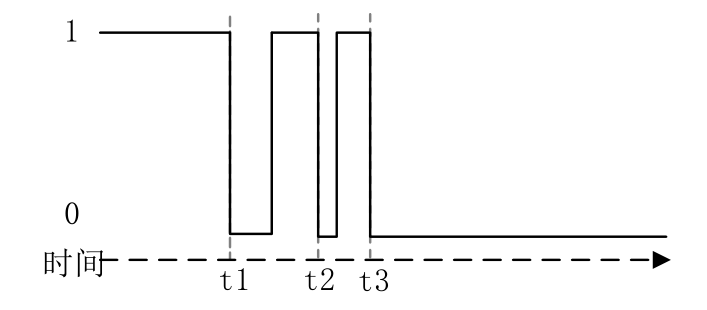
这个给定的初值开始递减，直到减为0，计数寄存器里面保存的就是当前的计数值，当计数值和比较值相等的时候产生中断。EPIT定时器有两种工作模式：set-and-forget模式和free-running模式，介绍如下。（具体寄存器信息查看参考手册1174页）

1）**set-and-forget模式**：计数寄存器中的值减到0后，只能从加载寄存器中取值继续计数，不能向计数寄存器中写值。

2）**free-running模式**：计数寄存器中的值减到0后，重新从0XFFFFFFFF开始计数，不从加载寄存器中取值。

**定时时间公式**：Tout = ((frac +1 )\* value) / Tclk （frac：分频寄存器值 value：计数器值 Tclk：时钟频率Hz）

**重要寄存器**：控制寄存器CR、状态寄存器SR、重装载寄存器LR、比较寄存器CMPR、计数寄存器CNR（只读）

**EPIT定时器结构图 按键消抖**

定时器可用于实现按键消抖，如图所示，t1~t3这段时间就是按键抖动，消抖原理为：t1、t2、t3这三个时刻都会触发按键中断，每次进入中断处理函数都会重新开启定时器中断，即t1、t2、t3这三个时刻会开启定时器，但由于t1~t2 和 t2~t3 这两个时间段小于设置的定时器中断周期，所以只有t3时刻能实现完整的定时周期，触发定时器中断。在定时器中断处理函数中就可以实现想要的按键处理了。

1. **GPT定时器**

I.MX6U有两个GPT定时器，GPT是一个32位的向上计数器，有2个输入捕获通道，3个输出比较通道，因此可以生

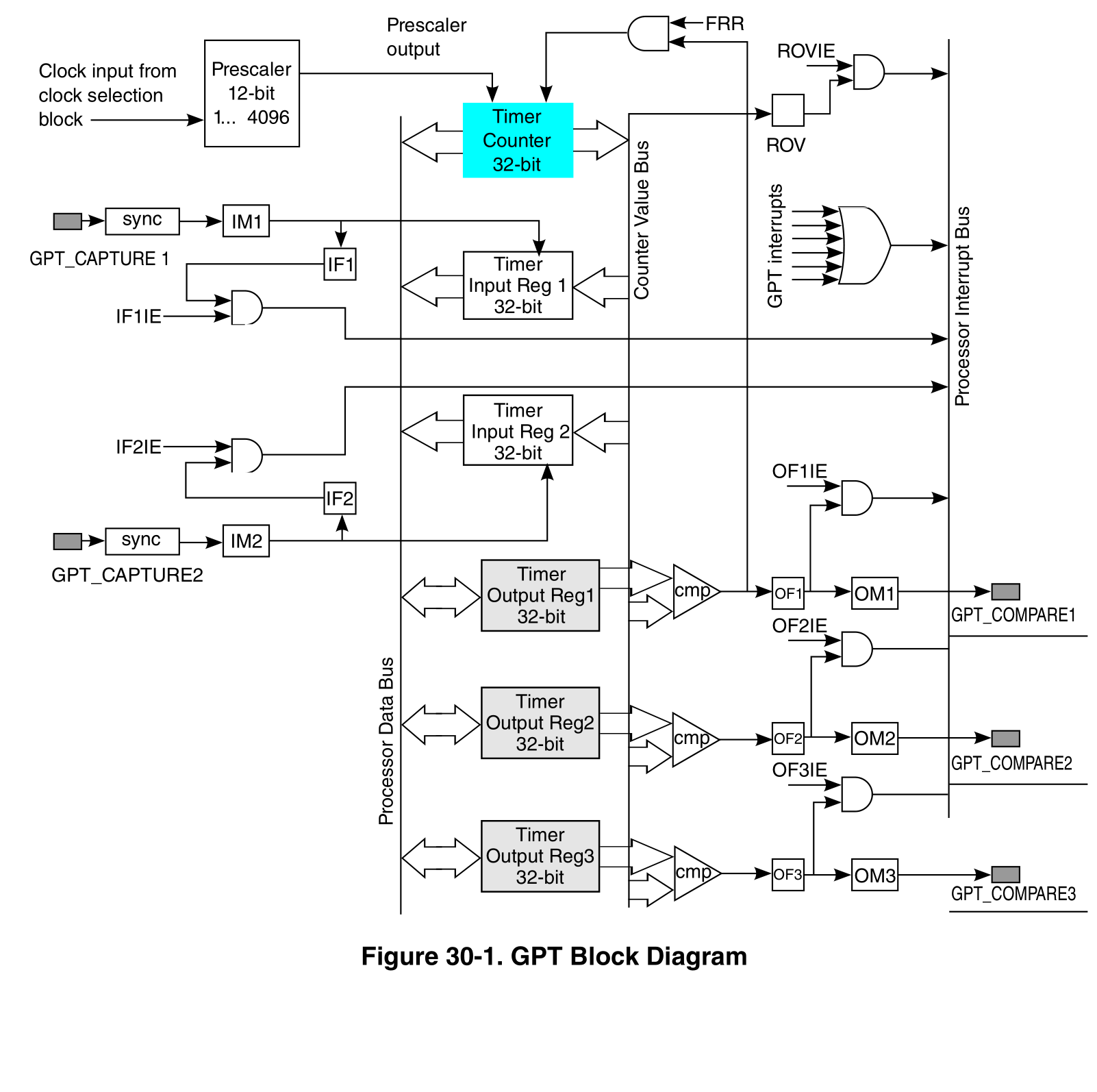
成捕获中断、比较中断和溢出中断。GPT定时器有两种运行模式：restart模式和free-run模式，介绍如下。（具体寄存器信息查看参考手册1434页）

**Restart模式**：计数值和比较值相等时计数值清零，重新从0X00000000开始向上计数，只有比较通道1才有此模式。

向比较通道1的比较寄存器写入任何数据都会复位GPT计数器。

**Free-run模式**：计数值和比较值相等时不会复位计数器，而是继续计数，直到计数值为0XFFFFFFFF，然后重新回滚

到0X00000000重新计数，此模式适用于所有三个比较通道。

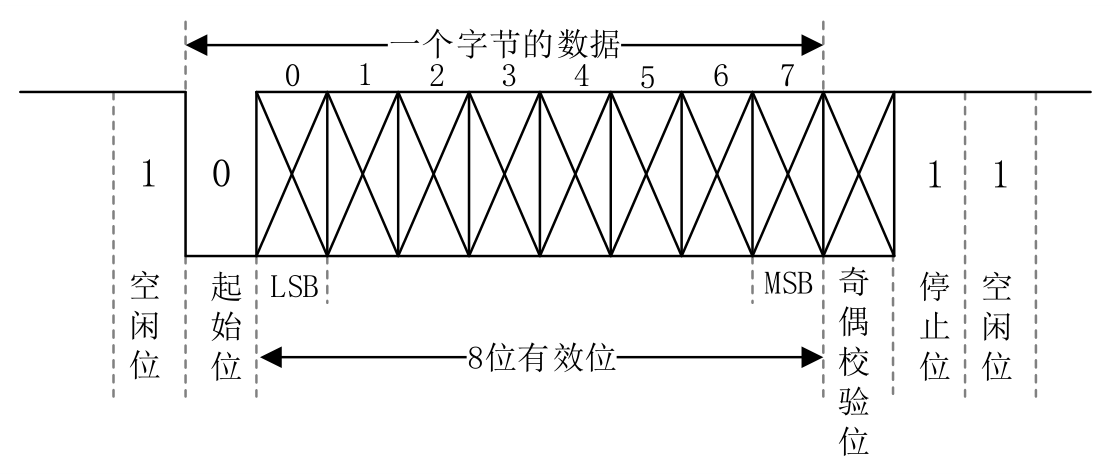


**GPT定时器结构图**

1. **UART串口**

I.MX6U有 8 个 UART，UART通信线路简单，是一种很常用的工业接口。UART与外界相连需要三根线：TXD（发送）、

RXD（接收）、GND（地），通信格式和相关寄存器介绍如下。（具体寄存器信息见参考手册3608页）



**UART通信格式**

**①初始化流程**

1）选择时钟源，设置UART时钟源为PLL3（480MHz），经过6分频得到80MHz。UFCR寄存器的bit7~bit9用于设置

时钟分频值，由[时钟树](#时钟树图)可知，UART的时钟源可选24MHz晶振和PLL3，通常选PLL3（480MHz）经过6分频，即80MHz，配置CSCDR1寄存器对UART的时钟源进行配置（见参考手册674页）。

2）初始化IO，将IO复用为UART，并设置波特率、奇偶校验、停止位、数据位等。UCR2的bit0为软件复位，bit1为

接收使能，bit2为发送使能，bit5设置数据位大小，一般设置为8位数据位，bit6设置停止位，通常设置为1位停止位，bit7为奇偶校验位，bit8设置使能奇偶校验位，一般关闭奇偶校验。bit14设置硬件流控，选择关闭RTS硬件流控。UCR3的bit2必须设置为1。UFCR、UBIR和UBMR寄存器配置串口波特率，波特率计算公式如下：



将波特率设置为115200时，各寄存器配置为：UFCR的RFDIV位配置为5，即1分频；UBIR=71，UBMR=3124。

3）使能UART，UCR1寄存器的bit0为UART使能位，bit14为自动检测波特率位，通常关闭。

4）编写数据收发函数，URXD寄存器的bit0~bit7保存接收到的数据，UTXD寄存器的bit0~bit7保存要发送的数据。

USR2为状态寄存器，bit0为1时表示已经接收到数据， bit3为1时表示数据发送完成。

注：1）编译时，若提示Waring：conflicting types for built-in function，表示我们编写的函数（putc()、getc()、puts()）与内建

函数发生冲突。则需在[例程通用MakeFile](#例程通用MakeFile)文件中如下位置加入选项“-fno-builtin”。“-fno-builtin”表示不使用内建函数，只使用自己创建的函数。

$(SOBJS) : obj/%.o : %.S

$(CC) -Wall -nostdlib -fno-builtin -c -O2 $(INCLUDE) -o $@ $<

$(COBJS) : obj/%.o : %.c

$(CC) -Wall -nostdlib -fno-builtin -c -O2 $(INCLUDE) -o $@ $<

2）配置UFCR、UBIR和UBMR设置波特率时，必须在写UBMR之前写UBIR。UBIR和UBMR两寄存器需一起配置，

如果只有一个寄存器被写入，则波特率仍为之前的值。（见参考手册3592页）

3）NXP官方提供了自动计算波特率的函数uart\_setbaudrate()，见正点原子裸机例程。使用本函数时，编译会提示许多

未定义，原因是此函数中使用到了除法运算，所以需要在MakeFile文件中链接数学库，使用如下指令指定库路径。

LIBPATH := -lgcc -L /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86\_64\_arm-linux-gnueabihf/

lib/gcc/arm-linux-gnueabihf/4.9.4

-lgcc -L后面为GCC编译器库函数路径，同理，第三方软件的库也可通过此方式进行链接。加入库路径后，链接

时将该路径链接进去即可，指令如下。

$(TARGET).bin : $(OBJS)

$(LD) -Timx6ul.lds -o $(TARGET).elf $^ $(LIBPATH)

$(OBJCOPY) -O binary -S $(TARGET).elf $@

$(OBJDUMP) -D -m arm $(TARGET).elf > $(TARGET).dis

再次编译，会提示raise未定义，按如下方式定义一个名为raise的空函数即可。

void raise(int sig\_nr)

{

}

4）可使用printf函数进行格式化输出，使用printf函数前需要移植相关头文件。将正点原子资料中的stdio文件夹复制

到工程中，此外还需要2个函数接口：putc()，getc()，例程中已经定义。在MakeFile文件中包含对应的头文件路径和源码路径后，编译报错：thumb conditional instruction should be in IT block -- `addcs r5,r5,#65536'，解决方法为：在编译C文件时添加“-Wa,-mimplicit-it=thumb”即可，如下：

$(COBJS) : obj/%.o : %.c

$(CC) -Wall -nostdlib -fno-builtin -Wa,-mimplicit-it=thumb -c -O2 $(INCLUDE) -o $@ $<

再次编译，编译成功。注意，我们移植的printf函数不支持浮点计算和输出，否则程序会卡死。

1. **DDR3**

①**RAM、ROM、SRAM、SDRAM和DDR**

**RAM**：**随机存取存储器**，可读写，为易失性存储器，一般用来保存程序数据、中间结果。使用电容作为存储单位，电

容存储的电荷会随时间的流逝而消失，所以存储器需要每隔一段时间给存储器内部的电容器进行一次充电或者一直保持充电状态。它的访问速度相对于ROM较快，因为只需要读取电容里的电荷值然后转化成对应的数字即可。

**ROM**：**只读存储器**，它是只读的。随着技术的发展，后来设计出了可以方便写入数据的ROM，而这个名称被沿用下

来了，现在一般用于指代非易失性半导体存储器。ROM掉电以后数据不会丢失，适合用来存储资料，比如音乐、图片、视频等信息。但向ROM或者Flash写入数据时，要先进行擦除，然后再发送要写的地址或扇区，最后才是要写入的数据，比RAM复杂很多。

**SRAM**：**静态随机存储器**，存储单元以锁存器来存储数据。这种电路结构不需要定时刷新充电，就能保持状态，只需

要一直向其发送特定的电流就可以了。SRAM价格很贵，通常作为SOC的内部RAM或Cache使用。

**SDRAM**：**同步动态随机存储器**，使用同步通讯，与SRAM相比，SDRAM 集成度高、功耗低、成本低、适合做大容量

存储，但是需要定时刷新来保证数据不会丢失。由于使用时钟同步的通讯速度更快，所以SDRAM使用更为广泛。

**DDR**：**双倍速率 SDRAM**，是SDRAM的升级版本， SDRAM 在一个 CLK周期内上升沿传输一次数据，DDR在一个

CLK周期传输两次数据，也就是在上升沿和下降沿各传输一次数据，这个概念叫做预取(prefetch)，相当于DDR的预取为2bit，因此DDR的速度直接加倍！随着技术的发展，DDR2在DDR基础上进一步增加预取(prefetch)，增加到了4bit，DDR3在DDR2的基础上将预取(prefetch)提高到8bit。

综上，RAM速度快，可以直接和CPU进行通信，但是掉电以后数据会丢失，容量不容易做大；ROM(Flash)速度虽然慢，但是容量大、适合存储数据。根据RAM的存储机制，又分为动态随机存储器DRAM以及静态随机存储器SRAM两种。DRAM中使用最广泛的是SDRAM，适合用来做内存条；SRAM适合做高速缓存或MCU内部的 RAM。为了进一步提高SDRAM的传输速度，设计出了DDR。

②**DDR3L**（不同厂家的DDR3L硬件相同，只是在驱动上可能有细微差异，不同芯片的DDR控制器不同）

I.MX6U的MMDC外设用于连接DDR，支持LPDDR2、DDR3、DDR3L，最高支持16位数据位宽、400MHz的时钟频率，800MT/S传输速率。I.MX6U为DDR提供了专用的IO（不可复用）与DDR进行连接（见参考手册2206页）。

LPDDR3、DDR3和DDR3L都是DDR3，区别主要在于工作电压。LPDDR3叫做低功耗 DDR3，工作电压为1.2V。DDR3叫做标压 DDR3，工作电压为1.5V，一般台式内存条都是DDR3。DDR3L是低压DDR3，工作电压为1.35V，一般手机、嵌入式、笔记本等都使用DDR3L。

本[开发板](#开发板)使用的RAM（DDR3L）型号为NT5CC256M16EP-EK，大小为512MB，位宽为16位，故其容量为4GB，工作电压为1.35V，传输速率为1866MT/S，框图如下。

图示, 示意图

描述已自动生成

**DDR3L结构框图**

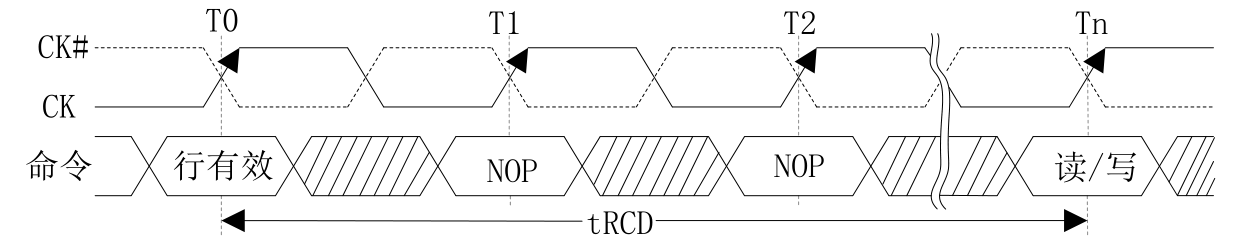
DDR3内部是一个存储阵列，可想象成一张表格，指定行地址和列地址就可以准确地找到所需要的单元格。对于内存，这个单元格成为存储单元，这个表格就是逻辑Bank。由于技术、成本等原因，不能做一个全容量的Bank，而且单一的BANK会带来严重的寻址冲突，减低内存访问效率。因此人们将其分隔成多块Bank。如图，DDDR3L中有8块Bank。

寻址的流程为：指定Bank地址，再指定行地址，然后指定列地址确定最终的寻址单元。在实际工作中，Bank地

址和行地址是同时发出的，此时这个命令称为“行激活”，之后发送列地址和读写指令，这两个指令也是同时发出的，所以一般都会以读写命令表示列寻址，下面介绍几个重要时间参数。

1）**传输速率**：常用的传输速率有1066MT/S、1600MT/S、1866MT/S等，不同传输速率，其时序也是不同的。

2）**tRCD**：行寻址（RAS）到列寻址（CAS）之间的延迟， 行有效(行激活)到读写命令发出的这段时间间隔叫做tRCD。



3）**CL**：列地址选通潜伏期，当列地址发出后就会触发数据传输，但数据从存储单元到IO口还需要一段时间，这段时

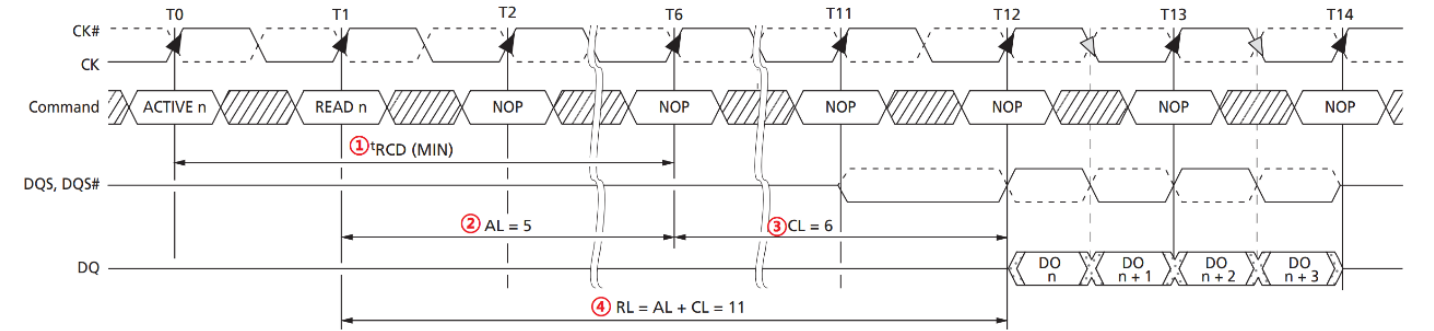
间就是CL。，一般tRCD和CL大小一样。

图示

描述已自动生成

4）**AL**：为了解决DDR中的指令冲突，提出了前置CAS，它允许CAS信号紧随着RAS发送，但读写没有因此提前，

仍要保证足够的潜伏期。AL和CL 组成了RL。



5）**tRC：**tRC是两个ACTIVE命令，或者ACTIVE命令到REFRESH命令之间的周期，DDR3L数据手册会给出这

个值，比如NT5CC256M16EP-EK的tRC值为47.91ns。

6）**tRAS**：tRAS是ACTIVE命令到PRECHARGE命令之间的最小时间，DDR3L的数据手册同样也会给出此参数，

NT5CC256M16EP-EK的tRAS值为34ns。

③**DDR时钟配置**

根据[时钟树](#时钟树图)对DDR时钟进行配置，选择PFD2(396MHz)作为DDR时钟源，MMDC\_CLK\_ROOT最大频率为396MHz （见参考手册643页），在前面[时钟配置](#配置PFD时钟)的例程中已经将PFD2设置为了396MHz。对于其他寄存器设置，I.MX6U内部的BOOT ROM已经设置好了，BOOT ROM已经设置 PLL2\_PFD2作为MMDC的最终时钟源，这就是I.MX6U的DDR频率为400MHz（实际只有 396MHz）的原因。

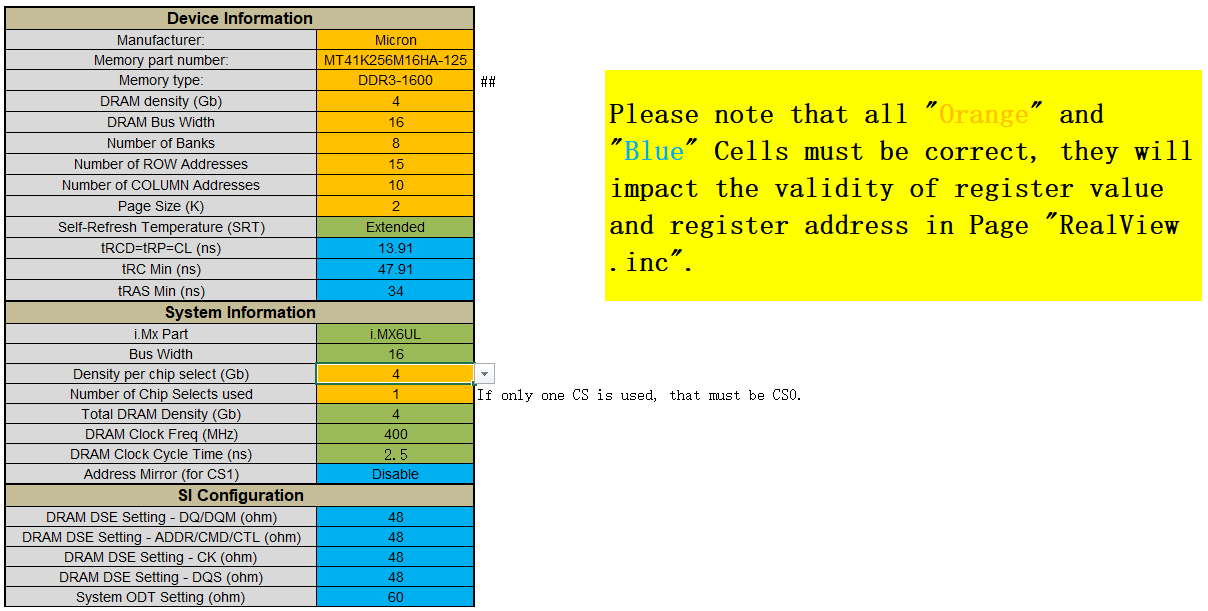
④**DDR3L初始化与测试**

我们使用NXP提供的DDR初始化工具ddr\_stress\_tester完成DDR3的初始化和测试，此工具通过USB OTG接口与开发板相连接，使用此工具需要一个配置文件，为Excel表。通过配置这个Excel表格生成一个.inc文件，这个文件内就包含了DDR初始化的信息，一般为寄存器地址和对应的寄存器值。ddr\_stress\_tester工具读取.inc文件后将其下载到I.MX6U中即可完成DDR初始化，下面具体介绍。

1）**工具准备：**测试工具为DDR\_Tester.exe，配置文件为I.MX6UL\_DDR3\_Script\_Aid\_V0.02.xlsx。

2）**生成.inc文件**：打开配置文件，文件中有三个标签：Readme、Register Configuration和RealView.inc。Register

Configuration标签下有一个表格，根据DDR数据手册的相关参数配置表格后，就会自动生成.inc文件。根据本开发板DDR信息配置表格图示如下，RealView.inc标签下的信息也会同步更新，可使用Notepad++将全部信息保存为.inc文件。



3）**.inc文件介绍**

这个.inc文件就是DDR3L的初始化脚本，其中包括了时钟、IO等初始化。之前介绍的，我们生成的.bin文件不是直接烧录到SD卡中的，烧录的是使用正点原子的烧录工具imxdownload在.bin文件加上头部信息后的.imx文件。可查看[镜像文件介绍](#镜像文件介绍)，.imx文件包含四部分：IVT + Boot data + DCD +用户代码及数据，其中DCD为设备配置信息，该部分的数据就是根据.inc文件进行配置的。

所以后续如果使用imxdownload对其他开发板烧录程序时，要根据对应的DDR的.inc将DCD中的值进行更改，程序才能正常运行。更改方式为，进入imxdownload.h文件，即可看到头部文件信息，搜索.inc中的寄存器地址，地址后面的就是寄存器值，将对应的值进行更改保存即可。注意，imxdownload.h中的寄存器地址是小端存储的，所以搜索时要进行倒序，如要更改地址0X020E04B4的值时，要搜索0XB4040E02。

4）**烧写.inc文件到芯片**：把SD卡弹出，将开发板设置为USB启动，见[启动设备选择](#启动设备选择)。打开DDR\_Tester软件，点击

Load Init Script,选择上面保存的.inc文件。TARGET选择MX6ULL，DDR Density选择512MB，ARM Speed选择528MHz(也可以尝试其他选项)，取消勾选Verify DCD Address，DDR Freq设置为400，其他选项默认即可，点击Download按钮进行下载，若下载成功，窗口会显示DDR信息，说明DDR初始化完成，图示如下。

（注意！.inc文件的保存路径不要有中文或者中文符号，否则无法打开，下载时开发板需要供电）

图形用户界面

描述已自动生成

5）**DDR校准**：此软件有三个测试栏：DDR Calibration、DDR Stress Test和32bit Memory Read/Write。DDR Calibration

为DDR校准，DDR Stress Test为DDR超频测试。注意，进行超频测试前需要进行DDR校准，否则容易卡死。校准流程如下，点击DDR Calibration下的Calibration按钮，窗口会打印校准信息，校准需要10分钟左右。窗口打印如下信息说明校准信息打印完成，这些信息就是我们需要校准的。

文本

描述已自动生成

打开.inc文件，搜索以上6个寄存器地址，按等号后面的字值对寄存器值进行更改。注意，寄存器0x021b0810和0x021b0840可能搜索不到，将能搜到的进行更改即可。修改完成后保存，再次下载到芯片中，即可进行超频测试。

6）**超频测试**：在DDR Stress Test标签下，设置Start Freq为400，End Freq为600，当指定开始频率和结束频率后，点

击Stress Test按钮则开始超频测试。一般DDR可以运行到超过标准10%~15%，则认为DDR完全满足要求。

1. **RGBLCD**

①**基础介绍**

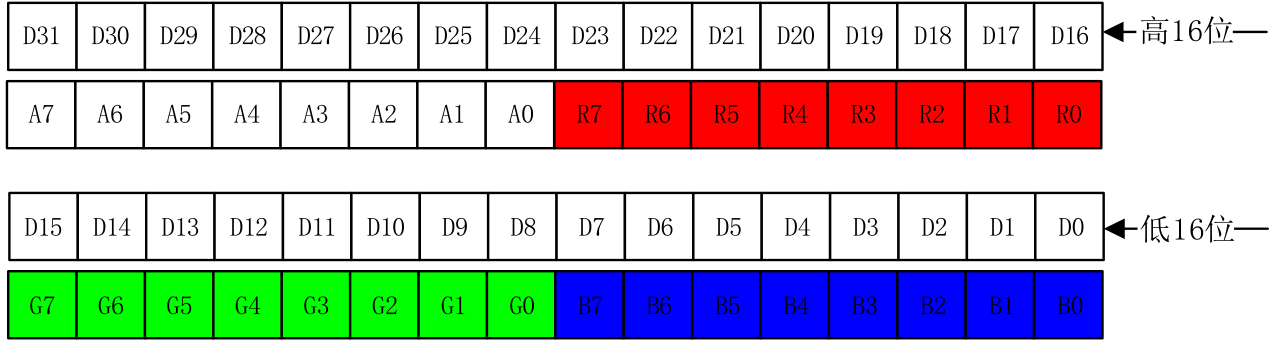
液晶显示器，主要参数有分辨率、像素格式、屏幕接口、时间参数、屏幕时序、像素时钟和显存等，下面简单介绍。

1）**分辨率**：常见的分辨率有720P、1080P、2K和4K等。以1080P为例， 1080p一般是指这个屏幕一列1080个像素

点，一共1920列。分辨率不变的情况下，LCD尺寸越小越清晰。

2）**像素格式**：常见的像素格式有RGB888、RGB565、ARGB8888，本开发板使用的像素格式就是ARGB8888。以

RGB888为例，RGB代表红绿蓝三种颜色，888表示每种颜色用8bit表示，通过调节不同颜色的比例即可显示不同的颜色。ARGB8888是在RGB基础上加上了透明通道A，也用8bit表示，即一个像素点使用32bit进行表示，示意图如下。



3）**屏幕接口**：常见的屏幕接口有SPI接口、MDDI接口、VSYNC接口、MCU接口（如8080接口）和RGB接口等，

本开发板使用的是RGB LCD，使用RGB模式接口。RGB模式接口使用TTL电平信号，属于并行接口，包含如下信号线。RGB LCD 一般有两种驱动模式：DE模式和HV模式，DE模式需要用到DE信号线，而HV模式不需要用到DE信号。在DE模式下是可以不需要HSYNC信号线的，即使不接HSYNC信号线LCD也可以正常工作。

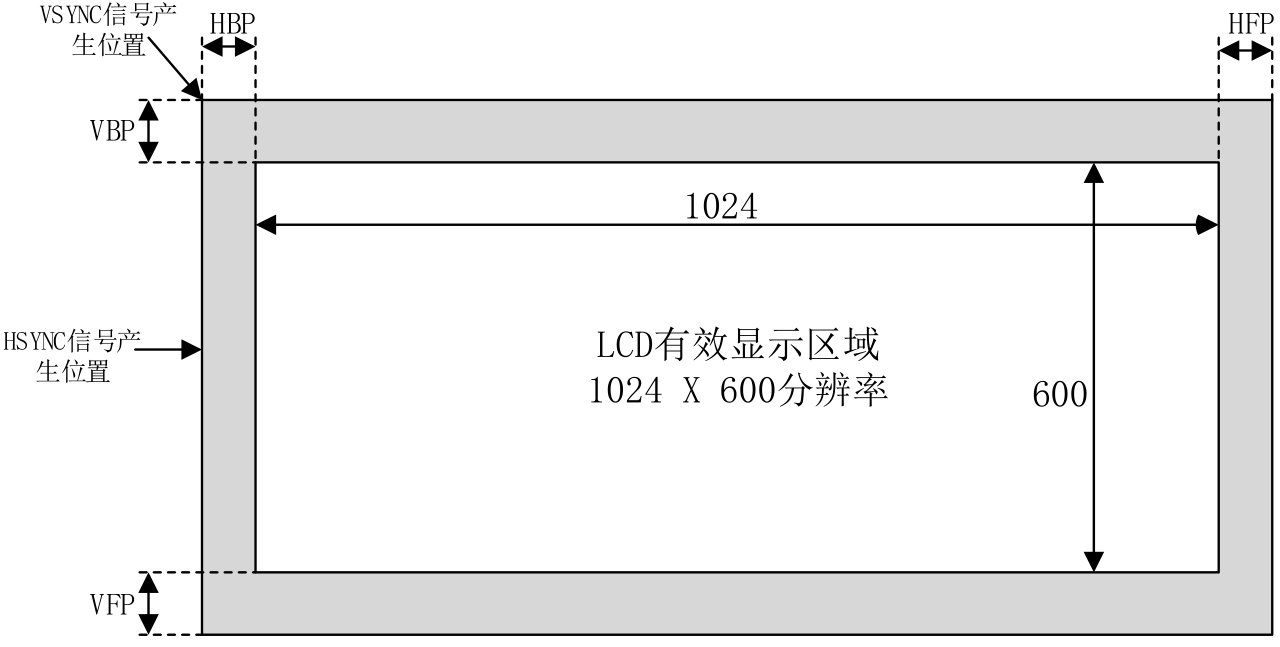
|  |  |
| --- | --- |
| **信号线名称** | **说明** |
| **RGB信号线** | RGB信号线各有8根，分别用于表示液晶屏一个像素点的红、绿、蓝颜色分量 |
| **CLK** | 同步时钟信号，液晶屏与外部使用同步通讯方式，每个时钟传输一个像素点数据 |
| **HSYNC** | 水平同步信号，在每进行一行的扫描之前，该信号就有效一次，每传输完成一行像素数据时，HSYNC会发生电平跳变 |
| **VSYNC** | 垂直同步信号，在每进行一帧的扫描之前，该信号就有效一次，每传输完成一帧像素数据时，VSYNC 会发生电平跳变 |
| **DE** | 数据使能信号，用于表示数据的有效性，高电平时RGB信号线表示的数据有效 |

为了提高程序的兼容性，正点原子的RGB LED通过控制R7、G7、B7三条信号线上拉/下拉即可设置LCD的ID，帮助MCU判断当前LCD的分辨率等相关参数，ID与LCD型号对应关系如下，我买的是1024\*600分辨率的。因为配置这三根信号线使用的是弱上拉和弱下拉，所以不会影响LCD屏幕的正常显示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **R7** | **G7** | **B7** | **ID** | **LCD型号** |
| 0 | 0 | 0 | 4342 | ATK-4342 RGBLCD 模块，分辨率：480\*272 |
| 0 | 0 | 1 | 7084 | ATK-7084 RGBLCD 模块，分辨率：800\*480 |
| 0 | 1 | 0 | 7016 | ATK-7016，RGBLCD 模块，分辨率：1024\*600 |
| 1 | 0 | 0 | 4384 | ATK-4384，RGBLCD 模块，分辨率：800\*480 |

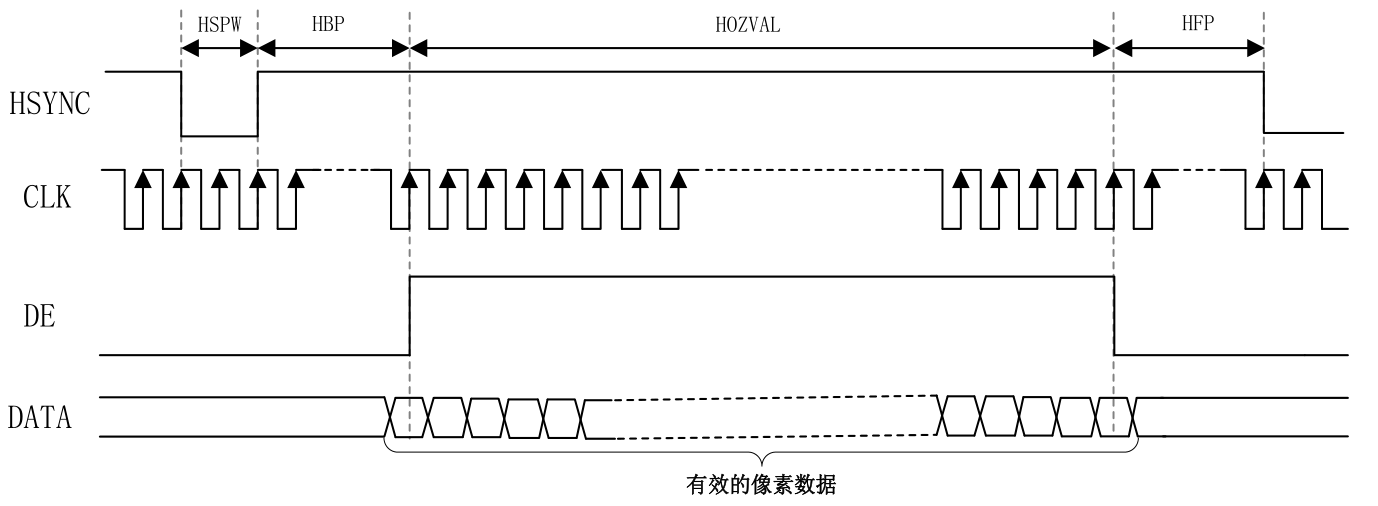
4）**时间参数**；RGB LCD有4个重要的时间参数，分别为HBP、HFP、VBP和VFP。RGB LCD是从左到右、从上到下，

挨个扫描像素点进行显示图像的。一行扫描完后，HSYNC信号会发生电平跳变，一帧图像扫描完后，VSYNC 会发生电平跳变。一行像素点扫描完到HSYNC信号跳变需要一段时间，即HFP；HSYNC信号跳变结束到开始扫描下一行需要一段时间，即HBP；一帧图像扫描完到HSYNC信号跳变需要一段时间，即VFP；HSYNC信号跳变结束到开始扫描下一帧图像需要一段时间，即VBP。这4个参数的主要目的是为了锁定有效的像素数据，编写LCD驱动时需要使用这4个参数，具体数值需要查看对应的LCD数据手册，参数示意图如下。



5）**屏幕时序**

a. **行显示时序**（由时序图知，显示一行数据所需要的时间为：HSPW + HBP + HOZVAL + HFP）



**HSPW(thp)**：HSYNC信号宽度，即HSYNC信号持续时间。HSYNC信号不是一个脉冲，而是需要持续一段时间

才是有效的，单位为CLK。

**HBP(thb)**：HSYNC信号跳变结束到开始扫描下一行所需要的时间。

**HOZVAL(thd)**：显示一行数据所需的时间，假如屏幕分辨率为1024\*600，则HOZVAL就是1024，单位为CLK。

**HFP(thf)**：一行像素点扫描完到HSYNC信号跳变所需要的时间。

b. **帧显示时序**

形状, 矩形

描述已自动生成

（由时序图知，显示一帧数据所需时间为：T = (VSPW+VBP+LINE+VFP) \* (HSPW + HBP + HOZVAL + HFP)）

**VSPW(tvp)**：VSYNC信号宽度，也就是VSYNC信号持续时间，单位为1行的时间(th)。

**VBP(tvb)**：HSYNC信号跳变结束到开始扫描下一帧图像所需要的时间。

**LINE(tvd)**：显示一帧有效数据所需的时间，假如屏幕分辨率为1024\*600，那么LINE就是600行的时间。

**VFP(tvf)：**一帧图像扫描完到HSYNC信号跳变所需要的时间。

c. **RGB LCD参数**（ATK-7016型号，分辨率：1024\*600）

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **值** | **参数** | **值** |
| 水平显示区域 | 1024 tCLK | 垂直显示区域 | 600 th |
| HSPW(thp) | 20 tCLK | VSPW(tvp) | 3 th |
| HBP(thb) | 140 tCLK | VBP(tvb) | 20 th |
| HFP(thf) | 160 tCLK | VFP(tvf) | 12 th |

6）**像素时钟**

像素时钟就是RGB LCD的时钟信号，以ATK7016这款屏幕为例，显示一帧图像所需要的时钟数计算如下：

(VSPW+VBP+LINE+VFP) \* (HSPW + HBP + HOZVAL + HFP)

= (3 + 20 + 600 + 12) \* (20 + 140 + 1024 + 160)

= 853440

即显示一帧图像需要853440个时钟，显示60帧就是853440 \* 60 = 51206400≈51.2M，即像素时钟为MHz。根据[时钟树](#时钟树图)对像素时钟进行配置，在[时钟系统](#时钟系统)中介绍过，PLL5用于给显示设备提供时钟，所以配置CSCDR2寄存器的LCDIF1\_CLK\_SEL位为0，LCDIF1\_PRE\_CLK\_SEL位为2，将RGB LCD时钟源设置为由PLL5提供。

a. **配置PLL5为768MHz（**见参考手册683页）

下面对PLL5进行配置，配置CCM\_PLL\_VIDEO、CCM\_PLL\_VIDEO\_NUM、CCM\_PLL\_VIDEO\_DENOM、CCM\_MISC2这四个寄存器对PLL5进行设置，其中CCM\_PLL\_VIDEO\_NUM和CCM\_PLL\_VIDEO\_DENOM这两个寄存器是用于小数分频的，为了计算简单，直接设置为0，不配置。PLL5的时钟计算公式如下：

PLL5\_CLK = OSC24M \* (loopDivider + (denominator / numerator)) / postDivider

= OSC24M \* loopDivider / postDivider

配置寄存器CCM\_PLL\_VIDEO的POST\_DIV\_SLECT位和CCM\_ANALOG\_CCMSC2的VIDEO\_DIV位对postDivider进行配置。将POST\_DIV\_SLECT位设置为2， VIDEO\_DIV位设置为0，即设置postDivider为1；配置寄存器CCM\_PLL\_VIDEO的ENABLE位为1，使能PLL5；配置寄存器CCM\_PLL\_VIDEO的DIV\_SELECT位为32，即设置loopDivider的值为32。此时，PLL5\_CLK = 24 \* 32 / 1 = 768MHz。

b. **对PLL5进行分频，得到51.2MHz**（见参考手册2165页**）**

根据时钟树对其配置分频，将频率调为51.2MHz即可。由时钟树知，配置寄存器CCM\_CSCDR2的LCDIF1\_PRED位为2，即3分频，配置寄存CCM\_CBCMR的LCDIF1\_PODF位为4，即5分频。此时最终进入LCDIF的时钟频率为768 / 3 / 5 = 51.2MHz。

7）**显存**

专门用于存储显示数据的存储器。ARGB8888像素格式，1个像素需要32bit，即1个像素数据需要4字节内存来存放，1024\*600 分辨率则需要1024\*600\*4 = 2457600B，约等于2.4MB。由于RGB LCD内部没有内存，所以需要在DDR中分出一段内存作为屏幕的显存，直接定义一个32位的数组，直接操作这个数组即可控制屏幕显示的内容。

②**eLCDIF接口**

eLCDIF是I.MX6U自带的液晶屏幕接口，用于连接RGB LCD接口的屏幕。eLCDIF支持三种接口：MPU接口、VSYNC 接口和DOTCLK 接口，本教程使用的是DOTCLK接口，介绍如下。

1）**MPU接口**：MPU接口用于在I.MX6U和LCD屏幕直接传输数据和命令，这个接口用于6080/8080接口的LCD屏

幕，比如学习STM32的时候常用到的MCU屏幕。

2）**VSYNC接口**：VSYNC接口和MPU接口时序基本一样，只是多了VSYNC信号来作为帧同步。

3）**DOTCLK接口**：DOTCLK接口通常被称为RGB接口，DOTCLK接口就是用来连接RGB LCD接口屏幕的，它包

括VSYNC、HSYNC、DOTCLK和 ENABLE(可选的)这四个信号。

③**配置eLCDIF接口**（具体寄存器信息见参考手册2165页）

1）**寄存器LCDIF\_CTRL**

RUN位为运行控制位，为1时开始传输数据；DATA\_FORMAT\_24\_BIT位为0时表示24位数据有效，我们使用的是RGB888格式，所以该位设置为0，只有当WORD\_LENGTH为3时此位才有效，所以要设置WORD\_LENGTH为3；MASTER位设置eLCDIF接口工作在主机模式下，要设置为1；LCD\_DATABUS\_WIDTH位设置总线宽度，设置为3，即24位总线宽度；CSC\_DATA\_SWIZZLE位设置字节交换，设置为0即可，比较换；INPUT\_DATA\_SWIZZLE也设置为0，不进行字节交换；DOTCLK\_MODE位为1时表示工作在DOTCLK模式下，配置为1；如果工作在DOTCLK 模式下，BYPASS\_COUNT位必须为1；CLKGATE位必须为1，否则时钟就进入不到eLCDIF；SFTRST位为软件复位，为1时强制复位LCD，将此位置0。

2)**寄存器LCDIF\_CTRL1**

BYTE\_PACKING\_FORMAT位决定32位数据中哪些字节有效，将此位设置为7，表示传输24位数据格式，即

ARGB格式，但A通道不传输。这个寄存器只配置了这一位。

3）**寄存器LCDIF\_TRANSFER\_COUNT**

该寄存器用来设置LCD屏幕分辨率大小，高16位为V\_COUNT，设置垂直分辨率，低16位为H\_COUNT，设置水平分辨率。如LCD分辨率为1024\*600，则设置V\_COUNT为600，H\_COUNT为1024。

4）**寄存器LCDIF\_VDCTRL0**

该寄存器用于设置VSYNC和DOTCLK模式下信号线的极性。信号线的极性需要根据LCD手册进行配置。

VSYNC\_PULSE\_WIDTH位设置VSPW；VSYNC\_PULSE\_WIDTH\_UNIT设置VSYNC信号的脉冲宽度单位，DOTCLK 模式下要设置为1；VSYNC\_PERIOD\_UNIT位设置VSYNC信号的周期单位，DOTCLK模式下要设置为1；ENABLE\_POL为数据线极性设置位，为0时低电平有效，为1时高电平有效；DOTCLK\_POL位为像素时钟CLK的极性设置位，为0时下降沿锁存数据，上升沿捕获数据，为1时上升沿所存数据，下降沿捕获数据；HSYNC\_POL位设置HSYNC的极性，为0时低电平有效，为1时高电平有效；VSYNC\_POL位设置VSYNC的极性，为0时低电平有效，为1时高电平有效；ENABLE\_PRESENT位为DE数据线使能位，为1时使能DE数据线，为0时不使能；VSYNC\_OEB位是VSYNC信号方向控制位，为0时VSYNC是输出，为1时VSYNC是输入。

5）**寄存器 LCDIF\_VDCTRL1**

该寄存器用于设置VSYNC总周期，即：屏幕高度+VSPW+VBP+VFP。

6）**寄存器 LCDIF\_VDCTRL2**

该寄存器分为高16位和低16位，高16位为HSYNC\_PULSE\_WIDTH，设置HSYNC信号宽度，即HSPW；低16位是HSYNC\_PERIOD，设置HSYNC总周期，即：屏幕宽度+HSPW+HBP+HFP。

7）**寄存器 LCDIF\_VDCTRL3**

该寄存器的VERTICAL\_WAIR\_CNT位用于设置VSYNC信号产生到有效数据产生之间的时间，即VSPW+VBP；HORIZONTAL\_WAIT\_CNT位用于设置HSYNC信号产生到有效数据产生之间的时间，即HSPW+HBP。

8）**寄存器 LCDIF\_VDCTRL4**

该寄存器的DOTCLK\_H\_VALID\_DATA\_CNT位用于设置LCD的宽度，即水平像素数量；SYNC\_SIGNALS\_ON位为同步信号使能位，为1时使能VSYNC、HSYNC、DOTCLK这些信号。

9）**寄存器LCDIF\_CUR\_BUF**

该寄存器用于设置LCD当前显存的首地址，通常在DDR中定义一个数据作为显存，则数组名即为显存首地址。

10）**寄存器LCDIF\_NEXT\_BUF**

该寄存器用于设置LCD下一帧数据的首地址，这里我们设置该寄存器与寄存器LCDIF\_CUR\_BUF相同。

1. **RTC实时时钟**

I.MX6U的RTC在外设SNVS内部，SNVS为安全非易失性存储。SNVS分为两部分：SNVS\_HP 和 SNVS\_LP，二者区别如下。SNVS\_HP内部有一个RTC，SNVS\_LP内部有一个SRTC，由SNVS\_HP 和 SNVS\_LP的区别知，主电源断电后RTC会清零，而SRTC由备用电源供电继续运行，本节使用的RTC实时时钟就是SNVS\_LP内部的SRTC。

**SNVS\_HP**：电源来源为系统电源，主电源断电后SNVS\_HP也会断电。

**SNVS\_LP**：电源来源为系统电源和备用电源（纽扣电池），主电源断电后可由备用电源为其供电。

SRTC需要外界提供一个32.768MHz的时钟，打开RTC后，RTC就开始工作，主电源断电后可由备用电源继续供

电。RTC时间一般以1970年1月1日为起点，加上经过的秒数即可得到现在的时间和日期。下面介绍配置RTC用到的寄存器。

1）**SNVS\_HPCOMR**：只用到了NPSWA\_EN位，该位为非特权软件访问控制位，我们将其置1，即所有软件都

可访问SNVS；

2）**SNVS\_LPCR**：只用到了SRTC\_ENV位，将该位置1，即打开STC。

3）**SNVS\_LPSRTCMR和SNVS\_LPSRTCLR**：保存着秒数，直接读取这两个寄存器的值就知道过了多长时间了。

SNVS\_SRTCMR的bit0~14位保存计数器的高15位，SNVS\_SRTCLR的bit15~31保存计数器的低17位。

在之前移植的NXP官方SDK包是针对I.MX6ULL编写的，其中MCIMX6Y2.h中结构体SNVS\_Type内的寄存器是不全的，所以需要补全寄存器，修改后的SNVS\_Type结构体如下。

typedef struct {

\_\_IO uint32\_t HPLR;

\_\_IO uint32\_t HPCOMR;

\_\_IO uint32\_t HPCR;

\_\_IO uint32\_t HPSICR;

\_\_IO uint32\_t HPSVCR;

\_\_IO uint32\_t HPSR;

\_\_IO uint32\_t HPSVSR;

\_\_IO uint32\_t HPHACIVR;

\_\_IO uint32\_t HPHACR;

\_\_IO uint32\_t HPRTCMR;

\_\_IO uint32\_t HPRTCLR;

\_\_IO uint32\_t HPTAMR;

\_\_IO uint32\_t HPTALR;

\_\_IO uint32\_t LPLR;

\_\_IO uint32\_t LPCR;

\_\_IO uint32\_t LPMKCR;

\_\_IO uint32\_t LPSVCR;

\_\_IO uint32\_t LPTGFCR;

\_\_IO uint32\_t LPTDCR;

\_\_IO uint32\_t LPSR;

\_\_IO uint32\_t LPSRTCMR;

\_\_IO uint32\_t LPSRTCLR;

\_\_IO uint32\_t LPTAR;

\_\_IO uint32\_t LPSMCMR;

\_\_IO uint32\_t LPSMCLR;

}SNVS\_Type;

**注**：1）本节笔记参考《I.MX6UL参考手册》的SNVS章节，《I.MX6ULL参考手册》的SNVS章节内容不完整。

2）SNVS章节内有些内容与加密有关，需要与NXP签订保密协议才能获取。

3）在实际产品应用中不要用芯片内部的RTC，内置RTC精度较差，建议使用外部RTC芯片。

4）参考手册中说SNVS\_SRTCMR保存着高15位，SNVS\_SRTCLR保存着低32位，这是错误的。

5）时间混乱时，要仔细查看SNVS\_LPSRTCMR和SNVS\_LPSRTCLR寄存器是否读取正确。

6）SRTC还有闹钟功能，可在寄存器SNVS\_LPAR中写入闹钟时间值，当时钟值与闹钟值相等时就会触发闹钟中断。

1. **NXP官方SDK移植**

NXP针对I.MX6ULL编写了一个SDK包，类似于STM32的标准库。该SDK包有Windows和Linux两个版本，为了熟

悉Linux开发环境，为后续学习 Uboot 移植、Linux 移植和 Linux 驱动开发做准备，我们选择使用Linux版本的SDK包进行裸机开发。注意，NXP只为I.MX6ULL提供了SDK包，其他Cortex-A 内核的芯片基本不会提供，因此，Cotex-A 芯片都需要我们自行编写，比如.s 启动文件、寄存器定义等等。主要是对三个文件进行移植，分别为：fsl\_common.h、fsl\_iomuxc.h 和 MCIMX6Y2.h。移植过程比较复杂，且移植这三个文件主要是降低Linux 驱动开发入门难度，因此，我直接使用正点原子移植完成的文件。三个文件介绍如下：

①**fsl\_common.h**：通用宏定义的相关内容。

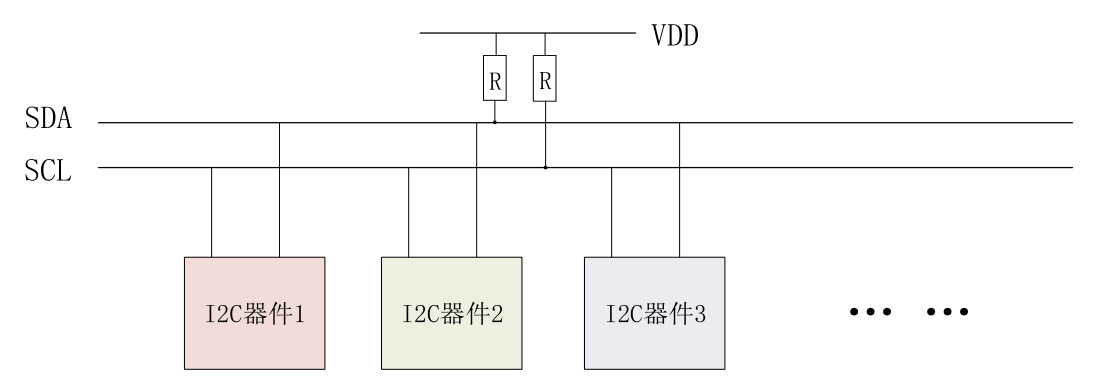
②**fsl\_iomuxc.h**：寄存器地址相的关内容。

③**MCIMX6Y2.h**：寄存器结构体和相关位的相关内容。

移植完成后，还需要创建一个cc.h文件，这个文件内包含了这三个文件中用到的变量类型，具体内容见裸机例程。

1. **I2C**

I2C是 NXP 公司设计的最常用的通信接口，众多的传感器都会提供I2C接口与主控相连接。I.MX6U 有 4 个 I2C 接口，可以通过这 4 个 I2C 接口来连接一些 I2C 外设。I.MX6U-ALPHA 开发板使用 I2C1 接口连接了一个距离传感器 AP3216C，下面介绍使用I2C接口与传感器 AP3216C进行数据传输。



**I2C多设备连接图**

如图， I2C使用两条线在主控制器和从机之间进行数据通信。一条是 SCL(串行时钟线)，另外一条是 SDA(串行数据线)，这两条数据线需要接上拉电阻，一般是 4.7K，总线空闲时 SCL 和 SDA 处于高电平。， I2C 总线标准模式下速度可以达到 100Kb/S，快速模式下可以达到 400Kb/S。I2C 控制器下可以挂多个 I2C 从设备，不同的 I2C从设备有不同的器件地址，这样 I2C 主控制器就可以通过 I2C 设备的器件地址访问指定的 I2C设备了。下面介绍I2C协议相关术语和读写时序。

①**I2C相关术语**

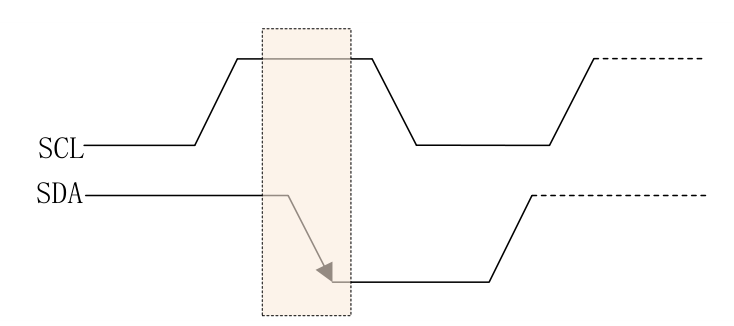
1）**起始位**：SCL 为高电平时，SDA 出现下降沿表示为起始位。

2）**停止位**：SCL 位高电平时，SDA出现上升沿表示为停止位。

3）**数据传输**：SCL 高电平期间，SDA 上的数据要保持稳定，SDA 上的数据变化只能在 SCL 低电平期间发生。

4）**应答**：I2C 主机发送完 8 位数据后会将 SDA 设置为输入状态，等待 I2C 从机应答，即从机告诉主机它收到数据了。

从机通过将 SDA 拉低来表示发出应答信号，表示通信成功，否则表示通信失败。

 矩形

描述已自动生成 形状, 多边形

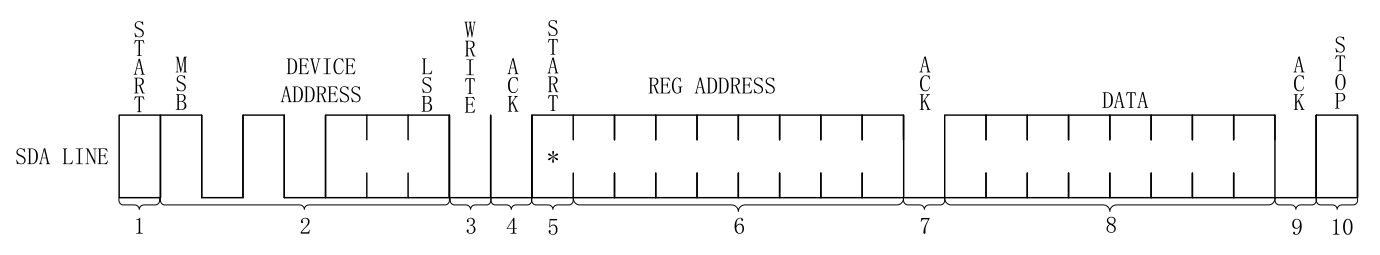
描述已自动生成

**起始位 数据传输 停止位**

②**读写时序**

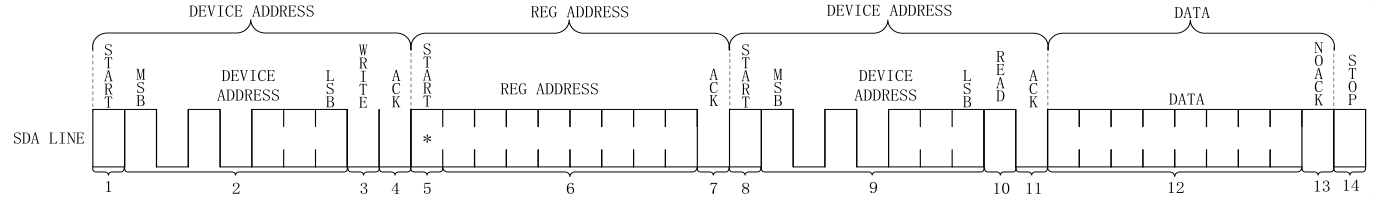
主机与从机进行I2C通讯就是进行读和写两个操作，下面介绍读、写时序。

1）**写时序**



如上图所示，主机发送开始信号，表示开始I2C通信；接着发送一个8位数据，高7位是设备地址，每个从机都有一个设备地址，主机通过设备地址与指定设备通讯；低1位是读写位，为1表示读操作，为0表示写操作，设置为0；这个8位字节发送完成后等待指定从设备应答，为0表示应答；从机应答后，主机重新发送开始信号，接着发送从机中要写入数据的寄存器地址，等待从机应答；从机应答后，主机就开始发送要写入的数据；接下来从机如果应答，则主机可继续发送数据，若从机非应答，则表示I2C通讯失败；主机发送结束信号后，I2C通讯结束。

2）**读时序**



如上图所示，主机发送开始信号，表示开始I2C通讯；接着发送一个8位数据，高7位为设备地址，低1位为读写位，设置为0，即写操作；等待从机应答，从机应答后，主机重新发送开始信号，接着发送从机中要读取数据的寄存器地址，等待从机应答；从机应答后，主机发送重新开始信号，接着重新发送从设备地址和读写位，注意此时读写位设置为1，即读操作，等待从机应答；从机应答后，从机开始向主机发送数据；接着主机若发送应答信号，则从机继续向主机发送数据，若主机发送非应答信号，则表示读取完成：主机发送结束信号后，I2C通讯结束。

③**I.MX6U的I2C外设**

I.MX6U有4个I2C外设，支持两种模式：标准模式（100Kbits/s）和快速模式（400Kbits/s）。下面对I2C的相关寄存器进行配置。主要配置以下5个寄存器： I2Cx\_IADR（地址寄存器）、I2Cx\_IFDR（波特率寄存器）、I2Cx\_I2CR（控制寄存器）、I2Cx\_I2SR（状态寄存器）和I2Cx\_I2DR（数据寄存器）。（具体寄存器信息见参考手册1463页）

1）**时钟源**：I2C有perclk\_clk\_root和ipg\_clk\_root两个时钟源。perclk\_clk\_root用于为从机寄存器进行读写提供时钟，

ipg\_clk\_root用于为I2C外设提供时钟。这两个时钟是同步的，前面已经将其均配置成了66MHz，见[时钟配置](#IPG_CLK_ROOT设置)。

2）**波特率**： I2Cx\_IFDR寄存器的IC位为分频值选择位。时钟源为66MHz，如果波特率为100KHz，则可设置IC为

0X15或0X38，即640分频，则波特率为：66000000/640=103.125KHz≈100KHz。（见参考手册1464页）

3）**使能I2C**： I2Cx\_I2CR寄存器的IEN位为I2C使能位，为1时使能I2C。

4）**主从模式**：I2Cx\_I2CR寄存器的MSTA位为主从模式选择位，为1表示主机，为0表示从机。

5）**停止信号**：I2Cx\_I2CR寄存器的MSTA位由1变为0时，会产生stop信号，并变为从模式。

6）**传输方向**：I2Cx\_I2CR寄存器的MTX位为方向选择位，为1时发送，为0时接收。

7）**应答使能**：I2Cx\_I2CR寄存器的TXAK位为应答使能位，为1时发送NO ACK 信号，为0时发送ACK 信号。

8）**重复开始**：I2Cx\_I2CR寄存器的RSTA位为重复开始位，为1时产生一个重新开始信号。

9）**数据缓存**：I2Cx\_I2DR寄存器为数据寄存器，低 8 位有效，发送数据时将要发送的数据写入到此寄存器，接收数据

时直接读取此寄存器即可得到接收到的数据。

10）**设备地址**：I2Cx\_IADR寄存器的ADR位用来保存从机设备地址，访问从设备时就需要将其地址写入到 ADR 里。

11）**传输状态**：I2Cx\_I2SR寄存器的ICF位为数据传输状态位，为1时表示数据传输完成，为0时表示数据正在传输。

12）**总线状态**：I2Cx\_I2SR寄存器的IBB位为总线状态标志位，为1时表示总线忙，为0时表示总线空闲。

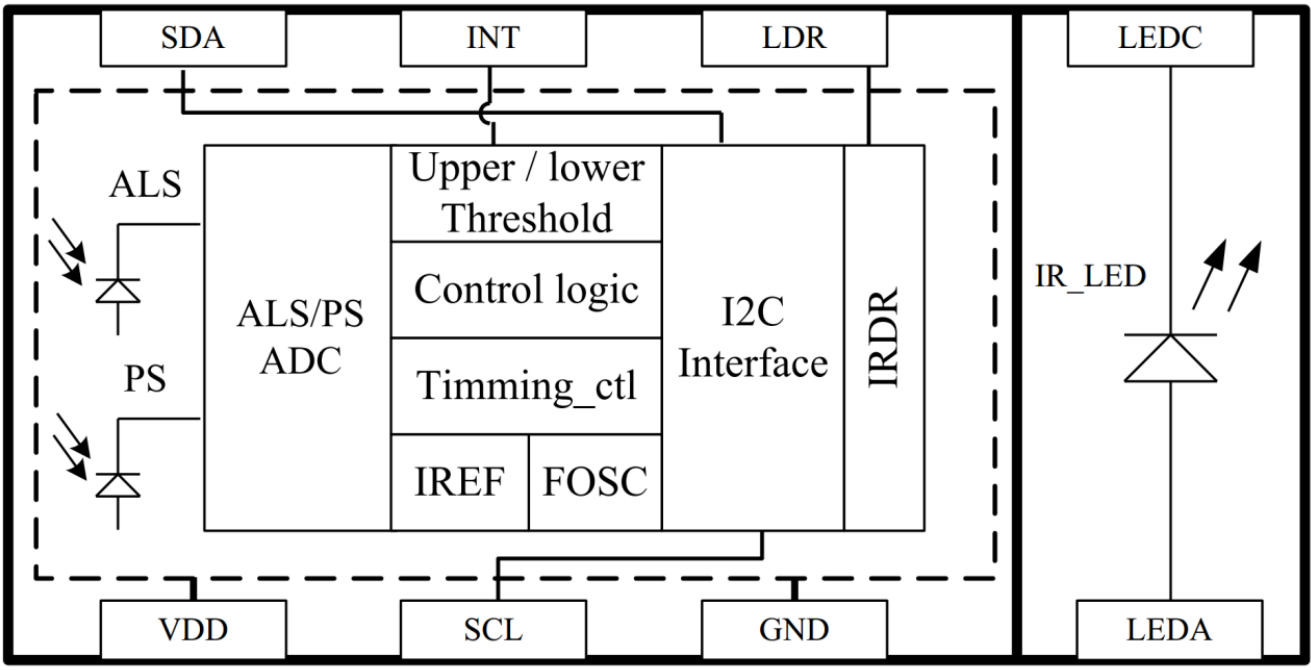
13）**应答确认**：I2Cx\_I2SR寄存器的RXAK位为应答信号标志位，为1时表示检测到NO ACK非应答信号，为0时表

示接收到ACK 应答信号。

14）**仲裁丢失**：I2Cx\_I2SR寄存器的IAL位为仲裁标志位，为1时表示仲裁丢失，为0时表示仲裁未丢失。

④**AP3216C**

教程中使用的是AP3216C作为I2C从机设备，下面进行简单介绍。AP3216C是由敦南科技推出的一款三合一环境传感器，其支持环境光强度(ALS)、接近距离(PS)和红外线强度(IR)这三个环境参数检测。ALS具有16位分辨率、PS 和IR具有10位分辨率。AP3216C常被用于手机、平板、导航设备等，其内置的接近传感器可以用于检测是否有物体接近，比如手机上用来检测耳朵是否接触听筒，如果检测到的话就表示正在打电话，手机就会关闭手机屏幕以省电。也可以使用环境光传感器检测光照强度，可以实现自动背光亮度调节，下面进行简单介绍。（具体AP3216C芯片信息需查看数据手册）



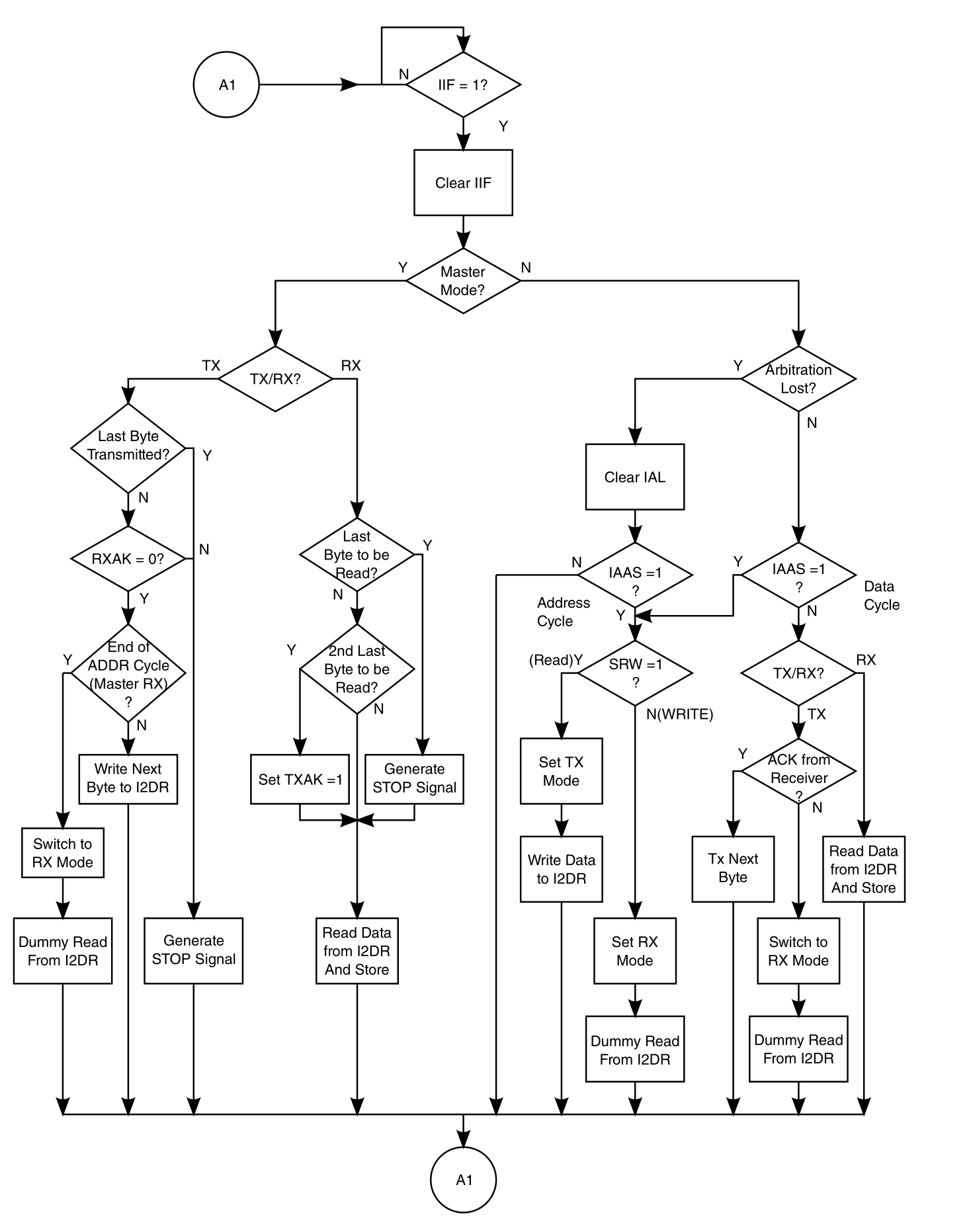
**AP3216C 结构图**

AP3216 的设备地址为 0X1E，其相关寄存器信息汇总如下。

|  |  |  |  |
| --- | --- | --- | --- |
| **寄存器地址** | **位** | **寄存器功能** | **描述** |
| **0X00** | 2:0 | 系统模式 | 000：掉电模式(默认)  001：使能 ALS  010：使能 PS+IR  011：使能 ALS+PS+IR  100：软复位  101：ALS 单次模式  110：PS+IR 单次模式  111：ALS+PS+IR 单次模式 |
| **0X0A** | 7 | IR 低位数据 | 0：IR&PS 数据有效 1:无效 |
| 1:0 | IR 最低 2 位数据 |
| **0X0B** | 7:0 | IR 高位数据 | IR 高 8 位数据 |
| **0X0C** | 7:0 | ALS 低位数据 | ALS 低 8 位数据 |
| **0X0D** | 7:0 | ALS 高位数据 | ALS 高 8 位数据 |
| **0X0E** | 7 | PS 低位数据 | 0：物体在远离 1：物体在接近 |
| 6 | 0：IR&PS 数据有效 1：IR&PS 数据无效 |
| 3:0 | PS 最低 4 位数据 |
| **0X0F** | 7 | PS 高位数据 | 0：物体在远离 1：物体在接近 |
| 6 | 0：IR&PS 数据有效 1：IR&PS 数据无效 |
| 5:0 | PS 最低 6 位数据 |

如上表所示，0X00 这个寄存器是模式控制寄存器，一般开始先将其设置为 0X04，即先软件复位一次 AP3216C，之后再根据实际使用情况选择合适的工作模式，后面我们设置为0X03，也就是开启 ALS+PS+IR。0X0A~0X0F这6个寄存器是数据寄存器，保存着ALS、PS和IR这三个传感器获取到的数据值。如果同时打开 ALS、PS 和 IR则读取间隔最少要 112.5ms，因为 AP3216C 完成一次转换需要112.5ms。本I2C实验使用AP3216C芯片与I.MX6U进行通讯，开机会先检测 AP3216C 是否存在，一般的芯片是有个 ID 寄存器，通过读取 ID 寄存器判断 ID 是否正确就可以检测芯片是否存在。但是 AP3216C 没有 ID 寄存器，所以我们就通过向寄存器 0X00 写入一个值，然后再读取 0X00 寄存器，判断读出得到值和写入的是否相等，如果相等就表示AP3216C存在，否则的话 AP3216C 就不存在。

⑤**I2C外设读写流程**



**I2C轮询程序流程图**

注：1）I.MX6ULL产生起始信号不需要按协议手动控制信号线高低电平，只需要向I2C\_I2DR寄存器写入从机地址和读写位， I.MX6ULL就会自动产生起始信号。（见参考手册1453页）

2）发送和接收数据前，要等待前面数据传输完成，即等待数据寄存器准备好进行数据传输，通过查看I2Cx\_I2SR寄存

器的ICF位判断数据传输状态，为1时表示数据传输完成，下次传输数据时，该标志位自动清除。

3）发送和接收数据后，也要检查标志位判断数据是否传输完成。传输完成后，I2Cx\_I2SR寄存器的ICF位会被置1，

同时，I2Cx\_I2SR的IIF位（中断状态）也会被置1，且必须手动清除IIF位。所以实际编写驱动时直接查看寄存器I2Cx\_I2SR的IIF位判断数据是否传输完成，随后手动清除该标志位。（见参考手册1454页）

4）读数据时，要在接收倒数第2个字节前，向从机发送非应答信号，在倒数第1个字节前发送停止信号。如主机想从

从机读取8个字节数据，则每接收一个字节后向从机发送一个应答信号，在接收第7个字节前，向从机发送非应答信号，随后从机发送第8个字节前发送停止信号。如果只接收1个字节，则直接向从机发送停止信号。通过配置I2Cx\_I2CR寄存器的TXAK(bit3)、MTX(bit4)、MSTA(bit5)位产生非应答信号和停止信号。

5）读数据时，在发送完从机地址和寄存器地址后，将MTX位置0，随后需要对数据寄存器I2Cx\_I2DR进行假读操

作，以便可以顺利读取下一个接收到的数据，假读代码如下。（见参考手册1458页）

volatile uint8\_t dummy = 0;

dummy++; //为了编译时避免警告

dummy = base->I2DR; //假读

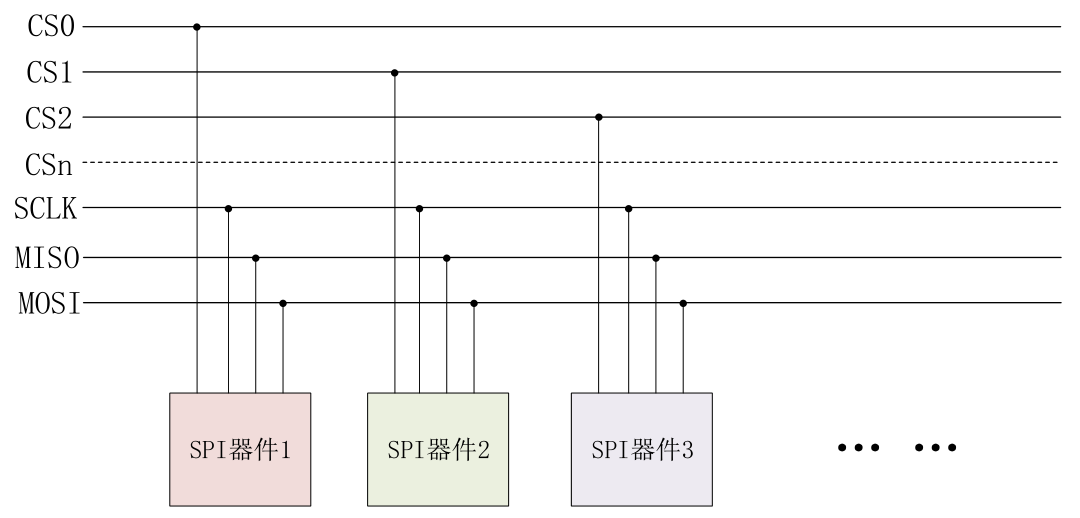
6）I2C驱动代码可参考NXP官方SDK包中例程，文件路径如下：（只有I.MX6ULL有SDK包）

[正点原子]阿尔法Linux开发板(A盘)-基础资料\07.I.MX6U参考资料\03.I.MX6ULL SDK包\devices\MCIMX6Y2\drivers

1. **SPI**

SPI 是 Motorola 公司推出的一种同步串行接口技术，是一种高速、全双工的同步通信总线，I.MX6U有4 个SPI接口。

与I2C相比，SPI通信速度更快，且为全双工通讯。I2C快速模式下最高速度为400KHz，SPI最高可达上百MHz。一般 SPI需要4 根线，可以使用三根线(单向传输)。4根信号线分别为CS/SS（片选）、SCK（时钟）、MOSI/SDO（主出从入）、MISO/SDI（主入从出）。如下图所示，SPI 通信都是由主机发起的且需要提供通信的时钟信号。SPI有四种工作模式，主要区别是总线空闲时SCLK的时钟状态以及数据采样时刻，由CPOL和CPHA决定，SPI的采样时刻不是由上升/下降沿决定的。



①**SPI模式及时序**

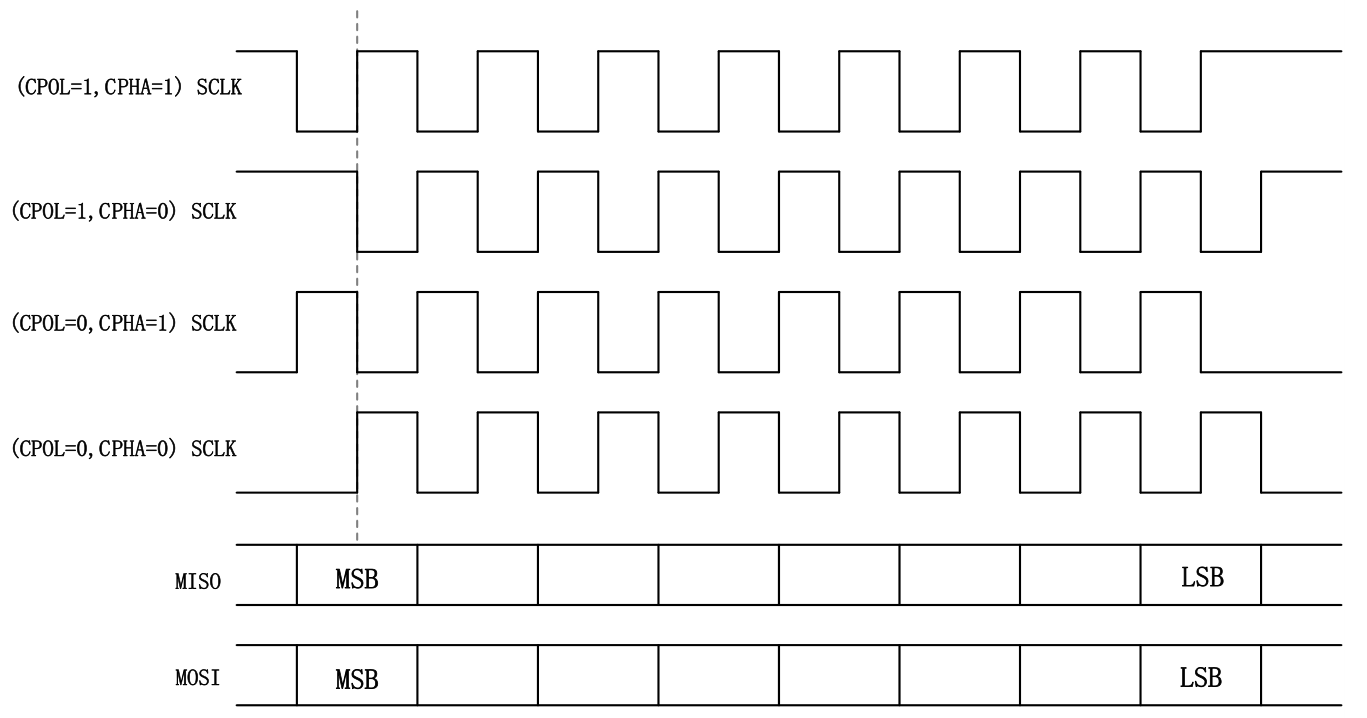
**CPOL**：时钟极性，指SPI通讯设备处于空闲状态时，SCK信号线的电平信号(即通讯开始前CS线为高电平时SCLK

的状态)。CPOL=0时， SCLK在空闲状态时为低电平，CPOL=1时，SCLK在空闲状态时为高电平。

**CPHA**：时钟相位，指数据的采样时刻。当CPHA=0时，MOSI或MISO数据线上的信号将会在 SCK时钟线的“奇数

边沿”被采样。当CPHA=1 时，数据线在SCK的“偶数边沿”采样。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SPI模式** | **CPOL** | **CPHA** | **空闲时SCLK状态** | **采样时刻** |
| **0** | 0 | 0 | 低电平 | 奇数边沿 |
| **1** | 0 | 1 | 低电平 | 偶数边沿 |
| **2** | 1 | 0 | 高电平 | 奇数边沿 |
| **3** | 1 | 1 | 高电平 | 偶数边沿 |



**SPI 四种工作模式**

SPI比I2C时序简单，不分读时序和写时序，因为SPI是全双工的，读写时序可以一起进行。CS片选线拉低后，即选

中对应的从设备，通过MOSI和MISO两根数据线进行收发数据。如下图所示，主机向从机发送0XD2，从机返回0X66。

图片包含 图表

描述已自动生成

**SPI模式0时序图**

②**ECSPI外设**（见参考手册806页）

I.MX6U 有4个 ECSPI外设，如果使用ECSPI的硬件片选信号，则只能连接4个SPI外设，且片选信号只能使用指定的片选IO；如果使用软件片选，则可使用任意IO作为片选信号线，且可连接任意数量外设。教程中使用软件片选，所以下面对片选信号的设置位可以忽略，下面介绍相关寄存器。

1）**数据寄存器**

ECSPIx\_TXDATA 和 ECSPIx\_RXDATA为数据寄存器，要发送数据就向寄存器 ECSPIx\_TXDATA 写入数据，接收数据时直接读取寄存器ECSPIx\_RXDATA里面的数据就可以了。数据写入寄存器后，随后会被压入FIFO中，再通过配置相应寄存器将FIFO中的数据进行传输。ECSPI 有 64\*32 个接收FIFO(RXFIFO)和 64\*32 个发送 FIFO(TXFIFO)。

2）**控制寄存器**

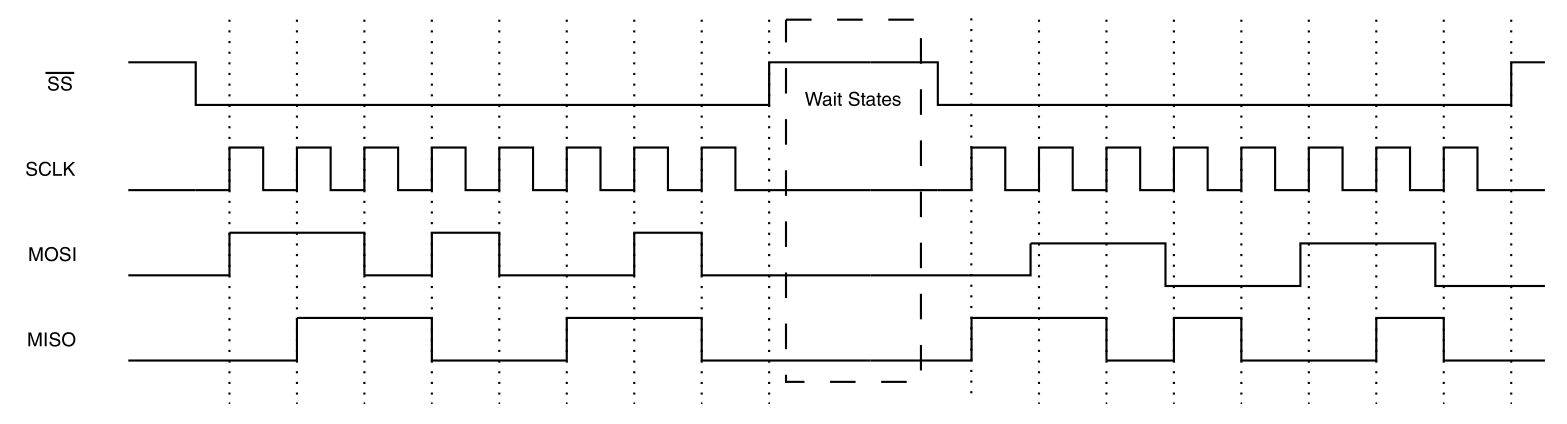
ECSPIx\_CONREG为控制寄存器，EN(bit0)位为使能位，为1时使能SPI，为0时关闭SPI；SMC(bit3)位为开始模式控制位，该位为1时，只要向TXTFIFO写入数据就开始数据传输，为0时则通过XCH位控制数据传输，我们设置为1；CHANNEL\_MODE(bit4~7)位为SPI通道主从模式控制位，分别对应SPI通道0~3，为1时设置为主模式，为0时设置为从模式，将该位设置为0X01，即设置通道0为主模式；设置PRE\_DIVIDER（bit12~15）位和POST\_DIVIDER（bit8~11）位对**SPI CLK**进行两级分频，分频后得到的时钟即为SPI外设时钟；CHANNEL\_SELECT(bit19:18)位为通道选择位，1个 ECSPI 有4个硬件片选信号，每个片选信号是1个硬件通道，本节实验使用的软件片选，使用软件片选时该位可不设置；BURST\_LENGTH(bit31:20)位为发送数据长度控制位，即1次数据发送中字节的长度，设置为7，即每次传1字节数据；

3）**配置寄存器**

ECSPIx\_CONFIGREG为配置寄存器，SCLK\_PHA(bit0~3)位设置通道0~3的CPHA（时钟相位），为1时第2个时钟跳变沿采集数据，为0时第1个时钟跳变沿采集数据；SCLK\_POL(bit7:4)位设置CPOL（时钟极性），为1时SCLK空闲时为高电平，为0时SCLK空闲时为低电平；SS\_POL(bit12~15)位设置通道0~3片选线极性，为1时高电平有效，为0时低电平有效。DATA\_CTL(bit16~19)位设置通道0~3的数据信号线空闲时电平状态，为1时为低电平，为0时为高电平；SCLK\_CTL(bit20~23)位设置通道0~3的时钟信号线空闲时电平状态，为1时为高电平，为0时为低电平；

4）**采样周期寄存器**

Wait states（等待状态）被插入在两次SPI数据传输之间，目的是为了通过软件的方式去降低SPI的传输速度，以满足那些SPI传输速度较慢的设备。示意图如下，这个采样周期寄存器就是对Wait states进行设置。



**带有等待状态的SPI传输**

ECSPIx\_PERIODREG（bit0~14）为采样周期寄存器，SAMPLE\_PERIOD（bit0~14）位设置两次数据传输间插入的Wait states的数量，设置为0x2000（参考官方例程）；CSRC（bit15）位设置SPI的时钟源，为1时选择32.768KHz晶振作为时钟源，为0时选择SPI CLK为时钟源，见**SPI CLK设置**；CSD\_CTL(bit16~21) 位为片选信号延时控制位，用于设置片选信号和第一个SPI时钟之间的时间间隔，范围为0~63。

5）**状态寄存器**

ECSPIx\_STATREG为状态寄存器，TE(bit0)位为TXFIFO标志位，为1时表示TXFIFO为空，为0时表示TXFIFO中至少有1字（32bit）数据；TF(bit2)位为TXFIFO满标志位，为1时表示TXFIFO已满，为0时表示TXFIFO没满；RR(bit3)位为RXFIFO标志位，为1时表示RXFIFO中至少有1个字（32bit）数据，为0时表示RXFIFO为空；RF(bit5)位为RXFIFO空标志位，为1时表示RXFIFO为空，为0时表示RXFIFO不为空；RO(bit6)位为RXFIFO溢出标志位，为1时表示RXFIFO溢出，为0时表示RXFIFO未溢出；TC(bit7)位为传输完成标志位，为1时表示传输完成，为0时表示正在传输。发送数据时等待TE(bit0)位为1；接收数据时等待RR(bit3)位为1。

6）**SPI CLK设置**（见参考手册682页）

参考[时钟树](#时钟树图)，配置寄存器CSCDR2的ECSPI\_CLK\_SEL位为0，将PPL3（480MHz）作为SPI时钟源，后接1个8分频，再配置寄存器CSCDR2的ECSPI\_CLK\_PODF位为0，即1分频，得到SPI CLK = 480/8/1=60MHz。本节使用ICM-20608 作为SPI从设备，ICM-20608最高支持SPI传输速度为8MHz，因此我们将SPI时钟频率设置为6MHz，配置寄存器ECSPIx\_CONREG的PRE\_DIVIDER（bit12~15）位为0X9，即10分频，配置寄存器ECSPIx\_CONREG的POST\_DIVIDER（bit8~11）位为0X0，即1分频，得到最后的时钟频率：SPI CLK = 60000000/10 = 6MHz。

③**ICM-20608** （使用时首先要退出休眠模式）

ICM-20608 是 InvenSense 出品的一款 6 轴 MEMS 传感器，包括 3 轴加速度和 3 轴陀螺仪。ICM-20608 尺寸非常小，只有3×3×0.75mm，采用16P的LGA封装。ICM-20608内部有一个 512字节的 FIFO。陀螺仪的量程范围可以编程设置，可选择±250，±500，±1000 和±2000º/s，加速度的量程范围也可以编程设置，可选择±2g，±4g，±8g 和±16g。陀螺仪和加速度计都是 16 位的 ADC，并且支持 I2C 和 SPI 两种协议，使用 I2C 接口的话通信速度最高可以达到400KHz，使用 SPI 接口的话通信速度最高可达到 8MHz。使用I2C接口时，ICM-20608 的 AD0脚决定从机地址的最后1位，为低电平时地址为0X68，为高电平时地址为0X69。

SPI通讯时需要至少16个时钟，发送的第1个字节中，高1位为读写位，为1时表示读，为0时表示写，低7位为从机地址；第2个字节为读写的数据，通过读写ICM-20608相应寄存器进行相应控制，寄存器为8位，介绍如下。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **寄存器** | **寄存器名称** | **位名称** | **寄存器功能** | **描述** |
| **0X19** | SMPLRT\_DIV | SMPLRT\_DIV[7:0] | 输出速率设置 | 公式：SAMPLE\_RATE=INTERNAL\_SAMPLE\_RATE/(1 + SMPLRT\_DIV) |
| **0X1A** | CONFIG | DLPF\_CFG[2:0] | 芯片配置 | 设置陀螺仪低通滤波。可设置 0~7 |
| **0X1B** | GYRO\_CONFIG | FS\_SEL[4:3] | 陀螺仪量程设置 | 00：±250dps 01：±500dps 10：±1000dps 11：±2000dps |
| **0X1C** | ACCEL\_CONFIG | ACCEL\_FS\_SEL[4:3] | 加速度计量程设置 | 00：±2g 01：±4g 10：±8g 11：±16g |
| **0X1D** | ACCEL\_CONFIG 2 | A\_DLPF\_CFG[2:0] | 加速度计低通滤波设置 | 设置加速度计的低通滤波，可设置 0~7 |
| **0X1E** | LP\_MODE\_CFG | GYRO\_CYCLE[7] | 陀螺仪低功耗使能 | 0：关闭陀螺仪的低功耗功能 1：使能陀螺仪的低功耗功能 |
| **0X23** | FIFO\_EN | TEMP\_FIFO\_EN[7] | FIFO 使能控制 | 0：关闭温度传感器 FIFO 1：使能温度传感器 FIFO |
| XG\_FIFO\_EN[6] | 0：关闭陀螺仪 X 轴 FIFO 1：使能陀螺仪 X 轴 FIFO |
| YG\_FIFO\_EN[5] | 0：关闭陀螺仪 Y 轴 FIFO 1：使能陀螺仪 Y 轴 FIFO |
| ZG\_FIFO\_EN[4] | 0：关闭陀螺仪 Z 轴 FIFO 1：使能陀螺仪 Z 轴 FIFO |
| ACCEL\_FIFO\_EN[3] | 0：关闭加速度计 FIFO 1：使能加速度计 FIFO |
| **0X3B** | ACCEL\_XOUT\_H | ACCEL\_XOUT\_H[7:0] | 数据寄存器 | 加速度 X 轴数据高 8 位 |
| **0X3C** | ACCEL\_XOUT\_L | ACCEL\_XOUT\_L[7:0] | 加速度 X 轴数据低 8 位 |
| **0X3D** | ACCEL\_YOUT\_H | ACCEL\_YOUT\_H[7:0] | 加速度 Y 轴数据高 8 位 |
| **0X3E** | ACCEL\_YOUT\_L | ACCEL\_YOUT\_L[7:0] | 加速度 Y 轴数据低 8 位 |
| **0X3F** | ACCEL\_ZOUT\_H | ACCEL\_ZOUT\_H[7:0] | 加速度 Z 轴数据高 8 位 |
| **0X40** | ACCEL\_ZOUT\_L | ACCEL\_ZOUT\_L[7:0] | 加速度 Z 轴数据低 8 位 |
| **0X41** | TEMP\_OUT\_H | TEMP\_OUT\_H[7:0] | 温度数据高 8 位 |
| **0X42** | TEMP\_OUT\_L | TEMP\_OUT\_L[7:0] | 温度数据低 8 位 |
| **0X43** | GYRO\_XOUT\_H | GYRO\_XOUT\_H[7:0] | 陀螺仪 X 轴数据高 8 位 |
| **0X44** | GYRO\_XOUT\_L | GYRO\_XOUT\_L[7:0] | 陀螺仪 X 轴数据低 8 位 |
| **0X45** | GYRO\_YOUT\_H | GYRO\_YOUT\_H[7:0] | 陀螺仪 Y 轴数据高 8 位 |
| **0X46** | GYRO\_YOUT\_L | GYRO\_YOUT\_L[7:0] | 陀螺仪 Y 轴数据低 8 位 |
| **0X47** | GYRO\_ZOUT\_H | GYRO\_ZOUT\_H[7:0] | 陀螺仪 Z 轴数据高 8 位 |
| **0X48** | GYRO\_ZOUT\_L | GYRO\_ZOUT\_L[7:0] | 陀螺仪 Z 轴数据低 8 位 |
| **0X6B** | PWR\_MGMT\_1 | DEVICE\_RESET[7] | 电源管理寄存器 1 | 1：复位 ICM-20608 |
| SLEEP[6] | 0：退出休眠模式 1：进入休眠模式 |
| **0X6C** | PWR\_MGMT\_2 | STBY\_XA[5] | 电源管理寄存器 2 | 0：使能加速度计 X 轴 1：关闭加速度计 X 轴 |
| STBY\_YA[4] | 0：使能加速度计 Y 轴 1：关闭加速度计 Y 轴 |
| STBY\_ZA[3] | 0：使能加速度计 Z 轴 1：关闭加速度计 Z 轴 |
| STBY\_XG[2] | 0：使能陀螺仪 X 轴 1：关闭陀螺仪 X 轴 |
| STBY\_YG[1] | 0：使能陀螺仪 Y 轴 1：关闭陀螺仪 Y 轴 |
| STBY\_ZG[0] | 0：使能陀螺仪 Z 轴 1：关闭陀螺仪 Z 轴 |
| **0X75** | WHO AM I | WHOAMI[7:0] | ID 寄存器 | ICM-20608G 的 ID 为 0XAF ICM-20608D 的 ID 为 0XAE |

ICM20608使用16位保存传感器数值，因此根据不同量程可得到不同的测量因子，实际值计算转换公式如下。

转换公式：实际值 = 寄存器值 / 测量因子

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **加速度计** | | **陀螺仪** | | **温度计** | | |
| **量程（º/s）** | **测量因子LSB/（º/s）** | **量程** | **测量因子(LSB/g)** | **量程（**°C**）** | RoomTemp\_Offset | Temp\_Sensitivity |
| ±250 | 131 | ±2 | 16,384 | -40 ~ +85 | 25 | 326.8 |
| ±500 | 65.5 | ±4 | 8,192 | **计算公式** | | |
| ±1000 | 32.8 | ±8 | 4,096 | Tem = ((TEMP\_OUT/RoomTemp\_Offset)/Temp\_Sensitivity) + 25 | | |
| ±2000 | 16.4 | ±16 | 2,048 |

④**开启I.MX6U的NEON和VFP**

1）**编写函数开启NEON和VFP(VFPV4\_D32)**

当使用I.MX6U进行浮点运算时，需要开启NEON和FPU(VFPV4\_D32)，代码如下。

  void imx6ul\_hardfpu\_enable(void)

{

    uint32\_t cpacr;

    uint32\_t fpexc;

    cpacr = \_\_get\_CPACR(); //获取CPACR寄存器中的内容

    cpacr = (cpacr & ~(CPACR\_ASEDIS\_Msk | CPACR\_D32DIS\_Msk)) //见注①

 |  (3UL << CPACR\_cp10\_Pos) | (3UL << CPACR\_cp11\_Pos);

    \_\_set\_CPACR(cpacr); //配置CPACR寄存器

    fpexc = \_\_get\_FPEXC(); //获取FPEXC寄存器中的内容

    fpexc |= 0x40000000UL; //见注②

    \_\_set\_FPEXC(fpexc);   //配置FPEXC寄存器

}

注：①CPACR为协处理器访问控制寄存器，用于控制对协处理器cp10和cp11的访问，配置信息如下。

**cp10**(bit20~21)：设置cp10。 00：拒绝访问 01：PL1或者更高时可访问 10：保留 11：完全访问

**cp11**(bit22~23)：设置cp11。00：拒绝访问 01：PL1或者更高时可访问 10：保留 11：完全访问

**D32DIS**(bit30)：控制D16~D31寄存器的使用。0：D0~D31寄存器均可正常访问；1：VFP无法访问D16~D31。

当FPU和SIMD都实现时，该位必须为0；当FPU实现但SIMD未实现时，该位时RAO/WI；当FPU和SIMD均未实现时，该位是UNK/SBZP。

**ASEDIS**(bit31)；控制高级SIMD功能。0：所有高级SIMD和VFP指令正常执行；1：所有的SIMD指令异常。

当FPU实现但SIMD未实现时，该位是RAO/WI；当FPU和SIMD都没实现时，该位是UNK/SBZP。

②FPEXC为浮点异常控制寄存器，为高级SIMD（NEON）和浮点(VFP)扩展提供全局启用，并指示如何记录这些扩展

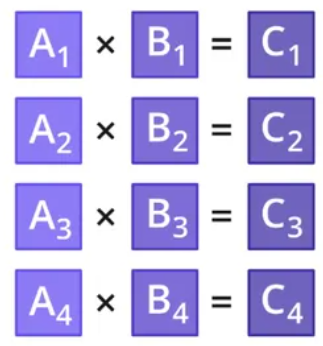
的状态，配置信息如下。

**EN**(bit30)：高级SIMD和浮点拓展的使能位。0：禁用高级SIMD和浮点拓展； 1：使能高级SIMD和浮点拓展

综上，我们要使能NEON和VFP，需将寄存器CPACR的cp10、cp11设置为11，将D32DIS、ASEDIS设置为0；将寄存器FPEXC的EN设置为1。下面介绍相关知识点。（CPACR寄存器具体信息见《Cortex-A7 MPCore》4-62页）（具体信息见《ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition》B4-1568页）

a. **SIMD**：一些现代软件，尤其是多媒体编解码软件和图形加速软件，有大量的少于机器字长的数据参与运算。例如，

在音频应用中16位以内数据是频繁的，在图形与视频领域8位以内数据是频繁的。当在32位微处理器上执行这些操作时，相当一部分计算单元没有被利用，但是依然消耗着计算资源。为了更好的利用这部分闲置的资源，SIMD技术使用一个单指令来并行地在同样类型和大小的多个数据元素上执行相同的操作。通过这种方法，硬件可以在同样时间消耗内用并行的4个8位数值加法运算来替代通常的两个32位数值加法运算。SIMD 是现代处理器（包括 CPU 和 GPU）中用于加速向量和矩阵操作的常见技术，示意图如下。

 图形用户界面, 应用程序, Teams

描述已自动生成

**标量运算** **SIMD运算**

c. **NEON**：用在ARM处理器上的高级SIMD扩展的实现称为NEON，这是架构规范之外使用的通用术语。NEON技术

在当前所有ARM Cortex-A系列处理器上得到了实现，旨在为消费性多媒体应用程序提供灵活、强大的加速功能。

NEON技术只适用于ARM Cortex-A系列处理器。ARM架构定义高级SIMD扩展作为协处理器cp10和cp11的一部分，cp10和cp11也同时用于向量浮点扩展(VFP)。NEON 寄存器组由 32 个 64 位寄存器组成， 可看作16个128位四字长寄存器组Q0~Q15或32个64位双字长寄存器组D0~D31，每一个Q0~Q15寄存器都映射到一对D寄存器上。NEON D0~D31寄存器和VFPv3 D0~D31寄存器是一样的，即NEON和VFPv3共享同一个寄存器组。

b. **FPU**：浮点运算单元，处理器内部用于执行浮点数计算的逻辑部件，或者说硬件电路。不是所有的处理器都有该功

能。对于含有FPU的处理器，我们可以通过控制编译选项的方式使编译器编译出的代码使用FPU指令。

d. **VFP**：在ARM中的FPU被称为VFP，即向量浮点运算，需要两个协处理器CP10和CP11来实现，其中CP10支持

单精度浮点操作，CP11支持双精度浮点操作。VFP除了提供浮点数基本运算（加、减、乘、除、开方、比较、取反）提供支持之外，最有特点是它的向量（vectors）功能。它同时支持最多8组单精度4组双精度浮点数的运算。

e. **PL**：PL（PowerLimit）用来限制CPU的功耗，笔记本上对CPU有两道枷锁，PL1与PL2。PL1叫做长时睿频功耗，

CPU在长时间负载时，基本最后都会停留到这个功耗；PL2叫做短时睿频功耗，通常会高过PL1许多，是在短时间负载内CPU能够达到的最高功耗，也就是一台笔记本的CPU性能上限。NEON使用32个双精度浮点寄存器，即32D，所以此为要配置为0。此位要配置为0。

f. **ARM词汇含义**

**RAZ**：Read-As-Zero 读到的总是0 **RAO**：Read-As-Oned读到的总是1

**WI**：Writes Ignoredx 写无效 **UNK**：unknown 读写时返回值未知

**SBZP**：Should-Be-Zero-or-Preserved on writes 写总是0 **SBOP**： Should-Be-One-or-Preserved on writes. 写总是1

综上，NEON经常和VFP一起出现，他们之所以耦合在一起是因为他们公用浮点寄存器。虽然架构层面并不要求VFP和NEON同时实现，但是鉴于这些扩展在编程模型层面的共同特征，一个支持VFP的操作系统仅需很少甚至无需修改即可支持NEON。

2）**修改MakeFile**

上面通过编写函数imx6ul\_hardfpu\_enable使能了MEON和VFP，在编译运行时，还需要对MakeFile进行修改，指定GCC编译器在编译时使用硬件浮点计算（VFP）。Cortex-A7支持NEON技术，浮点单元（FPU），支持VFPv4-D32 架构，修改位置见[通用MakeFile](#MakeFile编译)，修改如下。（具体信息见参考手册387页）

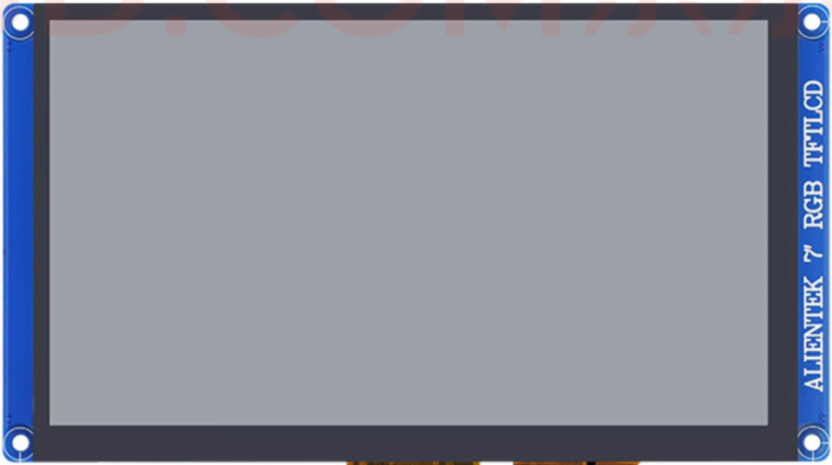
$(COBJS) : obj/%.o : %.c

$(CC) [-Wall](#MakeFile中Wall的解释) -march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=hard [-Wa, -mimplicit-it=thumb](#移植printf后修改MakeFile) [-nostdlib](#MakeFile中Wall的解释) [-fno-builtin](#MakeFile中设置不适用内建函数) [-c -O2](#GCC指令) $(INCLUDE) -o $@ $<

-march：指定内核架构 -mfpu：指定为NEON+VFP结构 -mfloat-abi：指定使用VFP和NEON指令

1. **电容触摸屏**

电容触摸屏是人机交互开发的常用设备，电阻屏需要手指给予一定的压力才有反应，而电容屏只需手指轻触就可以了，且电容屏不需要校准。电容触摸屏内包含驱动IC，一般提供I2C接口，主控制器可通过I2C接口读取驱动IC内的触摸坐标数据，进而对触摸屏进行控制。正点原子7寸电容屏由两部分组成：TFT LED、触摸屏，底下是 LCD 面板，上面是触摸面板，将两个封装到一起就成了带有触摸屏的 LCD 屏幕。



驱动IC型号为FT5426，此IC采用15\*28的驱动结构，即 15 个感应通道，28 个驱动通道，最多支持 5 点电容触摸。触摸屏部分有 4 个 IO 用于连接主控制器：SCL、SDA、RST 和 INT，一般通过 INT 引脚来通知主控制器有触摸点按下，然后在 INT 中断服务函数中读取触摸数据。也可以不使用中断功能，采用轮询的方式不断查询是否有触摸点按下，下面介绍FT5426中常用的寄存器。

|  |  |  |  |
| --- | --- | --- | --- |
| **寄存器** | **位** | **功能** | **描述** |
| **0X00** | [6:4] | 设置工作模式 | 000：正常模式 001：系统信息模式 100：测试模式 |
| **0X02** | [3:0] | 记录触摸状态 | 记录有多少个触摸点，有效值为 1~5 |
| **0X03** | [7:6] | 第一个触摸点X坐标高位数据 | 事件标志：00：按下 01：抬起 10：接触 11：保留 |
| [3:0] | X 轴坐标值高 4 位 |
| **0X04** | [7:0] | 第一个触摸点 X 坐标低位数据 | X 轴坐标值低 8 位 |
| **0X05** | [7:4] | 第一个触摸点 Y 坐标高位数据 | 触摸点的 ID |
| [3:0] | Y 轴坐标高 4 位 |
| **0X06** | [7:0] | 第一个触摸点 Y 坐标低位数据 | Y 轴坐标低 8 位 |
| **0X09** | [7:6] | 第二个触摸点X坐标高位数据 | 事件标志：00：按下 01：抬起 10：接触 11：保留 |
| [3:0] | X 轴坐标值高 4 位 |
| **0X0A** | [7:0] | 第二个触摸点 X 坐标低位数据 | X 轴坐标值低 8 位 |
| **0X0B** | [7:4] | 第二个触摸点 Y 坐标高位数据 | 触摸点的 ID |
| [3:0] | Y 轴坐标高 4 位 |
| **0X0C** | [7:0] | 第二个触摸点 Y 坐标低位数据 | Y 轴坐标低 8 位 |
| **0X0F** | [7:6] | 第三个触摸点X坐标高位数据 | 事件标志：00：按下 01：抬起 10：接触 11：保留 |
| [3:0] | X 轴坐标值高 4 位 |
| **0X10** | [7:0] | 第三个触摸点 X 坐标低位数据 | X 轴坐标值低 8 位 |
| **0X11** | [7:4] | 第三个触摸点 Y 坐标高位数据 | 触摸点的 ID |
| [3:0] | Y 轴坐标高 4 位 |
| **0X12** | [7:0] | 第三个触摸点 Y 坐标低位数据 | Y 轴坐标低 8 位 |
| **0X15** | [7:6] | 第四个触摸点X坐标高位数据 | 事件标志：00：按下 01：抬起 10：接触 11：保留 |
| [3:0] | X 轴坐标值高 4 位 |
| **0X16** | [7:0] | 第四个触摸点 X 坐标低位数据 | X 轴坐标值低 8 位 |
| **0X17** | [7:4] | 第四个触摸点 Y 坐标高位数据 | 触摸点的 ID |
| [3:0] | Y 轴坐标高 4 位 |
| **0X18** | [7:0] | 第四个触摸点 Y 坐标低位数据 | Y 轴坐标低 8 位 |
| **0X1B** | [7:6] | 第五个触摸点X坐标高位数据 | 事件标志：00：按下 01：抬起 10：接触 11：保留 |
| [3:0] | X 轴坐标值高 4 位 |
| **0X1C** | [7:0] | 第五个触摸点 X 坐标低位数据 | X 轴坐标值低 8 位 |
| **0X1D** | [7:4] | 第五个触摸点 Y 坐标高位数据 | 触摸点的 ID |
| [3:0] | Y 轴坐标高 4 位 |
| **0X1E** | [7:0] | 第五个触摸点 Y 坐标低位数据 | Y 轴坐标低 8 位 |
| **0XA1** | [7:0] | 版本寄存器 | 固件版本号高字节 |
| **0XA2** | [7:0] | 固件版本号低字节 |
| **0XA4** | [7:0] | 设置中断模式 | 0：轮询模式 1：触发模式 |
| **从机地址：0X38** | | | |

1. **IO属性配置**

I.MX6ULL由五组GPIO：GPIO1(32个IO)、GPIO2(22个IO)、GPIO3(29个IO)、GPIO4(29个IO)、GPIO5(12个IO)。

一共有124个IO，I.MX6ULL的IO是根据IO所拥有的功能进行命名的。GPIO 模块是每个 IO 都具有的外设，它具有 IO 控制最基本的功能，如输出高低电平、检测电平输入等。同时，这些GPIO可配置相应寄存器复用为其他外设的引脚。下面主要介绍IO属性配置。（具体复用信息可查看参考手册第191页）（IO相关寄存器见参考手册1495页）

图示

描述已自动生成

**GPIO 功能图**

|  |  |  |
| --- | --- | --- |
| **位** | **功能** | **描述** |
| **SRE[0]** | 压摆率设置位 | 0：低压摆率 1：高压摆率 |
| **DSE[3:5]** | IO驱动能力设置位 | 000：输出驱动关闭 001：R0 010：R0/2 011：R0/3  100：R0/4 101：R0/5 110：R0/6 111：R0/7 |
| **SPEED[6:7]** | IO输出速度设置位 | 00：50MMHz 01：100MHz 10:100MHz 11：200MHz |
| **ODE[11]** | 开路输出使能位 | 0：禁止开路输出 1：使能开路输出 |
| **PKE[12]** | 状态保持器/上下拉使能位 | 0：禁止状态保持器和上下拉 1：使能状态保持器和上下拉 |
| **PUE[13]** | 状态保持器/上下拉选择位 | 0：使用状态保持器 1：使用上下拉 |
| **PUS[14:15]** | 上下拉电阻设置位 | 00：100K下拉 01：47K 上拉 10：100K 上拉 11：22K 上拉 |
| **HYS[16]** | 施密特触发器使能位 | 0：禁止施密特触发器 1 ：使能施密特触发器 |

1）压摆率就是 IO 电平跳变所需要的时间，所需时间越短压摆率越高；跳变时间越长波形就越缓，压摆率就越低。

2）状态保持器在IO作为输入时有效，作用是外部电路断电后IO可保持以前的状态。

3）施密特触发器在IO 作为输入功能时有效，需要对输入波形进行整形时可以使能HYS[16]位。

上面是对IO属性进行配置的相关寄存器信息，注意是对IO进行配置，不是GPIO。GPIO只是IO众多复用功能中的一种。下面介绍对GPIO属性进行配置的相关寄存器信息。

|  |  |  |
| --- | --- | --- |
| **寄存器** | **名称** | **描述** |
| DR | 数据寄存器 | 保存对应 IO 的电平值。如GPIO1有32个IO，则bit0~31保存GPIO1\_IO00~ IO31的电平值。 |
| GDIR | 方向寄存器 | 设置某个 GPIO 的工作方向。0：输入 1：输出 |
| PSR | 状态寄存器 | 获取某个GPIO的高低电平值，也是一个GPIO对应一个位。 |
| ICR1 | 中断寄存器 | 设置IO0~15的中断方式。00：低电平触发 01：高电平触发 10：上升沿触发 11：下降沿触发 |
| ICR2 | 设置IO16~31的中断方式。00：低电平触发 01：高电平触发 10：上升沿触发 11：下降沿触发 |
| IMR | 中断使能 | 中断使能寄存器，也是一个GPIO对应一个位。1：使能中断 0：禁用中断 |
| ISR | 中断状态 | 中断状态寄存器， GPIO发生中断时，对应位置1。注意，写1清除标志位，不是写0。 |
| EDGE\_SEL | 边沿选择 | 设置边沿中断，该寄存器会覆盖 ICR1 和 ICR2 的设置， 为1时对应GPIO双边沿触发中断。 |

通过汇编语言、C语言配置相应寄存器即可对相应IO进行设置，教程中使用的是NXP提供的SDK包中的IO配置函数，下面介绍两个IO配置函数： IOMUXC\_SetPinMux（IO复用函数）和IOMUXC\_SetPinConfig（IO属性配置函数）。

①IO复用函数

static inline void IOMUXC\_SetPinMux(uint32\_t muxRegister, //IO复用寄存器地址

uint32\_t muxMode, //IO复用值，即ALT0~ALT8

uint32\_t inputRegister, // 外设输入 IO 选择寄存器地址

uint32\_t inputDaisy, //寄存器 inputRegister 的值

uint32\_t configRegister, //未使用，函数 IOMUXC\_SetPinConfig会使用

uint32\_t inputOnfield) // IO软件输入使能

{

  \*((volatile uint32\_t \*)muxRegister) = IOMUXC\_SW\_MUX\_CTL\_PAD\_MUX\_MODE(muxMode) |

IOMUXC\_SW\_MUX\_CTL\_PAD\_SION(inputOnfield);

    if (inputRegister)

    {

        \*((volatile uint32\_t \*)inputRegister) = IOMUXC\_SELECT\_INPUT\_DAISY(inputDaisy);

    }

}

使用此函数时，示例如下。（将 GPIO1\_IO03 的复用功能设置为GPIO）

IOMUXC\_SetPinMux(IOMUXC\_GPIO1\_IO03\_GPIO1\_IO03, 0);

IOMUXC\_SetPinMux 函数含有6个参数，SDK包中根据IO复用的不同功能，已经将前5个参数配置好了，且定义为了指定的宏，最后一个参数inputOnfield设置的是IO强制输入， 1：强制输入； 0：输入由复用的功能决定。所以上述函数就是将GPIO1\_IO03复用为了GPIO1\_IO03，且工作方向由功能决定。IO复用设置完成后还需要对IO属性进行配置，IO属性配置函数介绍如下。

static inline void IOMUXC\_SetPinConfig(uint32\_t muxRegister,

                                       uint32\_t muxMode,

                                       uint32\_t inputRegister,

                                    uint32\_t inputDaisy,

                                        uint32\_t configRegister, // IO 配置寄存器地址

                                        uint32\_t configValue) //写入到寄存器 configRegister 的值

{

    if (configRegister)

    {

        \*((volatile uint32\_t \*)configRegister) = configValue;

    }

}

如上所示，此函数只用到了configRegister 和 configValue两个参数，IOMUXC\_SW\_PAD\_CTL\_PAD\_XX寄存器是对IO进行属性配置的，configRegister就是寄存器地址，configValue是写入到寄存器 configRegister 的值。同样，SDK包中将前5个参数进行打包，定义为了指定的宏，最后一个参数就需要根据IO具体复用的功能进行设置。下面介绍常见的configValue值。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **configValue** | **HYS[16]** | | **PUS[15:14]** | **PUE[13]** | | | **PKE[12]** | **ODE[11]** | | **SPEED[7:6]** | **DSE[5:3]** | | **SRE[0]** |
| **0X10B0** | 0 | | 00 | 0 | | | 1 | 0 | | 10 | 110 | | 0 |
| **0X70B0** | 0 | | 01 | 1 | | | 1 | 0 | | 10 | 110 | | 0 |
| **0X10B1** | 0 | | 00 | 0 | | | 1 | 0 | | 10 | 110 | | 1 |
| **0XF080** | 0 | | 11 | 1 | | | 1 | 0 | | 10 | 000 | | 0 |
| **0XB090** | 0 | | 10 | 1 | | | 1 | 0 | | 10 | 010 | | 0 |
| **0X10B0**：普通GPIO | | **0X70B0**：I2C从设备 | | | | **0X10B1**：SPI从设备 | | | **0XF080**：LCD屏幕 | | | **0XB090**：PWM输出 | |
| **位** | **功能** | | | | **描述** | | | | | | | | |
| **SRE[0]** | 压摆率设置位 | | | | 0：低压摆率 1：高压摆率 | | | | | | | | |
| **DSE[3:5]** | IO驱动能力设置位 | | | | 000：输出驱动关闭 001：R0 010：R0/2 011：R0/3  100：R0/4 101：R0/5 110：R0/6 111：R0/7 | | | | | | | | |
| **SPEED[6:7]** | IO输出速度设置位 | | | | 00：50MMHz 01：100MHz 10:100MHz 11：200MHz | | | | | | | | |
| **ODE[11]** | 开路输出使能位 | | | | 0：禁止开路输出 1：使能开路输出 | | | | | | | | |
| **PKE[12]** | 状态保持器/上下拉使能位 | | | | 0：禁止状态保持器和上下拉 1：使能状态保持器和上下拉 | | | | | | | | |
| **PUE[13]** | 状态保持器/上下拉选择位 | | | | 0：使用状态保持器 1：使用上下拉 | | | | | | | | |
| **PUS[14:15]** | 上下拉电阻设置位 | | | | 00：100K下拉 01：47K 上拉 10：100K 上拉 11：22K 上拉 | | | | | | | | |
| **HYS[16]** | 施密特触发器使能位 | | | | 0：禁止施密特触发器 1 ：使能施密特触发器 | | | | | | | | |

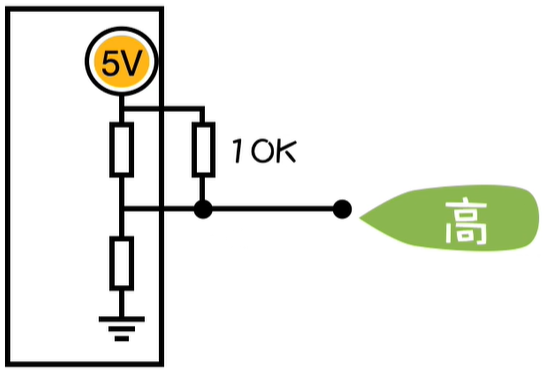
1. **上下拉电阻**

在进行IO配置时，经常看到上拉电阻将IO上拉至高电平，或下拉电阻将IO下拉至低电平；还有人说上下拉电阻能够

增强驱动能力，电阻不是负载么，怎么会增强驱动能力呢？通过查阅相关资料，上拉电阻和下拉电阻主要作用为：钳位、提高驱动能力，具体介绍这两个作用，还有其他功能，这里没有全部总结。

图示

低可信度描述已自动生成 图示

描述已自动生成 

**IO内部示意图 开漏输出 接上拉电阻**

**钳位**：开漏输出模式下，避免IO状态不确定。如IO内部示意图，上面MOS管导通，下面MOS管截止，此时IO输出即为

高电平；上面MOS管截止，下面MOS管导通，此时IO输出即为低电平。当上、下NOS管均截止时，IO处于悬空状态，此时需要加上拉或下拉电阻，以免受到随机电平的影响。若此时在电源与IO之间接一个10K的上拉电阻，相当于上拉电阻与内部的无穷大的电阻进行并联，此时电源与IO之间的电阻约等于10K，远小于地与IO之间的电阻值（无穷大），则此时IO被上拉电阻上拉至高电平。同理，当上、下MOS管均截止时，若在IO与地间接一个下拉电阻，与电源与IO间的阻值（无穷大）相比相当于导线，即可将IO下拉至低电平。

**提高驱动能力：**上拉加强拉电流能力，下拉加强吸电流能力。单片机输出高电平驱动力有限，由于后续电路的影响，IO可

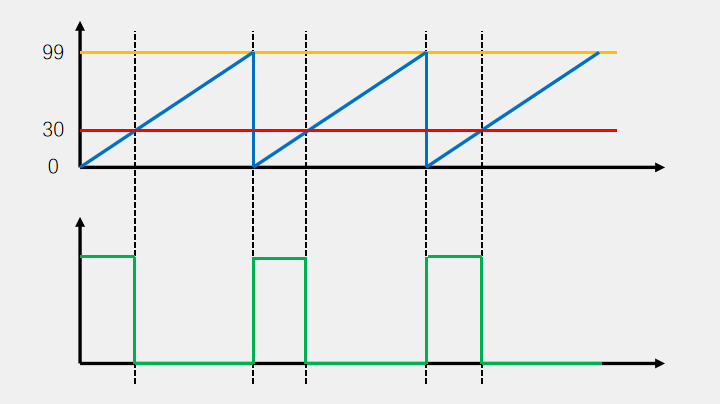
能达不到VCC和GND。外接上下拉电阻相当于与芯片内部电阻进行并联，所以IO输出高电平时，外接一个上拉电阻，则电源与IO间电阻减小，输出高电平的驱动能力也就增强了。注意，上面MOS管导通时，内部也是有电阻的，如MOS管本身就有阻值，电源内部也有阻值。下拉电阻同理， IO输出低电平时，外接一个下拉电阻也会减小IO与地之间的阻值，即增强吸电流的能力。

**强/弱上、下拉**：上拉电阻越小，电源与IO间的电阻就越小，驱动能力就越强，常见上拉电阻阻值为1k、4k7或10k。阻值

太小会导致电流过大，阻值太大会导致抗干扰能力变弱。下拉电阻越小，流过下拉电阻的电流也就越大，即损失的功耗就越大，所以下拉电阻阻值应尽可能的大，常见的下拉电阻阻值为200K，1M等。

1. **PWM**

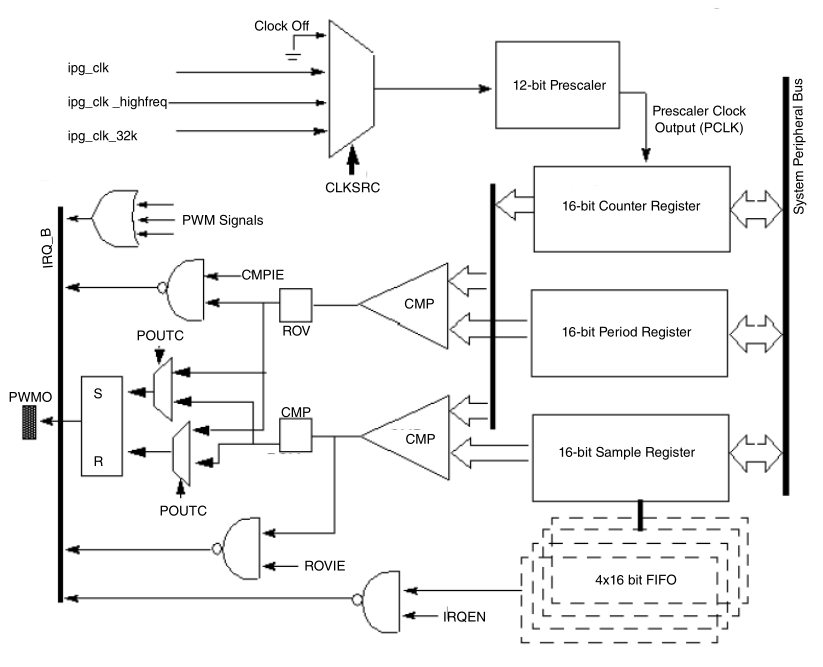
PWM全称为脉冲宽度调制，在具有惯性系统中，可以通过对一系列脉冲的宽度进行调制来等效地获 得所需要的模拟参量，常用于电机控速等领域。PWM主要有两个关键参数：频率和占空比。I.MX6ULL提供了PWM外设，配置相应寄存器设置指定频率和占空比，即可输出预期的PWM信号，介绍如下。

 图表, 折线图

描述已自动生成

**PWM信号 等价的模拟量**

I.MX6U一共有8路PWM外设，每个PWM外设包含1个16位的计数器和1个4×16的数据FIFO，结构图如下。



如图，PWM有三个时钟源可选择，我们选择ipg\_clk作为时钟源；随后有1个12位的分频器，可对时钟进行分频；右下角的三个寄存器分别为：计数寄存器、周期寄存器、采样寄存器。PWM开启后默认输出高电平，计数器从0开始计数，采样寄存器设置一个比较值，存放到数据FIFO中，当计数器寄存器中的值与数据FIFO中的值相等时，电平跳变为低电平，当计数器寄存器中的值等于周期寄存器中的值加1时，电平跳变回高电平，此时1个周期结束，计数器重新从0开始计数，下面介绍相关寄存器。

**PWMx\_PWMCNR**：计数器寄存器，低16位有效，可读取该寄存器获取当前的计数值。

**PWMx\_PWMSAR**：采样寄存器，该寄存器是数据FIFO的输入，因此该寄存器控制着占空比。

**PWMx\_PWMPR**：周期寄存器，当计数器中的值等于周期寄存器中的值加1时，重新从0开始计数。

**频率**： Freq = PCLK / (period +2) **占空比**：Duty = (PERIOD + 2) / SAMPLE

1）**控制寄存器**

PWMx\_PWMCR为控制寄存器，EN(bit0)为PWM使能位，为1时使能，为0时关闭；REPEAT(bit2:1)设置重复采样次数，即数据FIFO中的数据可以用几次，可设置0~3，分别表示FIFO可用1~4次，我们设置为0，即每个数据只用1次；SWR(bit3) 设置软件复位，为1时复位PWM，复位完成后此位自动清零；PRESCALER(bit15:4)位设置时钟分频值，可设置0~4095，即1~4096分频；CLKSRC(bit17:16)选择时钟源，为0时关闭时钟，为1时选择ipg\_clk为时钟源，为 2 时选择 ipg\_clk\_highfreq 为时钟源；为 3 时选择 ipg\_clk\_32k 为时钟源，我们设置为 1，即选择 ipg\_clk（66MHz） 为 PWM 的时钟源；POUTC(bit19:18) 设置PWM默认输出电平，为0时表示先输出高电平，为1时先输出低电平，为2或3时PWM不输出，我们设置为0，此时采样值越大，输出高电平的时间就越长，占空比就越大；FWM(bit27:26)位设置FIFO中数个数的水位线，可设置为0~3，对应1~4个数据，则FIFO中数据个数低于水位线时，状态寄存器中FE标志位FE 置1；

2）**状态寄存器**

PWMx\_PWMSR为状态寄存器，FIFOAV(bit2:0)表示FIFO中剩余数据的个数，0~4分别表示FIFO中剩余0~4个数据；FE(bit3)为FIFO为空标志位，低于控制寄存器中FWM位设置的水位线时，该位置1；ROV(bit4)为翻转标志位，当计数值等于采样值并回滚到 0X0000 时该位置1；CMP(bit5)为比较事件状态位，当发生比较事件时该位置1；FEW(bit6)为写入FIFO错误标志位，当FIFO满了之后继续向FIFO写入数据时，该位置1。

3）**中断寄存器**

PWMx\_PWMIR为中断寄存器，FIE(bit0)为FIFO空中断使能位，当FIFO为空时触发中断；RIE(bit1)为翻转中断使能位，当计数值等于采样值并回滚到 0X0000 时触发中断；CIE(bit2)为比较中断使能位，当比较事件发生时触发中断。

**注**：PWM外设中有4个16位的数据FIFO，当FIFO没满时，可通过采样寄存器向FIFO中写入数据；当FIFO写满后，再

写入数据时就会报错，且FIFO中的值不会改变。FIFO中的值可在任何时刻写入，但只有再PWM使能后才能读取；计数寄存器与FIFO中的数值进行对比，当两个值相等时，电平发生跳变，此时FIFO中的数据也会减1，当FIFO中的数据为空时，PWM将在最后一次设置的占空比中运行，直到FIFO中重新写入数据或关闭PWM，当FIFO重新写入数据后，PWM将在下个周期改变占空比。

**第三部分 LINUX系统移植**

1. **Linux系统构成**（bootloader、Linux内核和 根文件系统）

①**bootloader**

芯片上电后需要先运行一段时间bootloader代码，bootloader程序会先初始化DDR等外设，然后将Linux内核从flash(NAND，NOR FLASH，SD，MMC 等)拷贝到 DDR 中，最后启动 Linux 内核。最常用的bootloader为U-Boot。

②**Linux内核**

Linux官网为<https://www.kernel.org> ，可通过官网下载Linux内核代码；同时，NXP会从这个官网下载某个版本的Linux内核移植到自己的CPU上，测试成功后会将其发放为NXP的CPU开发者，所以我们可以移植NXP提供的Linux源码。

③**根文件系统**

根文件系统也叫做rootfs，是Linux 内核启动以后挂载(mount)的第一个文件系统，然后从根文件系统中读取初始化脚本，比如 rcS，inittab 等。根文件系统和 Linux 内核是分开的，单独的 Linux 内核无法正常工作，必须搭配根文件系统，否则Linux 内核在启动时就会提示内核崩溃(Kernel panic)。Linux中的根文件系统像是一个文件夹，文件夹里面有很多子文件夹，根文件夹和子文件夹中有许多文件，这些文件是运行Linux所必需的，如库、常用的软件和命令、设备文件、配置文件等。

1. **固化系统**（阿尔法EMMC开发板）

正点原子开发板有两种固化系统的方式：①使用正点原子修改过的NXP官方上位机工具Mfgtool、②制作SD卡系统卡，

通过插卡的方式固化系统，下面介绍具体步骤。

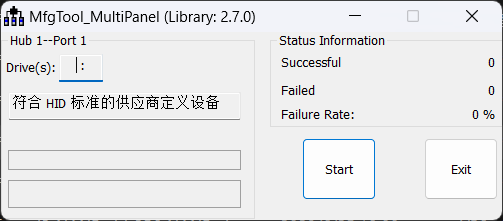
①**使用Mfgtool固化系统**（OTG）

进入正点原子资料中，在开发工具文件夹下找到Mfgtool文件夹。可通过Mfgtool将系统固化到EMMC或SD卡中，固化完成后通过拨动拨码开关选择对应启动方式即可Mfgtool工具是先烧写一个系统到DDR，然后再将要烧录的系统固化到EMMC。正点原子资料中，Mftool烧写到DDR的系统保存在Profiles文件夹中firmware文件夹下，最终烧写的系统保存在files文件夹下，具体操作步骤如下。

1）**固化到EMMC**

a. 开发板接电源、OTG，拨码开关拨到01xxxxxx（USB串行下载），拔出SD卡。

b. 双击文件Mfgtool2-eMMC-ddr512-eMMC.vbs，出现符合HID标准的供应商定义设备字样说明与电脑连接成功。



c. 点击Start开始烧写，可通过串口上位机（如SecureCRT）查看烧写信息。

d. 烧写完成，后点击Stop，点击Exit。

e. 拨码开关拨到10100110（EMMC启动），可看到系统正常启动。

2）**固化到SD卡**

a. 开发板接电源、OTG，拨码开关拨到01xxxxxx（USB串行下载）。

b. 插入SD卡，注意要先将开发板上电后再插入SD卡，否则Mfgtool会检测不到SD卡。

c. 双击Mfgtool2-eMMC-ddr512-SDCard.vbs文件，出现符合HID标准的供应商定义设备字样说明与电脑连接成功。

d. 烧写完成，后点击Stop，点击Exit。

e. 拨码开关拨到10000010（SD卡启动），可看到系统正常启动。

②**使用脚本固化系统**

脚本固化系统一般可用于批量固化与升级系统，用户可以自行修改固化系统脚本，进行自动化固化测试，即可以无需专业人员参与，完成批量固化系统。同样可固化到EMMC或SD卡中，需在Ubuntu环境下操作，具体操作步骤如下。

1）**固化到SD卡**

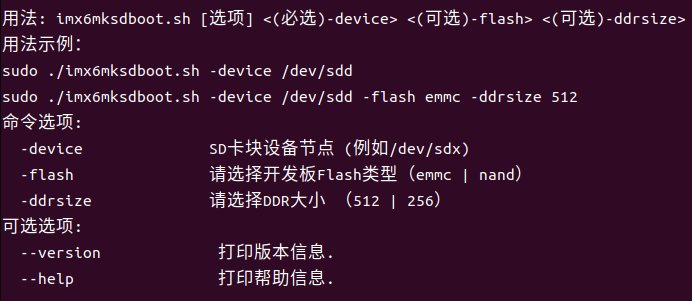
a. Ubuntu环境下新建文件夹，命名为image\_download。

b. 正点原子资料中找到开发工具文件夹，使用FileZilla将mfgtool文件夹中的file文件夹发送至新建的文件夹中。

c. 进入image\_download文件夹，打开终端。

c. 输入chmod 777 imx6mkemmcboot.sh imx6mknandboot.sh imx6mksdboot.sh给3个脚本文件赋予可执行权限。

d. 输入./imx6mksdboot.sh --help可查看帮助信息，帮助信息如下。



e. 输入sudo ./imx6mksdboot.sh -device /dev/sdc -flash emmc -ddrsize 512，根据开发板参数制作SD系统启动卡。

f. 点击回车键，开始烧写，烧写完成后会看到有两个分区：boot、rootfs，还有一个Uboot分区没显示。

g. 开发板插入SD卡，拨码开关拨到10000010（SD卡启动），可看到系统正常启动。

2）**固化到EMMC（接着上述步骤继续）**

a. 准备SD系统卡，即按上述流程将系统固化到SD卡中。

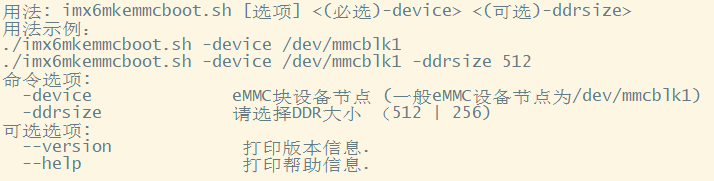
b. 进入image\_download文件夹，打开终端。

c. 输入sudo cp ~/linux/IMX6ULL/image\_dowmload/files/ /media/lxc/rootfs/home/root/ -r 将files文件夹复制到SD卡中。

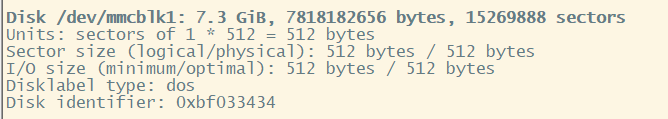
d. 将SD插入开发板，拨码开关拨到10000010（SD卡启动）。

e. 打开串口上位机SecureCRT，进入files文件夹中。

f. 输入./imx6mkemmcboot.sh –help可查看帮助信息，帮助信息如下。



g. 输入 fdisk -l查看EMMC 挂载节点，一般为/dev/mmcblk1。



h. 输入./imx6mkemmcboot.sh -device /dev/mmcblk1 -ddrsize 512，根据开发板参数将系统固化到EMMC。

i. 点击回车键，开始烧写，等待烧写成功。

j. 拨码开关拨到10100110（EMMC启动），复位开发板，可看到系统正常启动。

1. **U-Boot简介**

U-Booot的全称是 Universal Boot Loader，是一个遵循 GPL 协议的开源软件，是一个比较复杂的裸机代码，可以看作是

一个裸机综合例程。现在的U-Booot已经支持液晶屏、网络、USB 等高级功能，官网为 <http://www.denx.de/wiki/U-Boot/>。 U-Boot是最常用的bootloader，支持多种架构，可用于启动Linux系统或其他系统（如vxworks）。

①**U-Boot的作用**

Linux镜像和设备树通常存放在SD卡、EMMC、NAND、SPI Flash等外部存储介质中，Linux是在DDR中运行的。 U-Boot最重要的功能是初始化DDR，同时，U-Boot可将Linux镜像从外置flash拷贝到DDR中，为系统的启动做准备。注意，前面[BOOT ROM的作用](#BOOTROM的作用)中介绍了，I.MX6U的DDR初始化是由BOOT ROM完成的，所以对于I.MX系列芯片不需要U-Boot对DDR初始化，但对于其他芯片的DDR初始化需要由U-Boot来完成。

②**U-Boot的获取**（通常使用SOC厂商提供的U-Boot，对其修改完善）

1）**官网**：<http://www.denx.de/wiki/U-Boot/>

2）**SOC厂商**：SOC厂商会在官网下载某个版本的U-Boot，然后在此版本上将自家芯片移植进去，即定制版U-Boot。

3）**开发板厂商**：开发板厂商会参考SOC厂商的板子，在SOC厂商定制的U-Boot基础上进行修改，以适应自己的板子。

1. **编译U-Boot**

①**编译正点原子官方U-Boot并烧写到开发板**

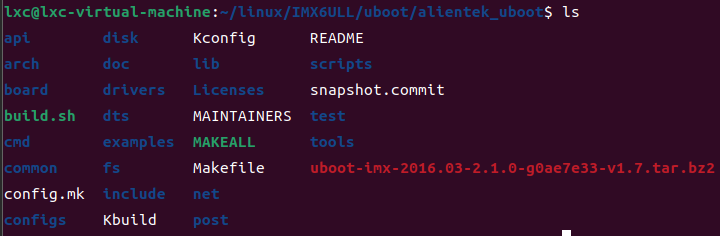
1）Ubuntu环境下创建文件夹，命名为uboot；在uboot中新建文件夹，命名为alientek\_uboot。

2）进入正点原子资料中的例程源码文件夹，进入正点原子Uboot和Linux出厂源码文件夹。

3）将uboot-imx-2016.03-2.1.0-g0ae7e33-v1.7.tar.bz2文件通过FileZilla发送到alientek\_uboot文件夹中。

4）在alientek\_uboot文件夹中打开终端，输入tar -vxjf uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2将压缩包解压缩。

5）解压后文件如下，可见有Makefile文件，下面使用make指令对U-Boot进行编译。



5）输入make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean清理工程。

6）输入make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- mx6ull\_14x14\_ddr512\_emmc\_defconfig配置U-Boot。

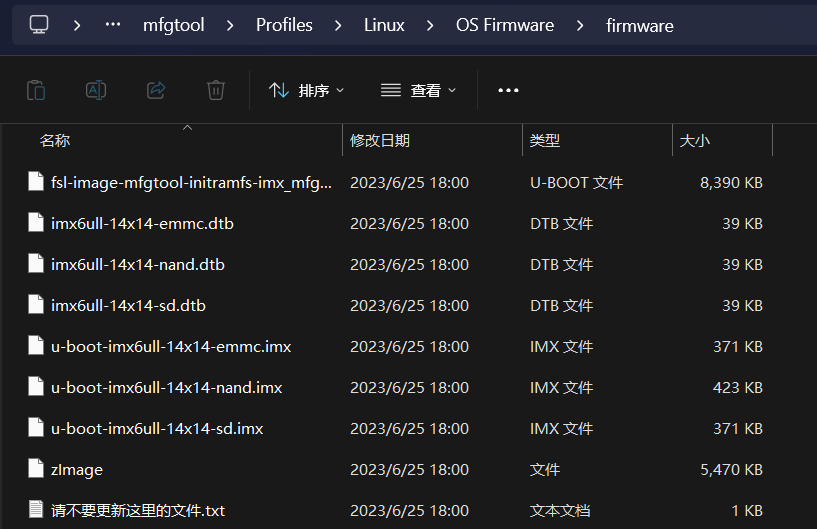
7）输入make V=1 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- 编译U-Boot。

8）将编译后生成的u-boot.imx文件通过FileZilla发送到Windows环境。

9）进入正点原子资料中开发工具文件夹，进入正点原子MFG\_TOOL出厂固件烧录工具文件夹。

10）将\files\boot文件夹下的u-boot-imx6ull-14x14-ddr512-emmc.imx替换为u-boot.imx，必须将名字改成和原来一样。

11）将firmware文件夹下的u-boot-imx6ull-14x14-emmc.imx替换为u-boot.imx，必须将名字改成和原来一样。

12）使用Mfgtool烧写系统，注意拔出SD卡。

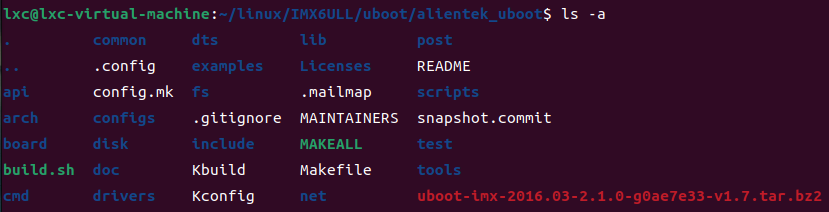
13）拨码开关拨到10100110（EMMC启动），复位开发板，系统正常启动。

注：1）第一条指令相当于make distclean，即清理工程，一般编译前都要清理一下工程；ARCH=arm指定目标为ARM架构，

CROSS\_COMPILE指定编译器为arm-linux-gnueabihf-。

2）第二条指令相当于make mx6ull\_14x14\_ddr512\_emmc\_defconfig，配置文件为 mx6ull\_14x14\_ddr512\_emmc\_defconfig，

用于对U-Boot进行配置，注意，配置文件名中mx6ull\_14x14\_为NXP定制的U-Boot的名字，以后自己做U-Boot时应该用自己的板子对这个配置文件命名，正点原子这里忘记改了。输入ls -a，可看到生成了.config文件，后续就会根据.config中的数据对U-Boot进行编译。

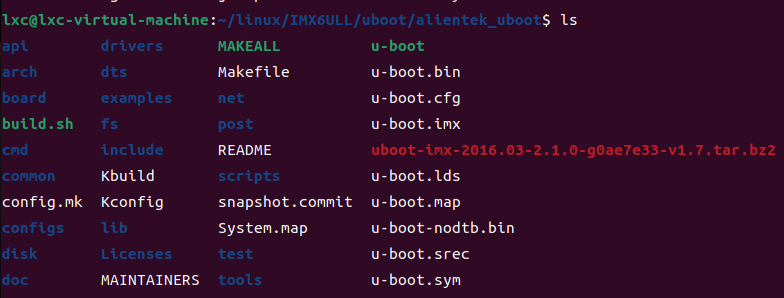


3）V=1的作用是打印详细的编译过程，第三条指令相当于make，编译U-Boot。编译完成后会生成1个u-boot.bin文件

和1个u-boot.imx文件。前面介绍了，需要向.bin文件前加一个头部信息生成.imx文件，才能烧录到I.MX6U中执行，U-Boot通过/tools/mkimage软件添加的头部信息。

4）系统启动时可通过SecureCRT查看启动信息。在启动时会出现1个3秒倒计时，在倒计时结束前按回车键即可停止

Linux系统启动。第一行可验证，这个U-Boot是刚编译的，在倒计时结束前点击回车，即可进入U-Boot命令行。

 文本

描述已自动生成

②**编写shell脚本**（编译U-Boot）

上述编译U-Boot时需要输入3个较长的指令，为方便起见可创建一个shell脚本，之后只需执行shell脚本即可完成编译，编译U-Boot的shell脚本步骤如下。

1）进入U-Boot源码目录，输入vi mx6ull\_alientek\_emmc.sh创建shell脚本。

2）按键“a”，进入输入模式，shell脚本内容如下。

#!/bin/bash #shell脚本规定，第一行必须是这个

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- mx6ull\_14x14\_ddr256\_nand\_defconfig

make V=1 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf-

3）按键“ESC”退出编辑，输入 “：”进入底行模式，按键“w”“q”保存并退出。

4）输入chmod 777 mx6ull\_alientek\_emmc.sh给shell脚本赋予执行权限。

5）输入./ chmod 777 mx6ull\_alientek\_emmc.sh执行shell脚本，U-Boot开始编译。

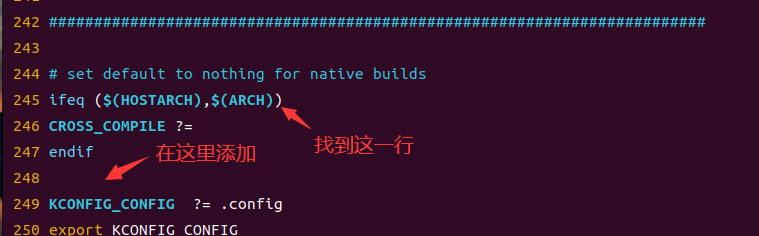
注：U-Boot是可以根据自己开发板的需求进行配置的，shell脚本中第一条指令为清理工程，当编译好shell脚本后，再执行

shell脚本会把配置文件.config清除，即之前的配置就全清除了。所以如果通过图形化界面将U-Boot配置好后，要注意shell脚本会清除配置项；如果是通过修改U-Boot源码进行配置的，则不必担心。为了方便开发，建议直接再U-Boot顶层Makefile中设置ARCH和CROSS\_COMPILE这两个变量的值，修改步骤如下。

1）输入vi Makefile进入Makefile文件。

2）按键“a”进入编辑模式。

3）在指定位置输入ARCH ?= arm，换行输入CROSS\_COMPILE ?= arm-linux-gnueabihf-。



4）按键“ESC”退出编辑，输入 “：”进入底行模式，按键“w”“q”保存并退出。

所以当使用shell脚本进行一次编译后，终端输入make V=1即可开始编译U-Boot，而不会清除配置项。

1. **U-Boot命令**

①**help命令**

输入help或者“?”，即可查看当前U-Boot所支持的所有命令，命令后面为命令说明，用于描述此命令的作用。注意这些是当前U-Boot所支持的命令，不是U-Boot只能支持这些命令， U-Boot是可以配置的，需要什么命令就使能什么命令，也可以在 uboot 中自定义命令。输入help 命令名即可查看命令的详细用法。

②**信息查询命令**

1）**bdinfo**：用于查看板子信息，如 DRAM 的起始地址和大小、启动参数保存起始地址、波特率、sp(堆栈指针)起始地

址等信息。直接输入“bdinfo”即可。

2）**printenv(print)**：用于输出环境变量信息，uboot 中的环境变量都是字符串，且值可以修改，有专门的命令来修改环

境变量的值。直接输入printenv或者print即可。

3）**version**：用于查看 uboot 的版本号，直接输入version即可。

③**创建、修改和删除环境变量命令**

1）**setenv**：用于创建环境变量或者修改环境变量的值。环境变量一般存放在外部 flash 中，uboot 启动时会将环境变量

从 flash 读取到 DRAM 中，所以使用命令 setenv 修改的是 DRAM中的环境变量值。

2）**saveenv**：用于保存新创建的或修改后的环境变量，将修改后的环境变量保存到 flash 中，否则U-Boot下一次重启会

继续使用以前的环境变量值。

**示例**：

**创建环境变量**：

如下，使用setenv创建了1个环境变量author，并赋值为luoxingchi。使用saveenv保存到flash中。重启U-Boot，

使用print查看环境变量，可看到变量创建成功。

setenv author luoxinngchi

saveenv

**修改环境变量：**

如下，使用setenv将环境变量 bootdelay 改为 5，使用saveenv将环境变量保存到了 MMC(1)中，即 EMMC 卡中，因为此时是EMMC启动，如果是SD卡启动，则环境变量会保存到MMC(0)，即SD卡。重启开发板，会发现3秒倒计时变成了5秒倒计时。

setenv bootdelay 5

saveenv

当修改的环境变量值中有空格时，需要用单引号括起来。如下指令，其中 console=ttymxc0,115200、“rootwait”、 “root=/dev/mmcblk1p2”和“rw”相当于四组“值”，这四组“值”之间用空格隔开，表示这四组“值”都属于环境变量 bootargs。

setenv bootargs 'console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw'

saveenv

**删除环境变量**：

删除环境变量时，给变量赋一个空值即可，如下指令，将环境变量author删除了。

setenv author

saveenv

④**内存操作命令**

内存操作命令就是用于直接对 DRAM 进行读写操作的，常用的内存操作命令有 md、nm、mm、mw、cp 和 cmp。

1）**md**（显示内存值）

用于显示内存值，格式为：md[.b, .w, .l] address [# of objects]。其中，[.b .w .l]指定显示内存值的格式，分别对应byte、word 和 long，即以1 个字节、2 个字节、4 个字节来显示内存值；address指定要查看的内存的起始地址；[# of objects]指定要查看的数据长度，这个长度与显示格式有关，下面举例解释。

md.b 80000000 10 //以.b格式显示以 0X80000000 为起始地址的16字节内存数据

md.w 80000000 10 //以.w格式显示以 0X80000000 为起始地址的32字节内存数据

md.l 80000000 10 //以.l格式显示以 0X80000000 为起始地址的64字节内存数据

注意，uboot 命令中的数字都是十六进制的，不是十进制的，所以可以不用写“0x”前缀。比如想查看以 0X80000000 开始的 20 个字节的内存值，应使用md.b 80000000 14，而不是md.b 80000000 20。所以上面的10分别表示读16个.b(1字节)、16个.w(2字节)和16个.l(4字节)，即读取16字节、32字节和64字节数据。

2）**nm**（修改内存值，地址不自增）

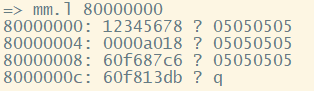
用于修改指定地址的内存值，命令格式为nm [.b, .w, .l] address。其中，[.b, .w, .l] 指定显示内存值的格式，address指定要修改的内存的起始地址，下面举例解释。

如上所示，输入nm.l 80000000，表示要修改地址为80000000处的数据；“？”前为此地址现在的值，“?”后可填写修改后的值；回车键，输入“q”即可退出；通过命令md.l 80000000 1可看到数据已修改成功。

3）**mm**（修改内存值，地址自增）

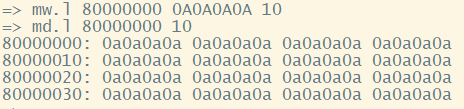
用于修改指定地址内存值，命令格式为nm [.b, .w, .l] address。其中，[.b, .w, .l] 指定显示内存值的格式，address指定要修改的内存的起始地址，下面举例解释。

如上所示，当输入05050505后，地址从80000000自增到了80000004、80000008，通过命令md.l 80000000 3可看到数据已修改成功。

4）**mw**（填充内存）

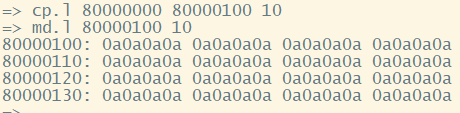
用于使用1个指定数据填充一段内存，命令格式为mw [.b, .w, .l] address value [count]。其中，[.b, .w, .l] 指定填充内存值的格式，address指定要填充内存的起始地址，value指定要填充的数据，[count]指定填充的长度，下面举例解释。



如上所示，输入mw.l 80000000 0A0A0A0A 10，表示从地址80000000开始，以0A0A0A0A填充16个long，即64个字节，通过命令md.l 80000000 10可看到数据已填充成功。

5）**cp**（拷贝数据）

用于将 DRAM 中的数据从一段内存拷贝到另一段内存中，或者把 NorFlash 中的数据拷贝到 DRAM 中，命令格式为cp [.b, .w, .l] source target count。其中，[.b, .w, .l] 指定拷贝内存值的格式，source指定源地址，targrt指定目标地址，count指定拷贝的长度，下面举例解释。



如上所示，输入cp.l 80000000 80000100 10，表示以80000000为起始地址，拷贝0x10个字节数据到地址80000100处，输入md.l 80000000 10可看到数据已拷贝成功。

6）**cmp**（比较数据）

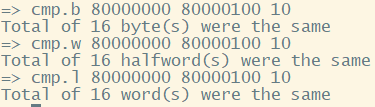
用于比较两段内存的数据是否相等，命令格式为cmp [.b, .w, .l] addr1 addr2 count，其中，[.b, .w, .l] 指定填充内存值的格式，addr1指定第一段内存的首地址，addr2指定第二段内存的首地址，count指定要比较的长度，下面举例解释。



如上所示，输入cmp.l 80000000 80000100 10，表示比较地址80000000和地址80000100处的数据，比较长度为0x10个.l，即64个字节长度，可看到输出信息Total of 16 word(s) were the same，表示数据相同。

注：关于指令中的[.b, .w, .l]，上述指令中均使用.b、.w和.l来指定操作数据的格式，.b、.w和.l看似为byte、word和long的

缩写，但其代表的数据位数不同。其中，.b：byte字节，8位；.w：halfword半字，16位；.l：word字，32位。

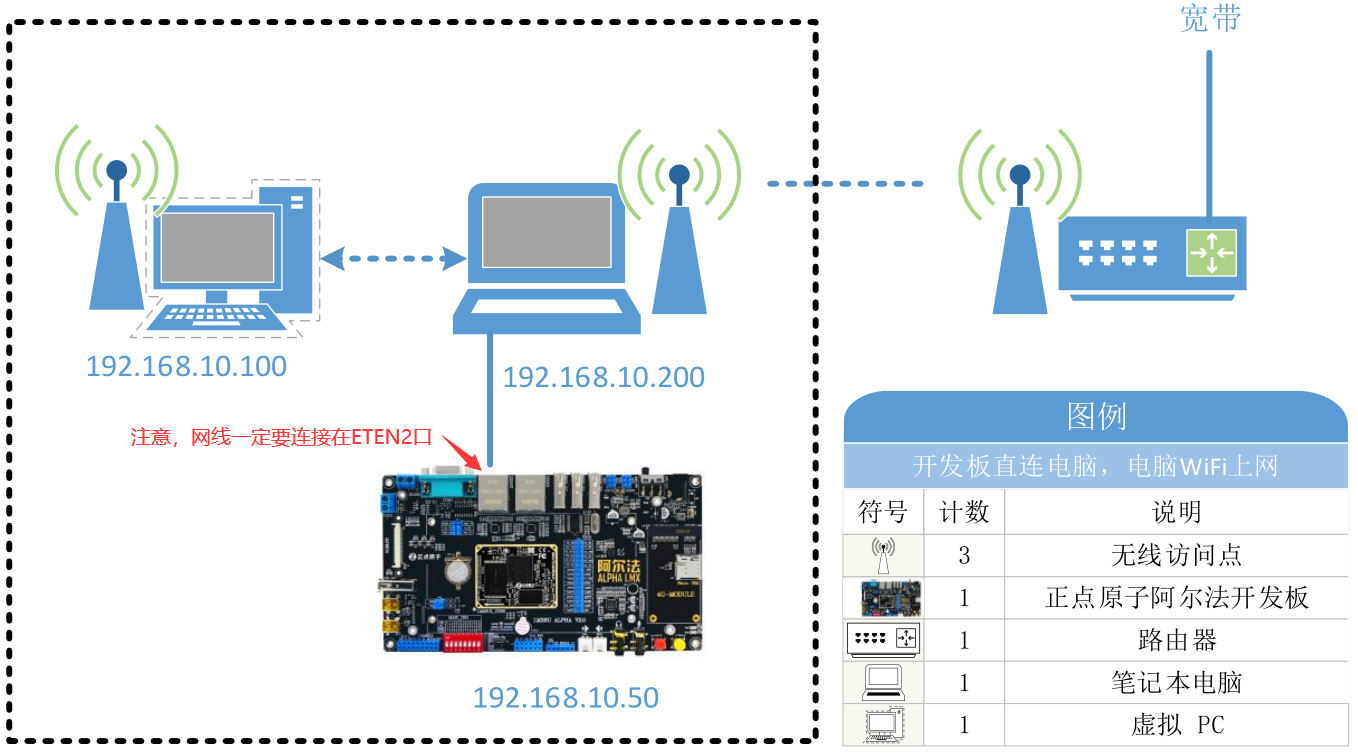


⑤**网络操作命令**

U-Boot支持网络功能，且移植U-Boot时一般都要调通网络功能，因为移植Linux内核时需要U-Boot的网络功能进行调试。U-Boot支持大量的网络命令，如dhcp、ping、nfs、tftpboot等，首先需要搭建开发板网络环境。

1）**开发板网络环境搭建**

**网络拓扑结构**：

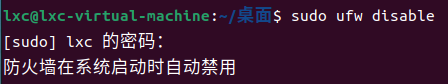


**关闭防火墙**：

a. 关闭Windows防火墙



b. 关闭Ubuntu防火墙，终端输入sudo ufw disable。



c. 如果安装了火绒，需要卸载火绒，卸载后重启电脑。（其他杀毒软件也可能导致网络连接失败）

**VMware设置**：

a. 打开虚拟机设置，网络适配器的网络连接选择桥接模式，取消勾选复制物理网络连接状态(P)。

图形用户界面, 文本

描述已自动生成

b. 因为电脑是 WiFi 上网，所以需要添加一个网络适配器并设置成 NAT 模式，供虚拟机上网。

图形用户界面, 应用程序

描述已自动生成 图形用户界面

描述已自动生成

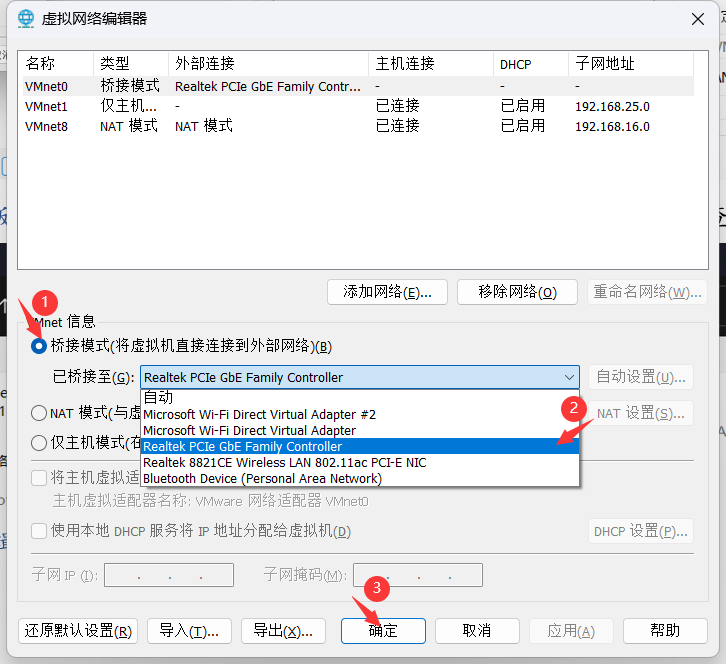
c. Windows下进入控制面板 -> 网络和Internet -> 网络和共享中心 -> 更改设备器设置，查看以太网信息。

图形用户界面, 文本, 应用程序

描述已自动生成

d. 进入VMware菜单栏编辑中的虚拟网络编辑器，点击右下角的更改设置，可看到已经自动生成了VMnet0，如果没

自动生成，则点击添加网络，选择VMnet0，点击确定即可，再按下图设置VMnet0的桥接信息，第②步与以太网信息一致。若虚拟机之前设置过网络，不清楚现在虚拟机网络的设置情况，可点击左下角的还原默认设置，再按上述步骤设置VMnet0信息，此前需要先关闭虚拟机。



**Ubuntu、Winsows网络设置**：

a. 启动虚拟机进入网络设置，以太网(ens37)即为新建的NAT 模式的网络适配器 2，Ipv4设置为自动(DHCP)。

图形用户界面, 应用程序

描述已自动生成 银色的手机截图

描述已自动生成

b. 设置以太网ens(33)，Ipv4设为手动，填写如下地址。注意，网段不能和VMnet1、VMnet8 的网段一样，否则会有冲

突。不设置DNS，否则可能为引起网络冲突；勾选仅对该网络上的资源使用此连接，设置完成后重新打开ens33。

 银色的手机截图

描述已自动生成

c. 终端输入ping www.baidu,com，测试Ubuntu上网功能，Ubuntu可正常上网。



d. Windows下进入控制面板 -> 网络和Internet -> 网络和共享中心 -> 更改设备器设置，右键点击以太网，点击属性，双

击Internet协议版本4(TCP/Ipv4)，按下图设置IP地址，点击确定即可。

**开发板设置静态IP**：

a. 启动开发板进入系统，打开串口上位机SecureCRT，输入ifconfig eth0 up打开网卡，eth0对应开发板的ENET2网口。

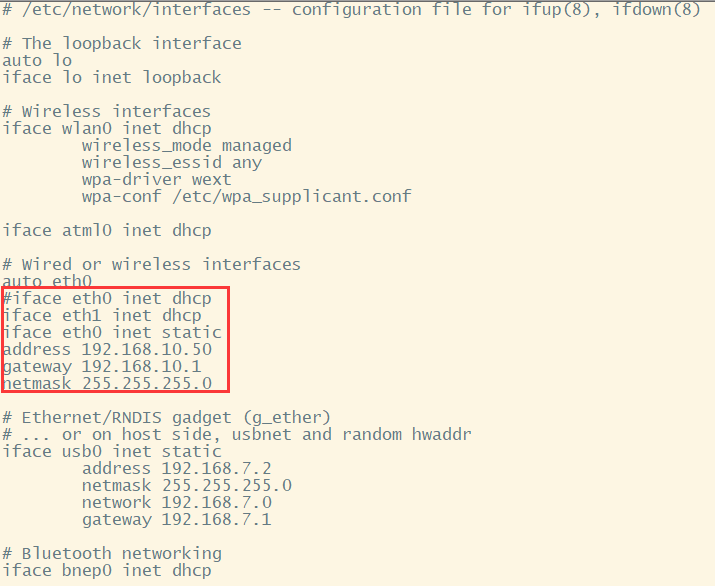
b. 输入mv /etc/init.d/connman /etc/init.d/connman2，重命名connman为connman2，目的是让系统找不到 connman 文件，

关闭 connmand 网络连接守护进程，不关闭可能会出现两个网络IP。

c. 下面设置开发板永久静态IP地址。输入vi /etc/network/interfaces，进入此文件做如下修改。按键“a”进入编辑，按键

“ESC”退出编辑，“：”进入底行模式，按键“w”“q”保存并退出。

文本

描述已自动生成 

d. 输入/etc/init.d/networking restart重启网络配置，重启开发板，输入ifconfig，可看到网卡设置成功。这种修改方法是

改完后执行/etc/init.d/networking restart 就永久生效。

文本, 信件

描述已自动生成

e. 此时开发板、Windows和Ubuntu就处于同一网段下了，对应网址如下。

开发板： 192.168.10.50 Winsows：192.168.10.100 Ubuntu：192.168.10.200

f. 进入U-Boot命令行，输入如下指令设置环境变量，进行ping测试，测试成功，开发板网络环境搭建完成。

setenv ipaddr 192.168.10.50 //设置开发板 ip 地址

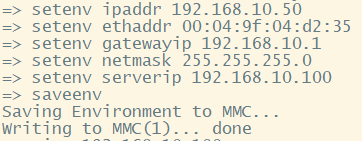
setenv ethaddr 00:04:9f:04:d2:35 //设置开发板的 MAC 地址，一定要设置

setenv gatewayip 192.168.10.1 //设置网关地址

setenv netmask 255.255.255.0 //设置子网掩码

setenv serverip 192.168.10.100 //设置服务器 IP 地址，即 Ubuntu 主机 IP 地址，用于调试代码

saveenv //保存环境变量

 文本

描述已自动生成

注意，U-Boot不能对 ping命令做处理，所以只能在U-Boot中 ping 其他的机器，其他机器不能 ping U-Boot。

2）**dhcp命令**（获取IP，启动Linux内核）

dhcp 用于从路由器获取 IP 地址，前提是开发板是连接到路由器上的，如果开发板是和电脑直连的，那么 dhcp 命令就会失效，直接输入dhcp即可。dhcp不仅可以获取 IP 地址，还会通过 TFTP 来启动 linux 内核。

3）**nfs命令**（网络调试）

nfs表示网络文件系统，用于在计算机之间通过网络来分享资源，命令格式如下。其中，loadAddress 指定文件在DRAM 中存放的地址，[[hostIPaddr:]bootfilename]指定要下载的文件地址。使用nfs命令之前需要开启Ubuntu的NFS服务，并为NFS新建一个目录，以后通过NFS访问的文件都需要放到这个目录中。

nfs [loadAddress] [[hostIPaddr:]bootfilename]

**开启NFS服务**：

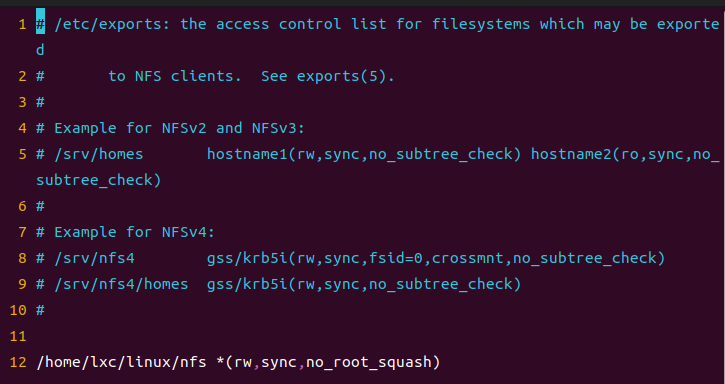
a. 进入Ubuntu终端，输入sudo apt-get install nfs-kernel-server rpcbind安装NFS服务

b. 输入sudo vi /etc/exports配置nfs

c. 进入文件，在最后一行添加/home/lxc/linux/nfs \*(rw,sync,no\_root\_squash)，保存退出

d. 输入sudo /etc/init.d/nfs-kernel-server restart重启 NFS 服务

e. linux文件夹内新建一个名为“nfs”的文件夹



嵌入式Linux开发不像单片机开发那样，可以通过JLINK等仿真器将代码一键下载到单片机内部flash中；嵌入式 Linux通常是烧写到EMMC等外置flash中，再将代码发送到内部DRAM。这时网络调试的优势就显现出来了，网络调试可以通过网络将编译好的 linux 镜像和设备树文件下载到 DRAM 中，然后就可以直接运行，网络调试是 Linux 开发中最常用的调试方法。

示例：

a. 使用FileZilla将zImage文件发送到nfs文件夹中，zImage文件的文件路径如下。

08.系统镜像\01.出厂系统镜像\02.kernel镜像\linux-imx-4.1.15-2.1.0-g3dc0a4b-v2.7\zImage

b. U-Boot命令行中输入nfs 80800000 192.168.10.100:/home/lxc/linux/nfs/zImage进行下载。

文本

描述已自动生成

4）**tftp命令**（网络调试）

Tfpt命令和nfs命令一样，都是通过网络下载数据到DRAM中，只是tftp命令使用TFTP协议，tftp命令格式如下。其中，loadAddress 指定文件在DRAM 中存放的地址，[[hostIPaddr:]bootfilename]指定要下载的文件地址。

tftpboot [loadAddress] [[hostIPaddr:]bootfilename]

与nfs不同的是，tftp不需要输入文件在 Ubuntu 中的完整路径，只输入文件名即可。Ubuntu 主机作为 TFTP 服务器，因此需要在 Ubuntu 上搭建 TFTP 服务器，操作步骤如下。

a. 进入终端。

b. 输入sudo apt-get install tftp-hpa tftpd-hpa和sudo apt-get install xinetd安装tftp-hpa 、 tftpd-hpa和xinetd。

c. 输入mkdir /home/lxc/linux/tftpboot创建存放传输数据的文件夹。

d. 输入chmod 777 /home/lxc/linux/tftpboot赋予可执行权限。

e. 输入sudo vi /etc/xinetd.d/tftp创建文件，若不存在/etc/xinetd.d/文件夹，则自行创建。

f. 文件内输入如下内容。

server tftp

{

socket\_type = dgram

protocol = udp

wait = yes

user = root

server = /usr/sbin/in.tftpd

server\_args = -s /home/lxc/linux/tftpboot/

disable = on

per\_source = 11

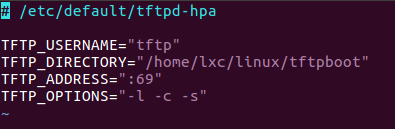
cps = 100 2

flags = IPv4

}

g. 输入sudo service tftpd-hpa start重启tfpt服务。

h. 输入sudo vi /etc/default/tftpd-hpa进入文件，修改如下内容。

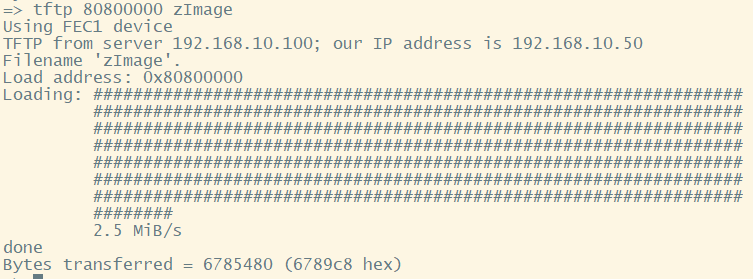


i. 输入sudo service tftpd-hpa restart重启 tftp 服务器。

j. 将Zimage文件复制到/home/lxc/linux/tftpboot中。

k. 终端输入chmod 777 zImage赋予权限。

l. U-Boot命令行输入tftp 80800000 zImage开始传输数据。



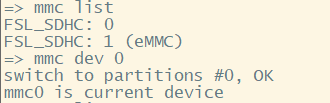
⑥**EMMC和SD卡操作命令**

U-Boot支持EMMC和SD卡，一般认为EMMC和SD卡是同一个东西，统称为MMC设备。U-Boot中用于操作MMC设备的命令为mmc。mmc是一系列命令，汇总如下。（浅蓝色指令不常用，简单了解即可）

|  |  |
| --- | --- |
| **命令** | **描述** |
| mmc info（mmcinfo） | 输出 MMC 设备信息 |
| mmc rescan | 扫描 所有的MMC 设备，包括EMMC和SD卡 |
| mmc list | 列出当前所有有效的MMC 设备 |
| mmc dev | 切换 MMC 设备，见注① |
| mmc part | 列出 MMC 设备的分区，见注② |
| mmc read | 读取 MMC 中的数据，见注③ |
| mmc wirte | 向 MMC 设备写入数据，见注④ |
| mmc hwpartition | 设置 MMC 设备的分区 |
| mmc bootbus…… | 设置指定 MMC 设备的 BOOT\_BUS\_WIDTH 域的值 |
| mmc bootpart…… | 设置指定 MMC 设备的 boot 和 RPMB 分区的大小 |
| mmc partconf…… | 设置指定 MMC 设备的 PARTITION\_CONFG 域的值 |
| mmc rst | 复位 MMC 设备 |
| mmc setdsr | 设置 DSR 寄存器的值 |

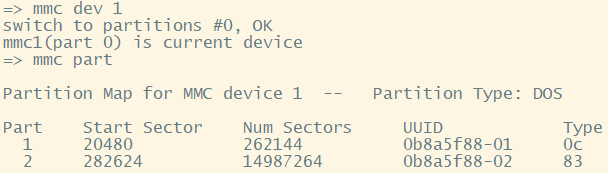
注：1）mmc dev用于切换MMC设备，命令格式为：mmc dev [dev] [part]。其中，[dev]指定MMC设备号，[part]指定分区号，

若不指定分区号，则默认为分区0。



如上所示，输入mmc list，可看到当前开发板中有两个MMC设备：EMMC和SD卡。EMMC的设备号为1，SD卡的设备号为0，默认会把EMMC设置为当前设备，此时输入mmc info，会打印EMMC的相关信息。输入mmc dev 0即可将SD卡设置为当前MMC设备，此时再输入mmc info，就会打印SD卡的相关信息了。

2）mmc part用于查看当前MMC设备分区情况，当EMMC烧写了Linux系统后，EMMC会有三个分区，如下所示。



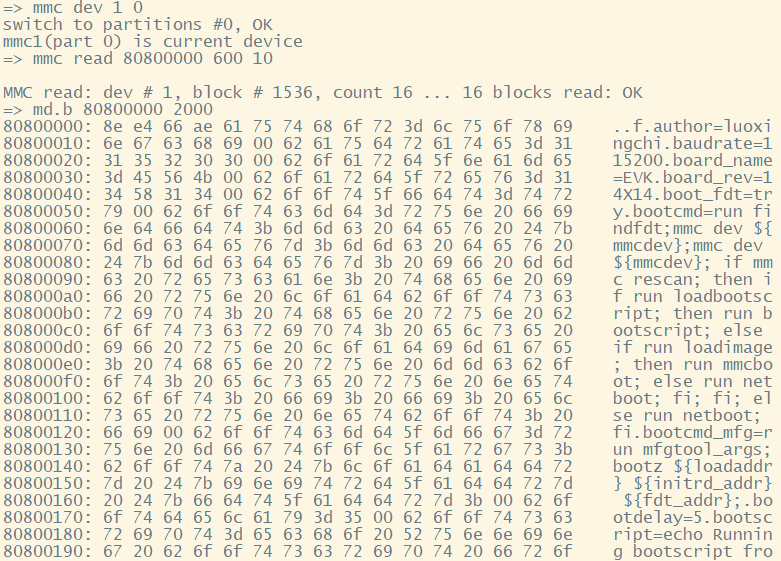
输入mmc dev 1将MMC设备更改为EMMC的分区0，输入mmc part打印EMMC的分区情况，可以看到EMMC有两个分区，第1分区的起始扇区为20480，长度为262144个扇区，此分区用于存放Linux 镜像文件和设备树；第2扇区的起始扇区为282624，长度为14987264个扇区，此分区用于存放根文件系统；其实还有第0分区，此分区存放 uboot，由于没有格式化，所以识别不出来。SD卡默认只有1个分区0，后面进行Linux移植时需要对SD卡创建并格式化第2个分区，并将Linux 镜像文件和设备树文件存放到第2个分区中。



如上所示，输入mmc dev 1 2即可将EMMC的分区2设置为当前的MMC设备。

3）mmc read用于读取MMC设备的数据，格式为mmc read addr blk# cnt。其中，sddr指定数据读取到DRAM中的

地址；blk指定读取的扇区的起始地址（16进制），一个扇区为512个字节；cnt指定要读取扇区的数量（16进制）。



如上所示，输入mmc dev 1 0将EMMC的分区0设置为当前MMC设备，输入mmc read 80800000 600 10表示从EMMC的第1536（0X600）个扇区读取16个（0X10）扇区的数据到DRAM的0X80800000地址处。输入 md.b 80800000 2000可发现读取的数据为U-Boot中的环境变量，EMMC核心板中U-Boot环境变量的起始存储地址就是第0X600个扇区，即地址1536×512=786432。

4）mmc write用于将数据写到MMC设备中，格式为mmc write addr blk# cnt。其中，addr 指定要写入 MMC 中的数据

在 DRAM 中的起始地址，blk 指定要写入 MMC的扇区的起始地址（16进制），cnt 指定要写入的扇区大小，一个扇区为 512 字节。mmc write指令可用于升级U-Boot，即在U-Boot中更新U-Boot，需要用到nfs或tftp命令，通过nfs或者tftp命令将新的 u-boot.imx 下载到开发板的 DRAM 中，然后再使用命令mmc write将其写入MMC设备中。下面使用mmc write指令更新SD卡中的U-Boot，步骤如下。

（注意，千万不要写 SD 卡或者 EMMC 的前两个扇区，里面保存着分区表！）

a. U-Boot命令行输入mmc dev 0 切换到 SD 卡。

b. 输入version查看U-Boot版本号。

c. Ubuntu下进入uboot文件夹下打开终端。

d. 终端输入make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean清理工程。

e. 输入make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- mx6ull\_14x14\_ddr512\_emmc\_defconfig配置

U-Boot。

f. 输入make V=1 ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- 编译U-Boot。

g. 将编译后生成的u-boot.imx文件复制到tftpboot文件夹中。

h. U-Boot命令行输入tftp 80800000 u-boot.imx将u-boot.imx文件传输到DRAM中。

i. 输入mmc dev 0 0选择SD卡的分区0为MMC设备。

j. 输入mmc write 80800000 2 2E6将文件烧写在分区0的第2个扇区，一共烧写 0x2E6个扇区。

k. 将开发板拨码开关设置为10000010（SD卡启动），重启开发板，进入U-Boot命令行。

l. 输入version查看U-Boot版本号，可看到U-Boot编译时间更新为这次的编译时间。

同理，将重新编译的u-boot.imx文件传输到DRAM中后，在U-Boot中更新EMMC中的U-Boot的步骤如下。

a. 输入mmc dev 1 0选择EMMC的分区0为MMC设备。

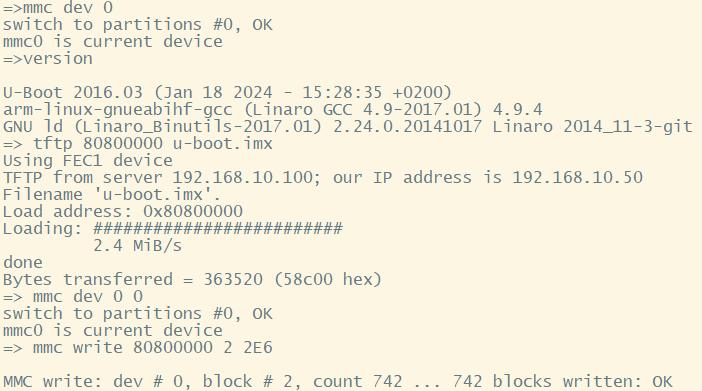
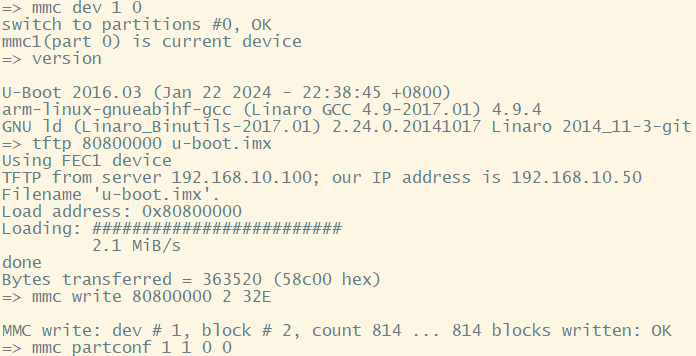
b. 输入version查看U-Boot版本号。

c. 输入mmc write 80800000 2 32E将文件烧写在分区0的第2个扇区，一共烧写0x32E个扇区。

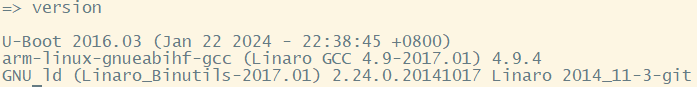
d. 输入mmc partconf 1 1 0 0进行分区配置，EMMC 需要这一步！

e. 将开发板拨码开关设置为10100110（EMMC启动），重启开发板，进入U-Boot命令行。

f. 输入version查看U-Boot版本号，可看到U-Boot编译时间更新为这次的编译时间。

**SD卡中U-Boot更新前 EMMC中U-Boot更新前**

 文本

描述已自动生成

**SD卡中U-Boot更新后 EMMC中U-Boot更新后**

5）mmc erase用于擦除MMC设备中的制定扇区，命令格式为mmc erase blk# cnt。其中，blk 指定要擦除的起始扇区，

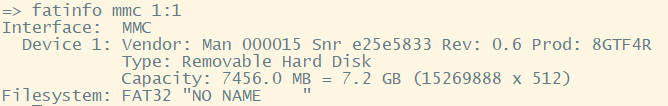
cnt 指定要擦除的数量。注意，不要随意使用mmc erase 擦除 MMC 设备。

⑦**FAT格式文件系统操作命令**

U-Boot对文件操作的相关命令为fatinfo、fatls、fstype、fatload 和 fatwrite，这些命令只支持FAT 格式的文件系统。

1）**fatinfo**（查询文件系统信息）

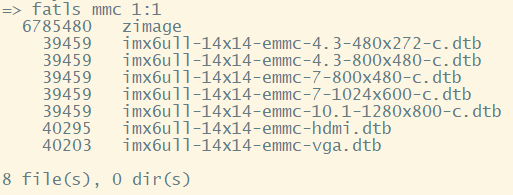
fatinfo用于查询指定 MMC 设备分区的文件系统信息，命令格式为fatinfo <interface> [<dev[:part]>]。其中，interface 指定要查询的接口，如MMC；dev指定查询的设备号，part指定要查询的分区。



如上所示，输入fatinfo mmc 1:1，表示查询MMC设备中，设备1（即EMMC）的第1个分区的文件系统信息，可见EMMC分区1的文件系统是FAT32格式的。

2）**fatls**（查询目录和文件信息）

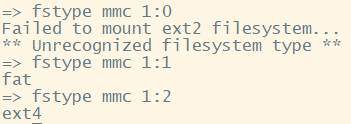
fatls用于查询 FAT 格式设备的目录和文件信息，命令格式为fatls <interface> [<dev[:part]>] [directory]。其中，interface 指定要查询的接口，如MMC；dev 指定查询的设备号，part 指定要查询的分区，directory指定要查询的目录。



如上所示，输入fatls mmc 1:1，表示查询EMMC 分区 1 中的所有的目录和文件，可看到EMMC的分区 1 中存放着 8 个文件，0个文件夹。

3）**fstype**（查看文件系统格式）

fstype 用于查看 MMC 设备某个分区的文件系统格式，命令格式为fstype <interface> <dev>:<part>。其中，interface 指定要查询的接口，如MMC；dev指定查询的设备号，part指定要查询的分区。



如上所示，输入fstype mmc 1:0，查看EMMC分区0的文件系统格式，分区0用于存放U-Boot，由于没有格式化，所以文件系统格式是未知的；输入fstype mmc 1:1，查看查看EMMC分区1的系统格式，分区 1 用于存放 linux 镜像和设备树，为fat格式；输入fstype mmc 1:2查看EMMC分区2的文件系统格式，分区2用于存放 Linux 的根文件系统，为ext4格式。

4）**fatload** （读取文件到DRAM）

fatload用于将指定的文件读取到 DRAM 中，命令格式如下。其中，interface指定接口，如MMC；dev指定设备号，part指定分区；addr指定保存在 DRAM 中的起始地址；filename指定要读取的文件的名字；bytes指定读取数据的字节数，若bytes为0或省略，则表示读取整个文件；pos指定要读的文件相对于文件首地址的偏移。

fatload <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]]]]]



如上所示，输入fatload mmc 1:1 80800000 zImage，表示读取EMMC的分区1中的zImage文件到DRAM中的0X80800000地址处。

5）**fatwrite**（向MMC设备中写入数据）

fatwrite用于将DRAM中的数据写入到MMC设备中，命令格式如下。其中，interface指定接口，如MMC；dev指定设备号，part指定分区；addr 指定要写入的数据在 DRAM中的起始地址，filename 指定写入数据的文件的名字，bytes 指定写入数据的字节数。

fatwrite <interface> <dev[:part]> <addr> <filename> <bytes>

注意，U-Boot默认没有使能fatwrite命令，所以需要修改开发板配置头文件，如 mx6ull\_alientek\_emmc.h 、mx6ullevk.h等，板子不同，其配置头文件也不同。进入配置头文件，添加宏定义#define CONFIG\_FAT\_WRITE，使能fatwrite命令，正点原子开发板已经配置好了。可通过 fatwrite 命令在 uboot 中更新 linux 镜像文件和设备树，示例如下。

a. Ubuntu环境下，将zImage文件复制到tftpboot文件夹中。

b. U-Boot命令行输入tftp 80800000 zImage，使用命令 tftp 将 zImage 下载到 DRAM 的 0X80800000 地址处。

c. 输入fatwrite mmc 1:1 80800000 zImage 6788f8将zImage写入到 EMMC 的分区 1 中。

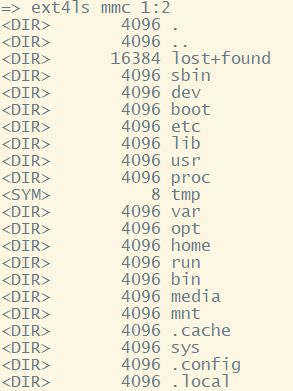
d. 输入fatls mmc 1:1可查看EMMC分区1中文件信息。

图片包含 文本

描述已自动生成

注：U-Boot也支持ext2 和 ext4两种格式的文件系统，常用的操作命令为ext2load、ext2ls、ext4load、ext4ls 和 ext4write。这

些命令和[FAT 格式文件系统操作命令](#FAT格式文件系统操作命令)使用方法相同，只是这些命令是针对ext 文件系统的。



如上所示，输入ext4ls mmc 1:2，表示查看EMMC分区2的目录和文件信息。

⑧**BOOT操作命令**

uboot 的本质工作是引导 Linux，与 BOOT 有关的常用命令有：bootz、bootm 和 boot。要启动Linux，需要先将 Linux 镜像文件拷贝到 DRAM 中，如果使用到设备树，则也需要将设备树拷贝到 DRAM 中。可以从 EMMC 或者 NAND 等存储设备中将 Linux 镜像和设备树文件拷贝到 DRAM，也可以通过 nfs 或者 tftp 将 Linux 镜像文件和设备树文件下载到 DRAM 中。

1）**bootz**（启动 zImage 镜像文件）

bootz 命令用于启动 zImage 镜像文件，bootz 命令格式为bootz [addr [initrd[:size]] [fdt]]。其中，addr 指定 Linux 镜像文件在 DRAM 中的位置；initrd指定initrd 文件在DRAM 中的地址，如果不使用 initrd ，使用‘-’代替即可；fdt 指定设备树文件在 DRAM 中的地址。

要启动Linux，需要将EMMC等存储设备中的Linux 镜像和设备树文件拷贝到 DRAM，或通过nfs或者tftp将 Linux 镜像文件和设备树文件下载到 DRAM 中，然后就可使用bootz指令启动Linux，使用网络和 EMMC 两种方法启动Linux的具体步骤如下。

**使用网络启动Linux**：

a. Ubuntu环境下把镜像文件zImage和设备树文件复制到到tftpboot 文件夹中，正点原子设备树文件路径如下。

08.系统镜像\01.出厂系统镜像\02.kernel镜像\linux-imx-4.1.15-2.1.0-g3dc0a4b-v2.7\ imx6ull-14x14-emmc-7-1024x600-c.dtb

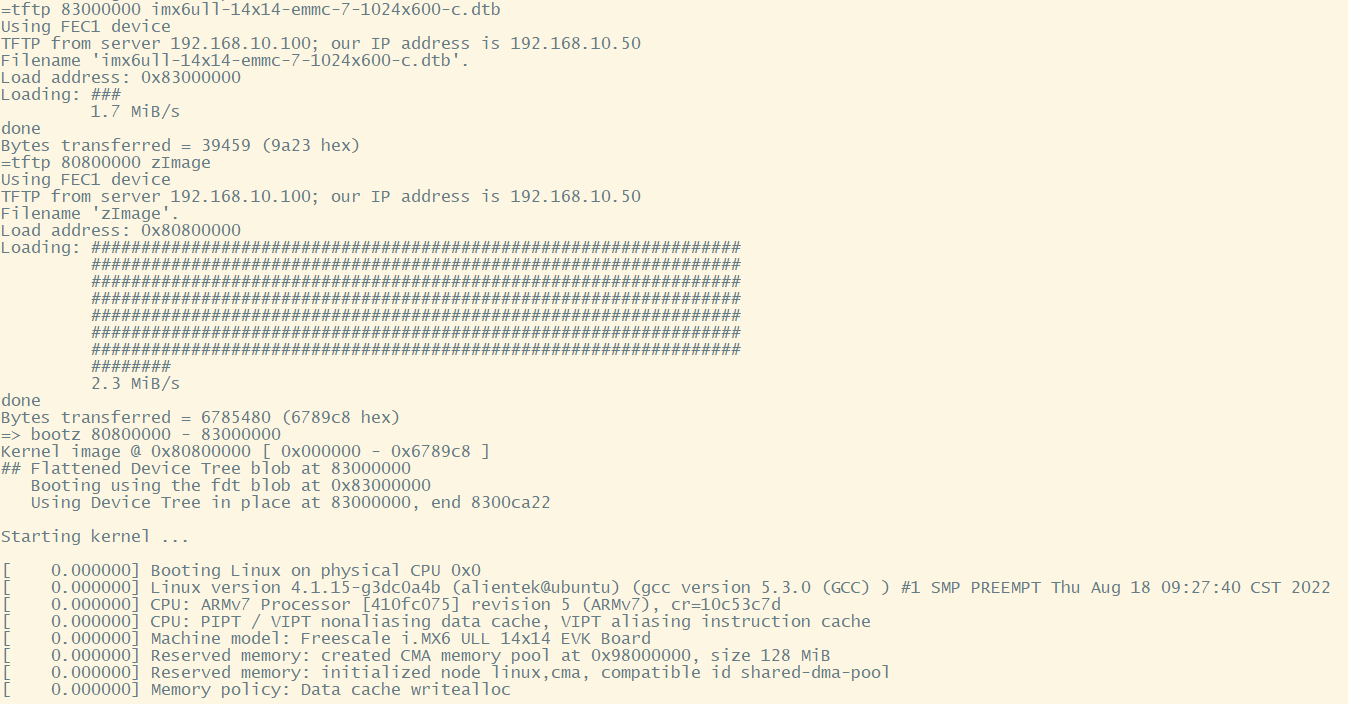
b. 进入终端，输入chmod 777 imx6ull-14x14-emmc-7-1024x600-c.dtb赋予权限。

c. U-Boot命令行输入tftp 80800000 zImage将zImage下载到DRAM的0X80800000地址处。

d. 输入tftp 83000000 imx6ull-14x14-emmc-7-1024x600-c.dtb将设备树文件下载到 DRAM 中的 0X83000000 地址处。

e. 输入bootz 80800000 – 83000000启动Linux。

（注意，启动Linux后会卡住，这是Linux的问题，这里只关注使用bootz启动Linux系统的流程，重启开发板可退出）



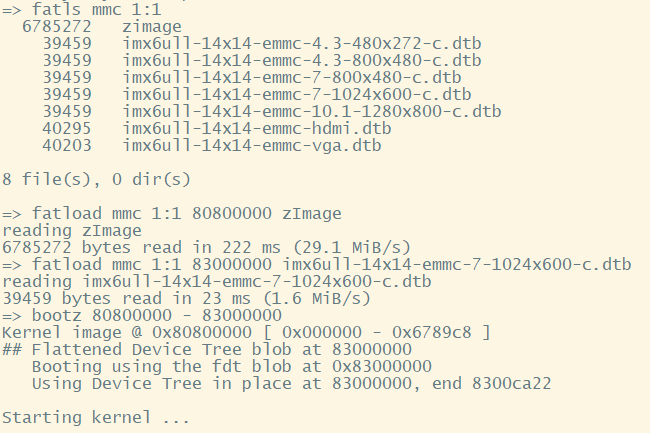
**从 EMMC 启动 Linux：**

a. U-boot命令行输入fatls mmc 1:1查看EMMC分区1中的文件目录信息。

b. 输入 fatload mmc 1:1 80800000 zImage将zImage文件从EMMC发送到DRAM的0X80800000地址处。

c. 输入fatload mmc 1:1 83000000 imx6ull-14x14-emmc-7-1024x600-c.dtb将设备树文件发送到0X83000000 地址处。

d. 输入bootz 80800000 - 83000000启动Linux。（直接卡死，还不明白原因，后续学习应该会解决）

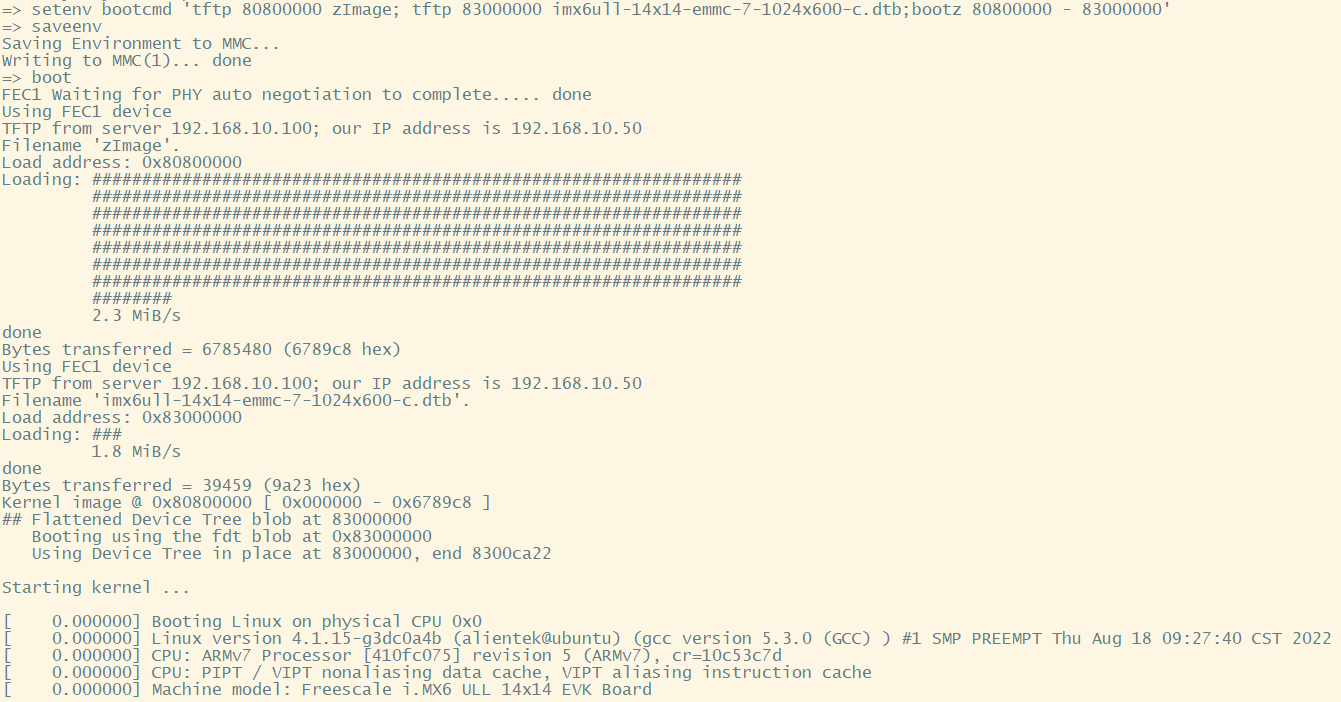


2）**bootm**（启动 uImage 镜像文件）

bootm 用于启动 uImage 镜像文件，uImage为U-Boot专用的镜像文件。不使用设备树时，启动 Linux 内核的命令为bootm addr，addr是uImage 镜像在DRAM中的首地址；使用设备树时，bootm 命令和 bootz 一样，命令格式为bootm [addr [initrd[:size]] [fdt]]，其中，addr是uImage在DRAM中的首地址，initrd是initrd 的地址，fdt 是设备树(.dtb)文件在 DRAM 中的首地址，如果 initrd 为空的话，同样是用“-”来替代。

3）**boot**（启动Linux系统）

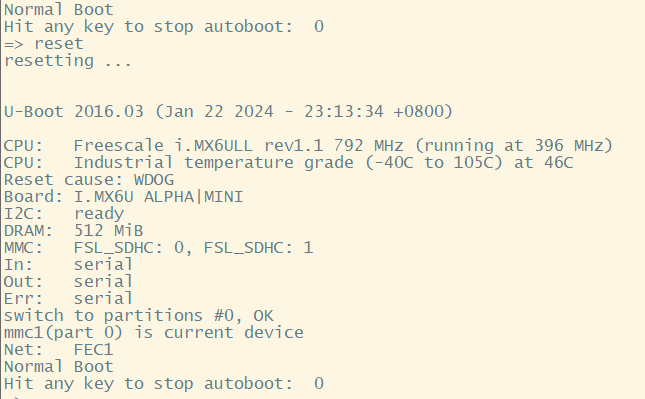
boot 命令也是用来启动 Linux 系统的，只是 boot 会读取环境变量 bootcmd 来启动 Linux 系统，bootcmd 是一个很重要的环境变量，这个环境变量保存着引导命令，即启动命令的集合，具体的命令内容是可以修改的。比如，想通过网络启动Linux，则可将环境变量bootcmd的值设置为相应指令，之后U-Boot命令行输入boot即可直接启动Linux。



之前uboot 倒计时结束以后就会启动 Linux 系统，其实就是执行的bootcmd中的启动命令。只要不修改bootcmd 中的内容，以后每次开机uboot倒计时结束以后都会使用tftp命令从网络下载 zImage 和 设备树文件，然后启动 Linux。同理，如果想从 EMMC 启动，则将环境变量设置为从EMMC启动的对应指令，每次开机 uboot 倒计时结束以后就会自动从 EMMC 里面读取zImage 和 imx6ull-14x14-emmc-7-1024x600-c.dtb，然后启动 Linux。

⑨其他常用命令

1）reset（复位重启）



2）**go**（跳到指定的地址处执行应用）

go 命令用于跳到指定的地址处执行应用，命令格式为go addr [arg ...]，其中，addr 是应用在 DRAM 中的首地址。下面介绍使用go指令执行.bin文件。

a. Ubuntu环境下将.bin文件（如pwm.bin）拷贝到tftpboot 文件夹中。

b. 输入chmod 777 pwm.bin赋予可执行权限。

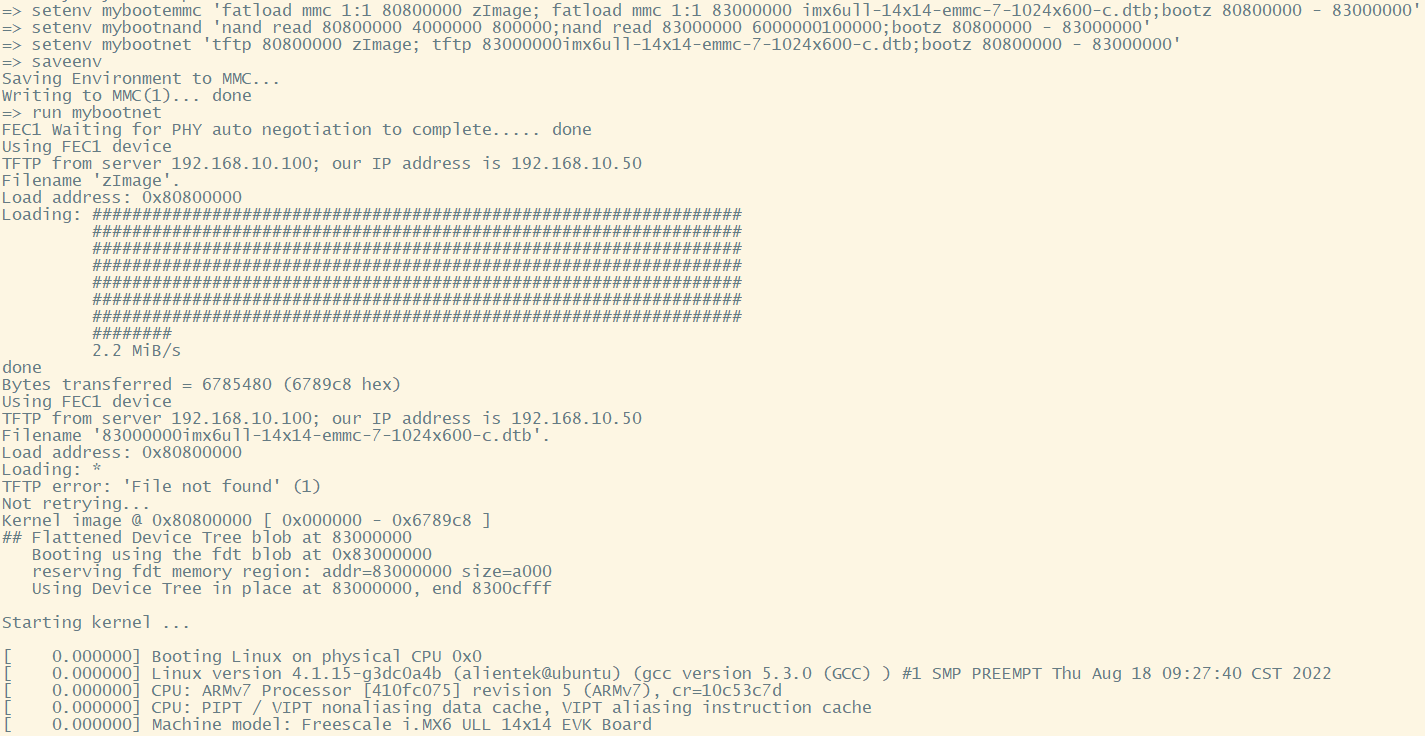
b. U-Boot命令行输入tftp 87800000 printf.bin将 pwm.bin下载到开发板 DRAM 的 0X87800000 地址处。

c. 输入go 87800000执行DRAM 的 0X87800000 地址处的printf.bin文件。

（注意，不需要在.bin前面添加 IVT 信息，因为 uboot 已经初始化好了 DDR 了）

3）**run**（运行环境变量中定义的命令）

run 命令用于运行环境变量中定义的命令，比如可以通过“run bootcmd”来运行 bootcmd 中的启动命令，但是 run 命令最大的作用在于运行我们自定义的环境变量。后面调试 Linux 系统时常常要在网络启动和 EMMC/NAND 启动之间来回切换，而 bootcmd 只能保存一种启动方式，如果要更换另外一种启动方式就需要重写 bootcmd，会很麻烦。我们可以通过自定义环境变量来实现不同的启动方式，比如定义环境变量 mybootemmc 表示从 emmc 启动，定义 mybootnet 表示从网络启动，定义 mybootnand 表示从 NAND 启动。如果要切换启动方式的话只需要运行“run mybootxxx(xxx为 emmc、net 或 nand)”即可。



4）**mtest**（内存读写测试）

mtest 命令是一个简单的内存读写测试命令，可以用来测试自己开发板上的 DDR，命令格式如下。

mtest [start [end [pattern [iterations]]]]

其中，start是要测试的DRAM的开始地址，end是结束地址。比如我们输入mtest 80000000 80001000，即可测试0X80000000~0X80001000这段内存，如下图所示，已经对这段内存测试了6252次，键盘点击“Ctrl + C”结束测试。

图片包含 公司名称

描述已自动生成