嵌入式Linux学习笔记

开发板：正点原子阿尔法I.MX6ULL（EMMC版）

**目录**

[**1. 学习Cortex-A汇编的目的** 2](#_Toc155180744)

[**2. IO初始化流程** 2](#_Toc155180745)

[**3. 程序编译步骤** 2](#_Toc155180746)

[**4. 烧写代码** 3](#_Toc155180747)

[**5. Ubuntu断网解决方法** 3](#_Toc155180748)

[**6. 启动方式选择** 3](#_Toc155180749)

[**7. 启动设备选择** 3](#_Toc155180750)

[**8. BOOT ROM作用** 4](#_Toc155180751)

[**9. 镜像文件介绍** 4](#_Toc155180752)

[**10. C语言运行环境构建** 4](#_Toc155180753)

[**11. 三极管基础** 5](#_Toc155180754)

[**12. 链接脚本** 6](#_Toc155180755)

[**13. Makefile基础** 6](#_Toc155180756)

[**14. 时钟系统** 10](#_Toc155180757)

[**15. 中断** 13](#_Toc155180758)

[**16. Ubuntu网络IP改为静态** 20](#_Toc155180759)

[**17. EPIT定时器** 20](#_Toc155180760)

[**18. GPT定时器** 20](#_Toc155180761)

[**19. UART串口** 21](#_Toc155180762)

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**：

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**

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，比如SD/EMMC、NAND)中将代码拷贝出来复制到指定的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

10000010：SD卡启动

10100110：EMMC启动

10001001：NAND FLASH启动

1. **BOOT ROM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](#时钟树)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BOOT\_CFG设置的外部存储中读取镜像，然后做相应处理。

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。

①Image 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。

②Boot 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。

③Device 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 RAM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 RAM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。固定IVT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

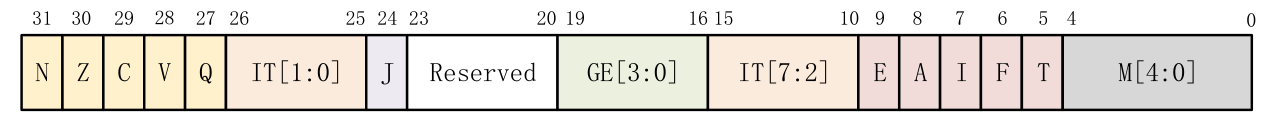
g) Hyp(HYP)：超级监视模式，用于虚拟化扩展

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**

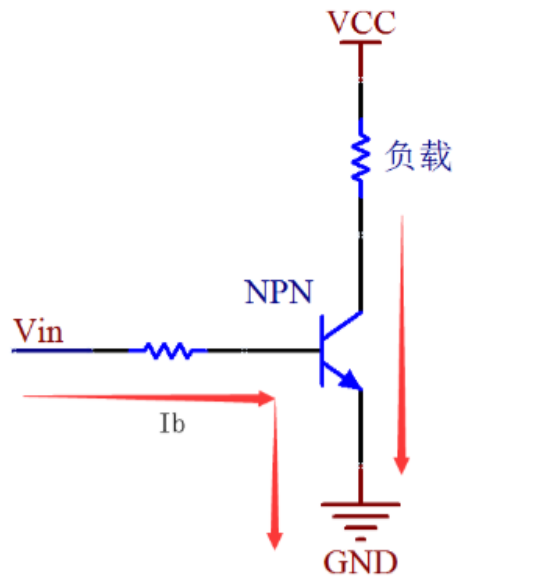
三极管做开关时，有两种工作状态：截止状态、饱和状态，一般是通过控制三极管的基极电压Ub来控制三极管

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，那到就会报链接错误码（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，这样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

⑧**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：查找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，他们是标准库文件。这些代码设置C程序的堆栈等，然后调用main函数。他们依赖于操作系统，在裸板上无法执行，这个选项常用于bootloader、编译内核等程序，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，即其频率固定为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。

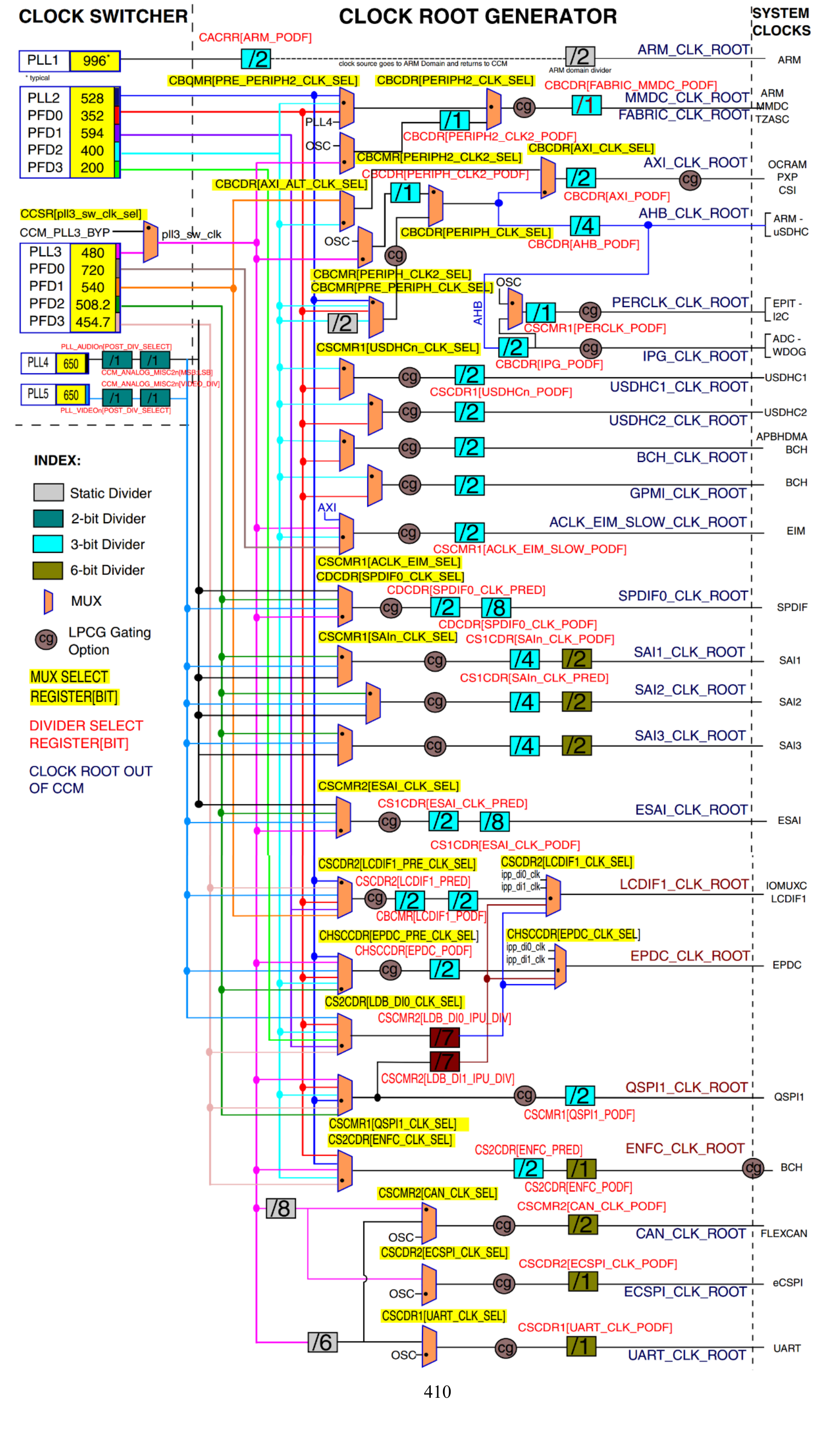
7）**PLL7**：USB2\_PLL，固定20倍频，即其频率固定为24MHz×20=480MHz，专门供USB2PHY使用。

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。而内部BOOT ROOM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。

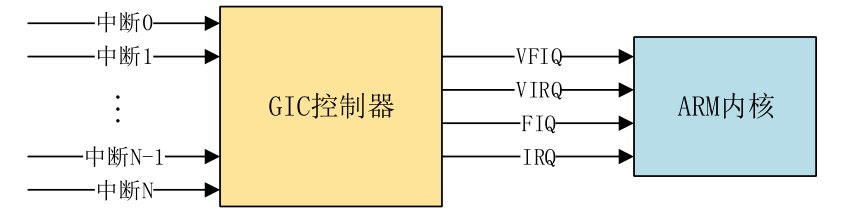
1）**中断向量表**

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，即所有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文件内的\_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，下面介绍具体的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

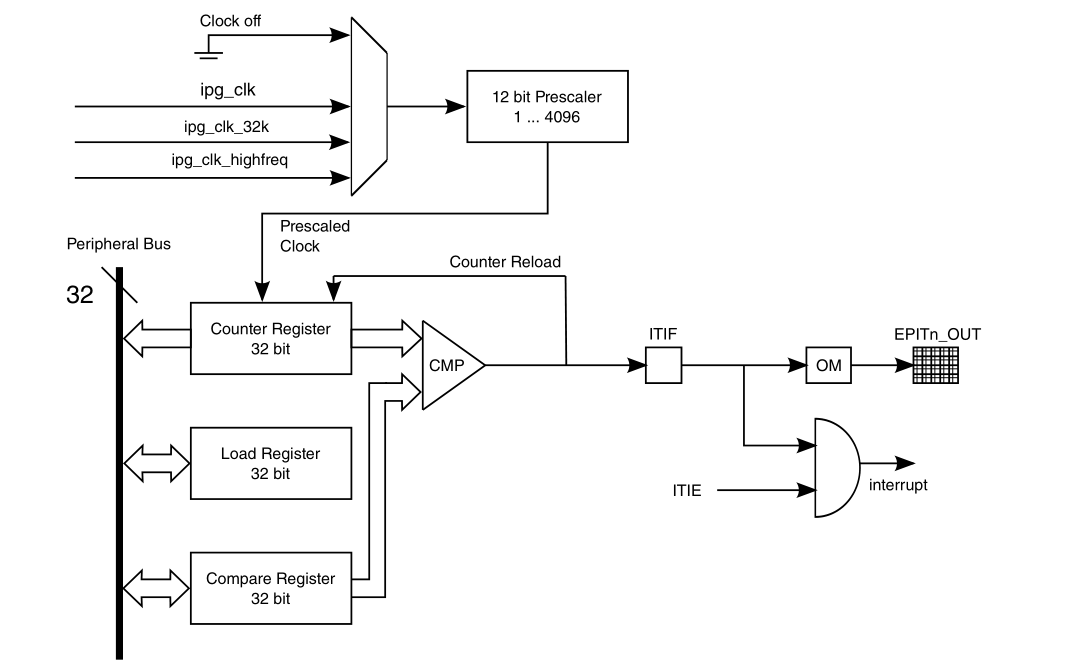
这个给定的初值开始递减，直到减为0，计数寄存器里面保存的就是当前的计数值，当计数值和比较值相等的时候产生中断。EPIT定时器有两种工作模式：set-and-forget模式和free-running模式，介绍如下。（具体寄存器信息查看参考手册1174页）

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。

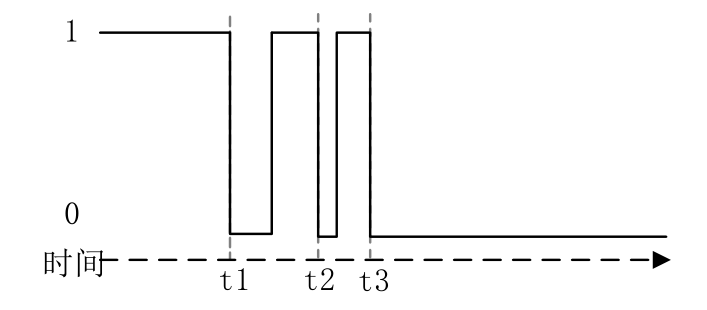
**定时时间公式**：Tout = ((frac +1 )\* value) / Tclk （frac：分频寄存器值 value：计数器值 Tclk：时钟频率Hz）

**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**

定时器可用于实现按键消抖，如下所示，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

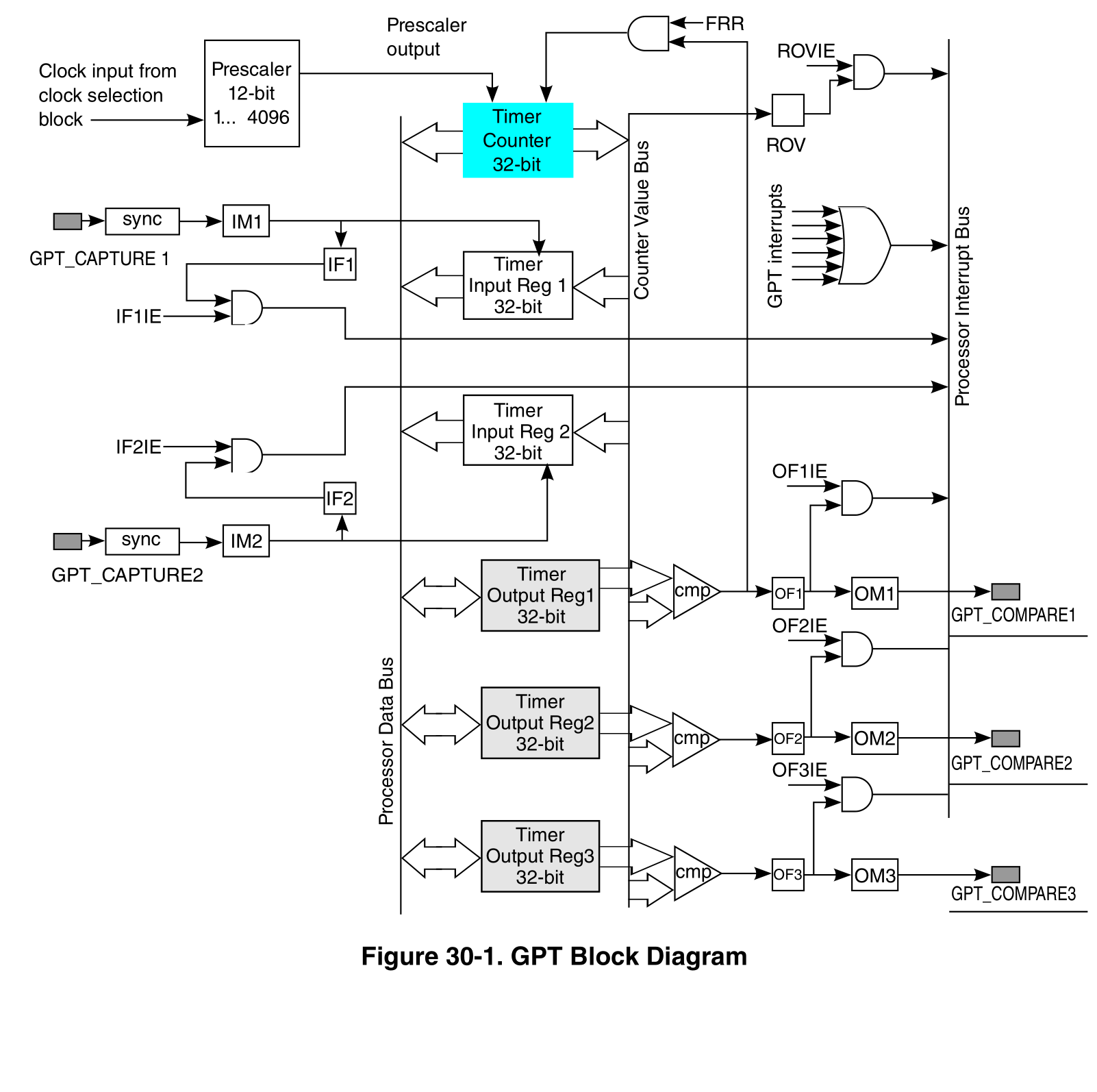
成捕获中断、比较中断和溢出中断。GPT定时器有两种运行模式：restart模式和free-run模式，介绍如下。（具体寄存器信息查看参考手册1434页）

**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。

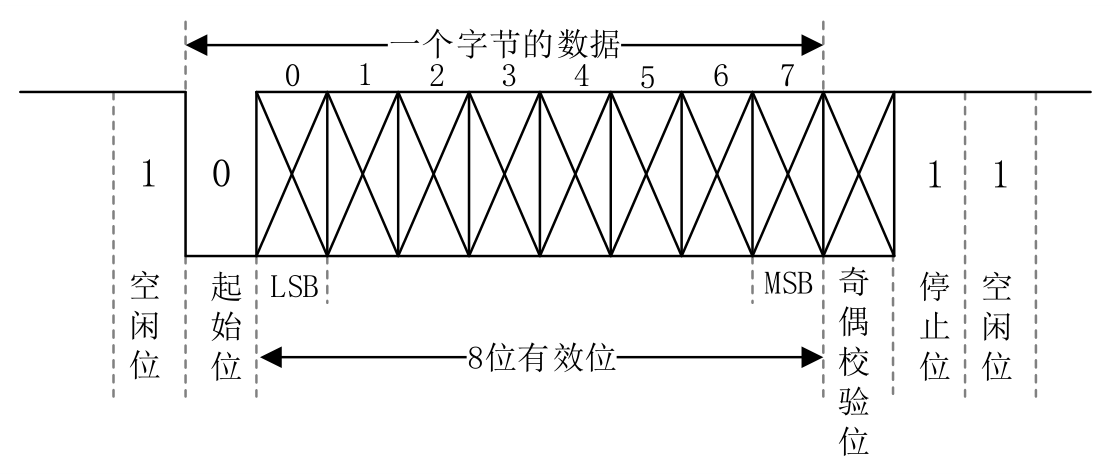


**GPT定时器结构图**

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，原因是这个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。

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。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)的RAM（DDR3L）为512MB，ROM(EMMC)为8GB。