目录

[第一章 回顾STM32中断系统 2](#_Toc67755065)

[一、 STM32中断向量表 2](#_Toc67755066)

[二、 中断向量偏移 2](#_Toc67755067)

[三、 NVIC中为控制器 2](#_Toc67755068)

[四、 中断服务函数的编写 2](#_Toc67755069)

[五、 补充一个V3S的UBOOT内存映象 2](#_Toc67755070)

[第二章 Cortex-A7中断系统 3](#_Toc67755071)

[一、 Cortex-A中断 3](#_Toc67755072)

[1. GIC中断控制器 3](#_Toc67755073)

[2. 中断ID,针对Cortex-A7内核，IMX6U、V3S都是Cortex-A7内核。 3](#_Toc67755074)

[二、 中断服务函数Reset\_Handler 4](#_Toc67755075)

[1. 关闭Ｉ、Ｄ　Cache和MMU 4](#_Toc67755076)

[2. 中断向量偏移 4](#_Toc67755077)

[3. 设置处理器9种工作模式下的SP指针 4](#_Toc67755078)

[第三章 按键中断例程(start.S) 5](#_Toc67755079)

[一、 添加中断向量表 5](#_Toc67755080)

[二、 复位中断服务函数(Reset\_Handler) 5](#_Toc67755081)

[1. 关闭I、D Cache和MMU需要操作CP15协处理器(Reset\_ Handler) 5](#_Toc67755082)

[2. 中断向量偏移设置也需要操作CP15协处理器(Reset\_ Handler) 6](#_Toc67755083)

[3. 清BSS段(Reset\_ Handler) 7](#_Toc67755084)

[4. 设置处理器9种工作模式下的SP指针(Reset\_ Handler) 7](#_Toc67755085)

[5. 开关总中断，防止设置期间的干扰 7](#_Toc67755086)

[6. 跳到C语言main函数(Reset\_ Handler) 8](#_Toc67755087)

[三、 IRQ中断服务函数IRQ\_Handler 8](#_Toc67755088)

[1. 现场保护 8](#_Toc67755089)

[2. 取得GICC\_IAR的值，存入R0 8](#_Toc67755090)

[3. 切换SVC模式 8](#_Toc67755091)

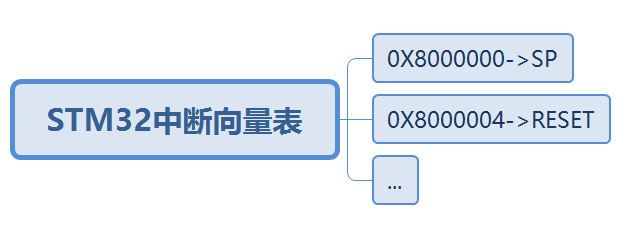
[4. 进入C语言system\_irqhandler，R0传参(GICC\_IAR) 8](#_Toc67755092)

[5. 完成system\_irqhandler之后，还原现场 9](#_Toc67755093)

# 回顾STM32中断系统

## STM32中断向量表

ARM芯片从0X00000000开始运行，执行指令。在程序开始的地方存放着中断向量表。中断向量表主要功能是描述中断对应的中断服务函数。



对于STM32来说，代码最开始的地址存放堆栈项指针

## 中断向量偏移

一般ARM从0X00000000地址开始运行，对于STM32我们设置连接首地址为0X8000000。

如果代码一定要从0X8000000开始运行，那么需要告诉一下SOC内核。也就是设置中断向量偏移，设置SCB(System Control Space 系统控制空间)的VTOR寄存器为新的中断向量表起始地址。

## NVIC中为控制器

**NVIC**就是中断管理机构，使能和关闭指定的中断、设置中断优先级。

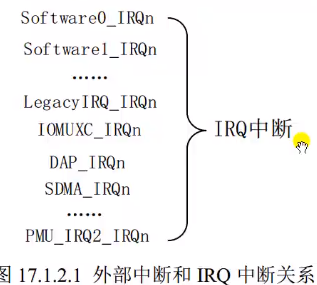
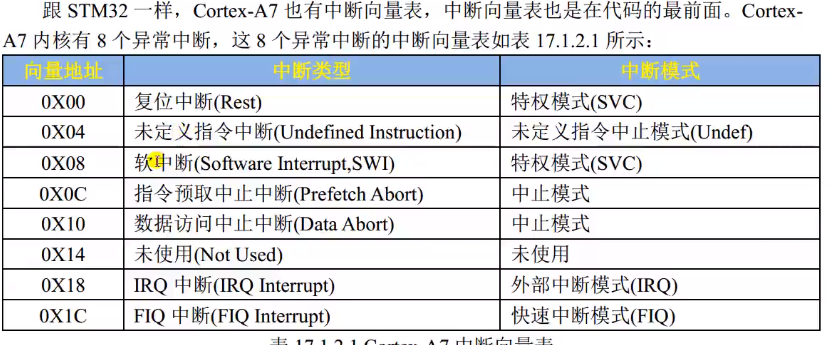
## 中断服务函数的编写

中断服务函数就是中断要作的事。

## 补充一个V3S的UBOOT内存映象



# Cortex-A7中断系统



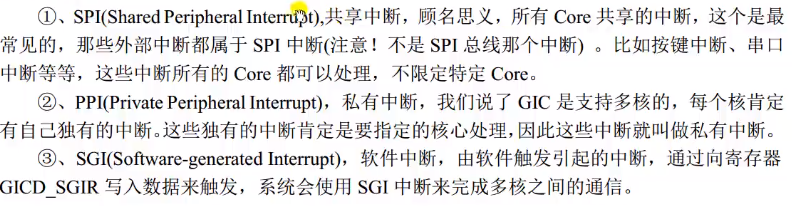
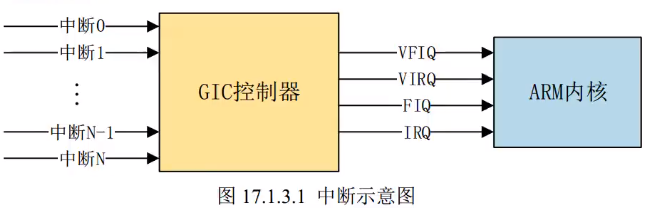
## Cortex-A中断

Cortex-A中断向量表有8个中断，其中重关注IRQ.

Cortex-A的中断向量表需要用户自己去定义。

### GIC中断控制器

同NVIC一样，GIC用于管理Cortex-A的中断，GIC提供了开关中断，设置中断优先级。



我们使用的是GIC V2(用于A7、A9,支持8核)，GIC将中断分三大类：SPI（重点）、PPI、SGI

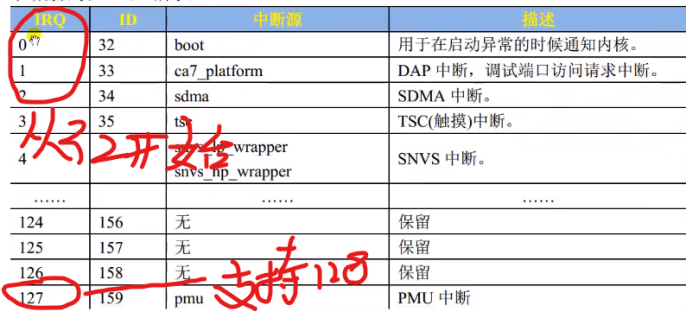
### 中断ID,针对Cortex-A7内核，IMX6U、V3S都是Cortex-A7内核。

为了区分不同的中断，引入了中断号（ID），支持1020个中断

|  |  |
| --- | --- |
| ID0-ID15 | SGI |
| ID16-ID31 | PPI |
| ID32-ID1019 | SPI，按键中断，串口中断。。。 |

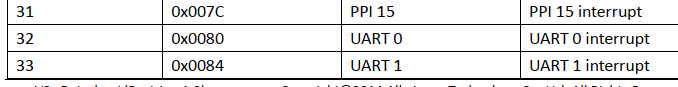
#### IMX6U中断号

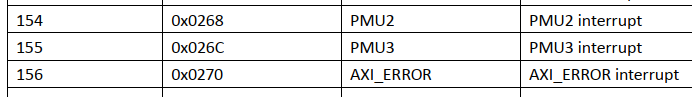
IMX6ULL支持128个SPI中断，从ID32开始



#### V3S中断号

V3S支持125个SPI中断(156-32+1=124+1=125)



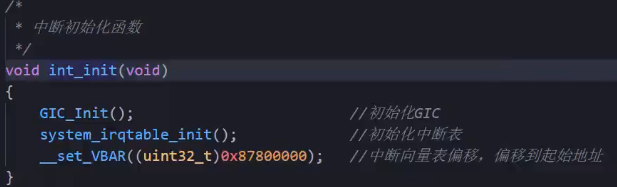
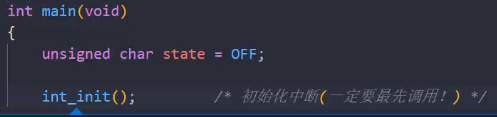


## 中断服务函数Reset\_Handler

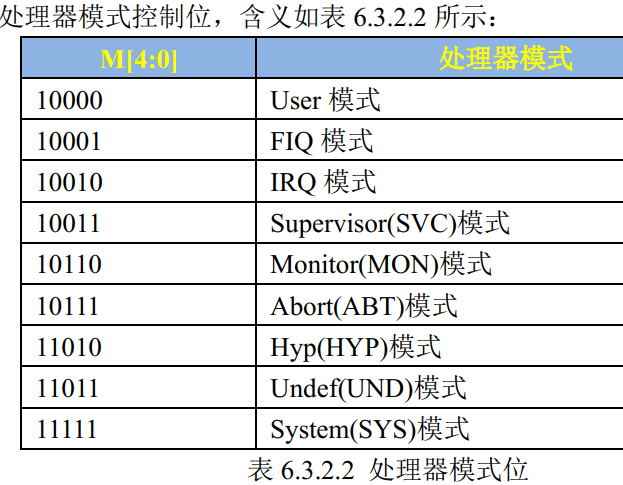
### 关闭Ｉ、Ｄ　Cache和MMU

### 中断向量偏移

我们的裸机历程都是从0X87800000开始的，因此要设置中断偏移（I.MX6U）.



### 设置处理器9种工作模式下的SP指针



# 按键中断例程(start.S)

**说明：**

* **中断使用PB1这个IO.编写PB1的中断代码，**
* **在前面章节的程序上改动（修改主频程序 8\_clk）**

## 添加中断向量表

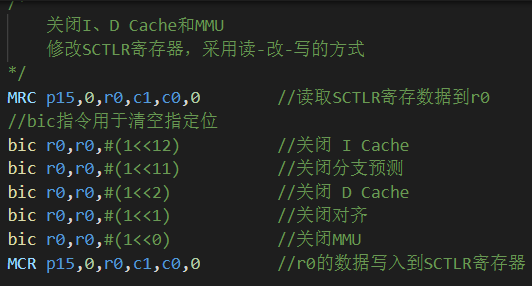
中断向量表在例程的开始位置，

即：ldr pc,=Reset\_Handler为例程执行的第一条指令。

写好对应的8个中断服务函数。

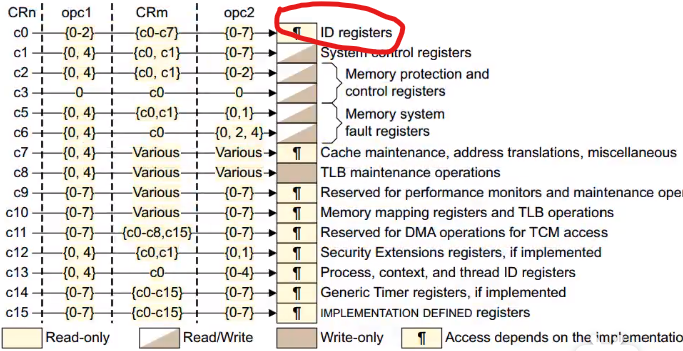
## 复位中断服务函数(Reset\_Handler)

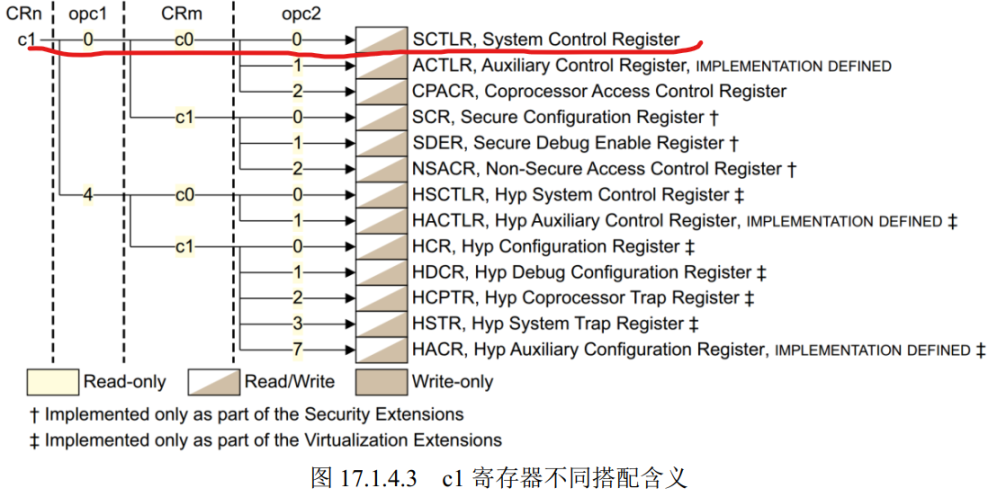
### 关闭I、D Cache和MMU需要操作CP15协处理器(Reset\_ Handler)



**MRC:** 将 CP15 协处理器中的寄存器数据读到 ARM 寄存器中。  
**MCR:** 将 ARM 寄存器的数据写入到 CP15 协处理器寄存器中。

MRC 就是读 CP15 寄存器， MCR 就是写 CP15 寄存器，

 **cond:**指令执行的条件码，如果忽略的话就表示无条件执行。  
**opc1**：协处理器要执行的操作码。  
**Rt**： ARM 源寄存器，要写入到 CP15 寄存器的数据就保存在此寄存器中。  
**CRn**： CP15 协处理器的目标寄存器。  
**CRm**： 协处理器中附加的目标寄存器或者源操作数寄存器，如果不需要附加信息就将CRm 设置为 C0，否则结果不可预测。  
**opc2**： 可选的协处理器特定操作码，当不需要的时候要设置为 0。



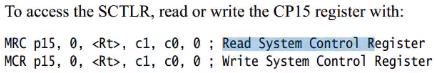
MCR 指令格式如下：**假设读ID registers**MCR{cond} p15, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

MRC p15, 0, r0, c0, c0, 0

读取 ID registers 到r0寄存器

**现在要关闭I、D Cache和MMU.**

打开Cortex-A7参考手册（P105），找到SCTLR寄存器（系统控制寄存器），此寄存器控制I、D Cache和MMU的打开和关闭等等，如下：

Bit0 用于开关MMU

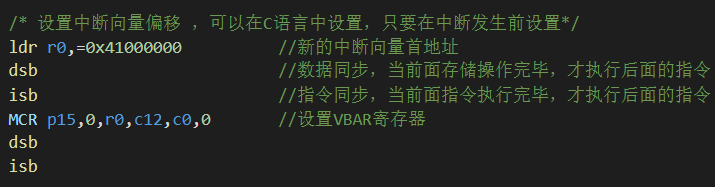
Bit1 控制对齐

Bit2 开关D Cache

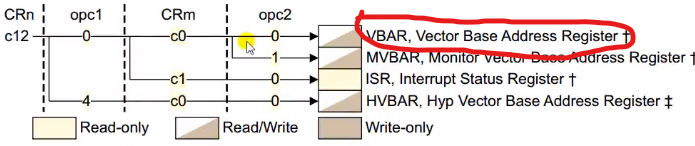
Bit11 开关分支预测

Bit12 开关I Cache

### 中断向量偏移设置也需要操作CP15协处理器(Reset\_ Handler)



将新的中断向量表首地址写入到CP15的**VBAR(Vector Base Address Register向量基地址寄存器)**



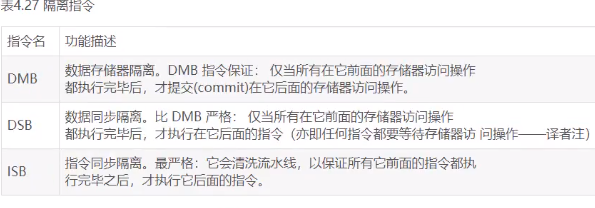
**读取VBAR寄存器示例：**

**Ldr r0,=0x41000000 //设置中断向量首地址**

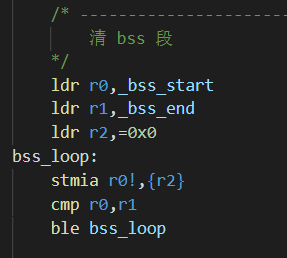
**//MCR{cond} p15, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>**

**MCR p15, 0, r0, c12, c0, 0 //首地址写入VBAR**

有必要进行数据同步与指令同步

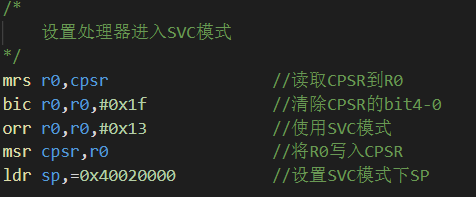
****

### 清BSS段(Reset\_ Handler)



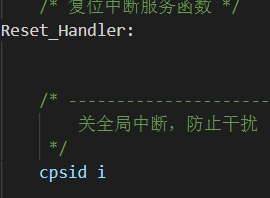
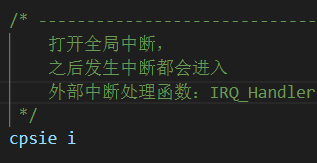
### 设置处理器9种工作模式下的SP指针(Reset\_ Handler)

要使用中断必须设置IRQ模式下的SP指针。



设置所有模式下的SP的指针。（**小细节：最后设置SVC模式，程序继续则在SVC模式下了）**

### 开关总中断，防止设置期间的干扰

### 跳到C语言main函数(Reset\_ Handler)

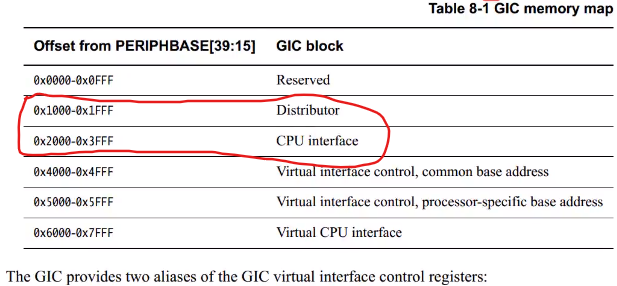
## IRQ中断服务函数IRQ\_Handler

重位中断处理函数：**Reset\_Handler**的最后，打开了全局中断:**cpsie I** ，此后，外部中断都会进入IRQ中断处理函数：**IRQ\_Handler**，**此时CPU处理IRQ中断模式**。

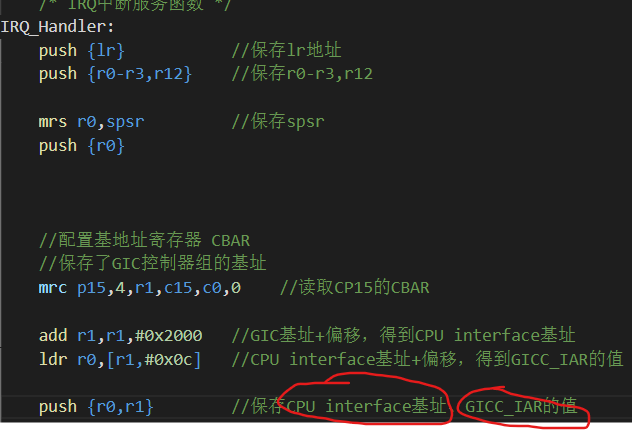
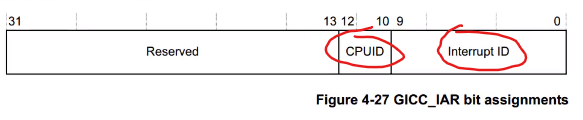
### 现场保护

### 取得GICC\_IAR的值，存入R0

通过读取CP15的CBAR，从而取得GIC控制寄存器组首地址（+偏移），我们就**可以访问GIC控制寄存器了**。





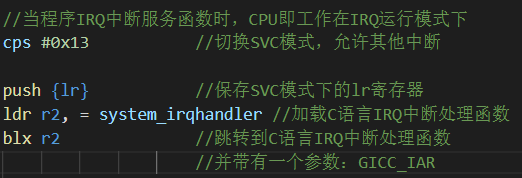
 

**GICC\_IAR**中我们可以取得**中断ID与CPUID**,从而我们可以知道发生了什么中断，是哪个CPU内核发生的。

我们读取**中断ID**的目的就是为了得到对应的**中断处理函数**。

### 切换SVC模式

### 进入C语言system\_irqhandler，R0传参(GICC\_IAR)

程序进入IRQ\_Handler,CPU即进入了IRQ运行模式，

**切换到SVC模式**，保存lr寄存器

加载C语言IRQ中断处理函数**system\_irqhandler**，并跳转至该函数运行，并带一个参数：**GICC\_IAR的值**，他是通过**R0传递参数**的，此时的R0=GICC\_IAR的值

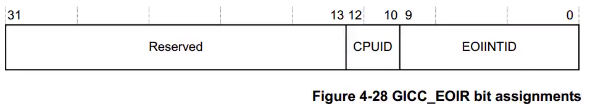
**Arm汇编给C语言传参<=3，用R0,R1,R2**

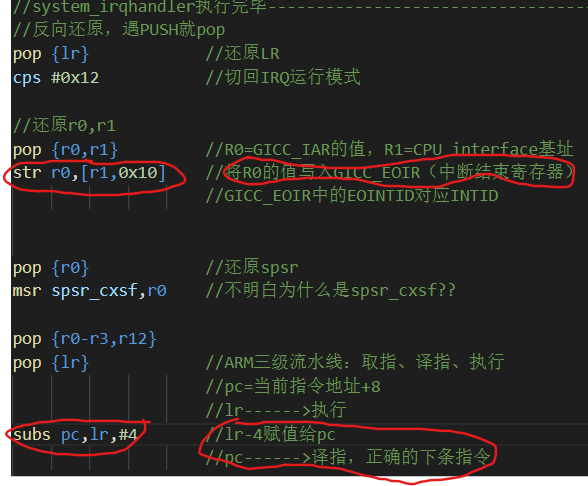
### 完成system\_irqhandler之后，还原现场

**LR寄存器，R14 也称为连接寄存器(LR)：**

* 每种处理器模式使用 R14(LR)来存放当前子程序的返回地址
* 当异常发生以后，该异常模式对应的 R14 寄存器被设置成该异常模式将要返回的地址





按入栈顺序，反向出栈。

IRQ中断处理函数结束后，一定要告诉GIC控制寄存器，中断处理完毕，方法是：

GICC\_EOIR的EOINTID写入INTID

最后修正pc地址：

ARM是三级流水线：取指、译指、执行，

Pop {lr} 之后pc=lr=当前指令地址 +8

必须pc = lr-4，使pc指向当前指令地址+4