# 一、异常

中断通常分为同步和异步：

同步中断（异常）：当指令执行时由CPU控制单元产生，只有在条指令终止执行后CPU才会发出中断。如：CPU操作错误指令，系统调用等。

异步中断（中断）：由其他硬件设备依照CPU时钟信号随机产生。如：网络收发中断等。

异常是有程序的错误产生，而中断是由间隔定时器和I/O设备产生。

## 1、类型

分为三类：故障（Fault）、陷阱（Trap）和终止（Abort），是源自CPU执行指令内部的事件。

陷阱：有意而为之的异常。最常见：操作系统的系统调用。

故障：由错误条件引起的，可能被故障处理例程修复。如果可以修复则继续干活；如果不能修复则会转化终止，进入下一步。如缺页处理。

终止：不可恢复的致命错误造成结果。终止处理程序结束后控制权不返回给引发终止的应用程序而是系统，系统通常做法是终止该应用程序。

## 2、异常处理

CPU产生的大部分异常都是有linux解释为出错条件。当一个异常发生时，跳转到相应的异常处理程序，然后向当前进程发送一个SIGFPE信号，这个进程将采取若干必要的步骤来恢复或者中止运行。

异常处理程序有个标准的结构，由以下三部分组成：

在内核堆栈中保存大多数寄存器的内容

用高级的C函数处理异常

通过ret\_from\_exception函数从异常处理程序退出。

在3.10.y中，arm平台arch/arm/kernel/entry-armv.S文件定义了各种异常处理程序入口，用汇编实现。

# 二、中断

后续讲的中断指异步中断，由外围设备或定时器产生。

## 1、类型

可屏蔽中断：I/O设备发出的所有中断请求都产生可屏蔽中断，有两种状态：屏蔽和非屏蔽。

不可屏蔽中断：只有几个危机事件（如硬件故障）才引起非屏蔽中断，CPU必须响应处理。

## 2、描述

每个能否发出中断请求的硬件设备都有一条与之对应的IRQ line。中断控制器收集各个IRQs的异步事件，用有序、可控的方式通知一个或者多个CPU。CPU跳转到相应中断向量入口执行中断处理程序。其基本框图如下：



对于arm平台，提供一个通用的中断控制器GIC(Generic Interrupt Controller)，它是通过AMBA总线连接到一个或者多个ARM processor上。GIC管理以下类型中断：

（1）外设中断（Peripheral interrupt）

外设的中断可以分成PPI（Private Peripheral Interrupt）和SPI（Shared Peripheral Interrupt）。PPI只能分配给一个确定的processor，而SPI可以由Distributor将中断分配给一组Processor中的一个进行处理。

（2）软件出发中断（SGI，Software-generated interrupt）

软件可以通过写GICD\_SGIR寄存器来触发一个中断事件，这样的中断可用于processor之间通信。

（3）虚拟中断（Virtual interrupt）和Maintenance interrupt

GIC最大支持1020个HW interrupt ID，具体ID分配情况如下（见海思datasheet）：

ID0~ID31用于分发到特定的process interrupt，ID0~ID15用于SGI，ID16~ID31用于PPI。

ID32~ID1019用于SPI。

在linux2.4之前的内核是不支持抢占特性的，而2.4开始支持抢占式内核，通过CONFIG\_PREEMPT选项配置内核支不支持抢占，抢占式内核中断处理基本流程图：



task1因等待中断的数据进入睡眠状态，调度了task2进入运行（调度发生在内核空间，为了清晰，图中省略）；task2运行过程中通过系统调用进入内核运行，然后产生硬件中断，如果没有临界区之类的关闭中断操作会马上响应中断，T0到T2都可以认为中断延迟（硬件产生到真正执行中断程序的时间），T1到T2的时间极短，图中夸大方便看；中断处理程序中唤醒了task1进入running状态（中断里不可调度）；待中断处理程序结束，中断返回（两者之间的结束处理时间也极短），因为可抢占，系统进行了任务调度；发现优先级task1>task2，先执行task1，task1执行完毕后在继续之前的task2执行。

## 3、中断处理程序

CPU响应一个外部中断时，会执行一个与之对应的特定函数，这个处理函数就称为中断处理程序或中断服务例程。每个中断设备都有一个相应的中断处理程序。

中断处理程序就是普通的C函数，只是在定义格式上按照规定类型声明，但是只能运行在称之为中断上下文的特殊上下文中。中断上下文和进程没有什么瓜葛，与current宏也是不相干，不同与进程上下文，它不可以阻塞。

（1）注册中断处理程序

驱动程序可以通过request\_irq()函数注册一个中断处理程序，其原型如下：

static inline int \_\_must\_check request\_irq(unsigned int irq,

irq\_handler\_t handler,

unsigned long flags,

const char \*name,

void \*dev)

irq：中断号

handler：中断的实际处理函数

flags：中断处理程序标志

name：与中断相关的设备的ASCII文本表示，显示在/proc/interrupts

dev：注册的设备，如果是共享中断号，则必须传递有效值，否则可以为NULL。

（2）中断处理程序标志

flags可以为0，也可以是下列一个或多个标志的位掩码：

IRQF\_TRIGGER\_XXX：描述该interrupt出发类型的flag

IRQF\_DISABLED：表示处理中断处理程序期间，禁止所有其他中断。一个废弃的flag，旧内核（2.6.35版本之前）两种handler：slow handler和fast handler，fast handler需要传递该参数关闭CPU中断；新内核不区分，皆为fash handler，都需要关闭CPU中断，需要后续处理的内容推到下半部处理。

IRQF\_SHARED：表示多个中断处理程序之间共享一个中断线

IRQF\_PERCPU：在SMP的架构下，中断有两种模式：一种是在所有processor之间共享，中断可以分配给任意一个CPU处理；另外一种是特定属于一个CPU，产生中断只能有特定CPU处理。如上面的arm GIC控制器。

IRQF\_NOBALANCING：在smp下，上面共享processor的中断会均衡的分配到各个CPU，如果不想均衡就设定该flag。

IRQF\_IRQPOLL

IRQF\_ONESHOT

IRQF\_NO\_SUSPEND

IRQF\_FORCE\_RESUME

IRQF\_NO\_THREAD

IRQF\_EARLY\_RESUME

IRQF\_TIMER

（3）释放中断处理程序

void free\_irq(unsigned int irq, void \*dev\_id)

（4）共享中断处理程序

flags有IRQF\_SHARED，在注册时必须传递参数dev，而且每个注册程序该dev都是唯一的。因为共享，所以通过该dev来区分该中断调用哪个中断处理程序处理；释放时也是根据dev释放对应的中断处理程序。当内核接收到一个中断后，将依次调用在该irq注册的每一个处理程序。

## 4、上半部和下半部

由于中断打断了其他的代码，因此中断处理程序应该迅速执行完毕，避免消耗大量时间，影响系统的实时性。因此，linux一般把中断切换为两个部分：上半部和下半部，上半部（即中断处理程序）只做有严格时限的工作，能够被允许稍后完成的所有工作尽量放在下半部（稍后一定时间内再去处理），细节见下一章。

## 5、/proc/interrupts

该文件存放的是系统中与中断相关的统计信息：

CPU0 CPU1 CPU2 CPU3

35: 694273 0 0 0 GIC timer4

36: 0 0 699620 0 GIC timer6

38: 1285660 0 0 0 GIC uart-pl011

43: 10509 0 0 0 GIC Hisilicon Dmac

51: 0 0 0 0 GIC hi\_mci

52: 1 0 0 0 GIC ohci\_hcd:usb2

53: 0 0 0 0 GIC ehci\_hcd:usb1

55: 6918815 0 0 0 GIC stmmaceth

56: 0 0 0 0 GIC stmmaceth

57: 0 0 0 0 GIC stmmaceth

58: 0 0 0 0 GIC stmmaceth

59: 0 0 0 0 GIC ahci

60: 0 0 0 0 GIC xhci-hcd:usb3

78: 0 0 0 0 GIC IVE

79: 0 0 0 0 GIC AIO Interrupt

80: 0 0 0 0 GIC hi\_tde\_irq

94: 0 0 0 0 GIC hi\_iommu

105: 6 0 0 0 GIC timer

107: 0 688852 0 0 GIC timer5

108: 0 0 0 699547 GIC timer7

112: 0 0 0 0 GIC VOU Interrupt

IPI0: 0 1 1 1 CPU wakeup interrupts

IPI1: 0 0 0 0 Timer broadcast interrupts

IPI2: 11100056 11902540 10996312 11892147 Rescheduling interrupts

IPI3: 6 6 6 3 Function call interrupts

IPI4: 0 0 0 1 Single function call interrupts

IPI5: 0 0 0 0 CPU stop interrupts

IPI6: 0 0 0 0 CPU backtrace

第一列：中断号

第二列：CPU0~CPU3 接收到中断的计数

第三列：相应的中断名字

## 6、中断控制接口

Linux内核提供了一组接口用于操作机器上的中断状态。

（1）禁止和激活中断

local\_irq\_disable：禁止本地中断

local\_irq\_enable：激活本地中断

local\_irq\_save：保存本地中断的当前状态，然后禁止本地中断传递

local\_irq\_restore：恢复本地中断传递给给定状态

必须在同一个函数内完成

（2）禁止指定中断号

disable\_irq：禁止给定中断，确保该函数返回之前在该中断号上没有处理程序在运行

disable\_irq\_nosync：禁止给定中断

enable\_irq：激活给定中断号

（3）中断系统状态

irqs\_disabled：如果本地中断被禁止，则返回非0，否则0

in\_interrupt：如果在中断上下文中，则返回非0，否则0

in\_irq：如果在当前正在执行中断处理程序，则返回非0，否则0

# 三、下半部处理

下半部的任务就是执行与中断处理密切相关但中断处理程序本身不执行的工作，最理想情况下，最好是中断处理程序将所有工作都交给下半部执行。对于上半部和下半部之间划分工作，不存在某种严格的规则，但还是有一些提示可借鉴：

如果一个任务对时间非常敏感，将其放在中断处理程序中执行

如果一个任务和硬件相关，将其放在中断处理程序中执行

如果一个任务要保证不被其他中断打断，将其放在中断处理程序中执行

其他所有任务都考虑防止在下半部执行

和上半部只能通过中断处理程序实现不同，下半部可以通过多种机制实现：软中断softirq、tasklet、工作队列work queue。

## 1、软中断

（1）描述

软中断使用的比较少，tasklet是下半部更常用的一种形式。但是tasklet也是在软中断基础上实现的，是其中一种的软中断。

软中断本身不能进入休眠。

软中断是在编译期间静态分配的，不想tasklet可以动态注册或注销。其实现结构体如下：

struct softirq\_action

{

void (\*action)(struct softirq\_action \*);

};

static struct softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp一共有NR\_SOFTIRQS个软中断，在smp情况下地址按照cacheline对齐，该数据结构多个CPU共享。目前内核总共有如下几类软中断：

enum

{

HI\_SOFTIRQ=0, //高优先级的tasklet，也就是最先执行

TIMER\_SOFTIRQ, //基于系统tick的software timer

NET\_TX\_SOFTIRQ, //网卡驱动发送

NET\_RX\_SOFTIRQ, //网卡驱动接收

BLOCK\_SOFTIRQ, //block设备

BLOCK\_IOPOLL\_SOFTIRQ, //block设备

TASKLET\_SOFTIRQ, //普通tasklet

SCHED\_SOFTIRQ, //用于多CPU之间的负载均衡

HRTIMER\_SOFTIRQ, //用于高精度timer

RCU\_SOFTIRQ, //处理RCU永远用于RCU

NR\_SOFTIRQS

};

一个软中断不会抢占另外一个软中断，但是中断处理程序可以抢占软中断。但是其他软中断（甚至同一个软中断）可以在其他处理器上同时运行。每个CPU通过下面结构来表示分配给自己的软中断，谁触发谁处理，例如：本次触发CPU0则有CPU0处理，同一硬件中断下次CPU1则CPU1处理。

typedef struct {   
    unsigned int \_\_softirq\_pending;   
#ifdef CONFIG\_SMP   
    unsigned int ipi\_irqs[NR\_IPI];   
#endif   
} \_\_\_\_cacheline\_aligned irq\_cpustat\_t;

irq\_cpustat\_t irq\_stat[NR\_CPUS] \_\_\_\_cacheline\_aligned;

\_\_softirq\_pending：每1位代表对应上面一个软中断，因此NR\_SOFTIRQS最多为32个，每个CPU都有自己的irq\_stat记录分配给自己的软中断。

（2）软中断执行

一个注册的软中断必须被触发（raise\_softirq）后才会执行。通常中断处理程序返回时会触发软中断，使其稍后被执行。一般有一下几种情况：

从硬件中断处理程序处理完退出时，如果在中断上下文irq\_exit-> invoke\_softirq执行

从硬件中断处理程序处理完退出时，如果在进程上下文ksoftirqd执行（有几个CPU就有几个这样的线程）

代码中显示调用执行，如网络子系统。

（3）使用软中断

软中断保留给系统中对时间要求最严格以及最重要的下半部使用。尽量使用现有的软中断，两个子系统（网络和SCSI）直接使用软中断。此外，内核定时器和tasklet都是建立在软中断上。

注册软中断处理程序：

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

参数为：软中断的索引号和处理函数（不能睡眠）。

在一个处理程序运行时候，当前处理器的软中断被禁止，但是其他处理器仍然可以处理其他中断。实际上，如果同一个软中断在它被执行的同时再次触发了，那么另外一个处理器可以同时运行其处理程序，这意味着软中断处理函数中的共享数据需要加锁保护。软中断的优势就是其可扩展到不同处理器，加速性能，如果加锁保护影响扩展还不如使用tasklet。因此软中断处理函数最好可重入。

触发软中断：

void raise\_softirq(unsigned int nr)；

inline void raise\_softirq\_irqoff(unsigned int nr)；

前者先禁止本地中断，再触发软中断；后者如果本地中断已经禁止则直接调用，常用于中断处理程序里。

## 2、tasklet

（1）描述

tasklet是利用软中断实现的一种下半部机制，和软中断本质上相似，也和进程没有任何关系，但是接口更简单，锁保护要求更低。因为同一个tasklet处理程序不能同时在多个处理器上运行，但其他tasklet处理程序可以在其他处理器同时运行，这点和软中断主要区别。

实质是一种软中断，因此也不能进入睡眠。

tasklet由两类软中断：HI\_SOFTIRQ和TASKLET\_SOFTIRQ，在初始化时注册：

open\_softirq(TASKLET\_SOFTIRQ, tasklet\_action);

open\_softirq(HI\_SOFTIRQ, tasklet\_hi\_action);

tasklet有tasklet\_struct结构体表示：

struct tasklet\_struct

{

struct tasklet\_struct \*next; //链表中下一个tasklet

unsigned long state; //tasklet状态

atomic\_t count; //引用计数

void (\*func)(unsigned long); //tasklet处理函数

unsigned long data; //处理函数参数

};

state: 只能在0，TASKLET\_STATE\_SCHED和TASKLET\_STATE\_RUN之间取值，TASKLET\_STATE\_SCHED表明已被调度，TASKLET\_STATE\_RUN表示正在运行确保同一个时刻只能在一个CPU上运行。

count：为0时才能激活运行，非0时表示被禁止

已经被调度TASKLET\_STATE\_SCHED的tasklet存放在两个单处理器数据结构：tasklet\_vec和tasklet\_hi\_vec，从下面声明就可以看到每个CPU都有一个这样的定义。该软中断触发时tasklet\_action和tasklet\_hi\_action就会从这个数据结构中获取tasklet进行执行。

static DEFINE\_PER\_CPU(struct tasklet\_head, tasklet\_vec);

static DEFINE\_PER\_CPU(struct tasklet\_head, tasklet\_hi\_vec);

通过tasklet\_schedule和tasklet\_hi\_schedule来调度一个tasklet。

（2）使用tasklet

A．声明一个tasklet

DECLARE\_TASKLET(name, func, data);

或者

tasklet\_init(t, tasklet\_handler, dev)

B．编写处理程序

void task\_handler(unsigned long data)

不能睡眠

C．调度tasklet

task\_schedule(&tasklet)

被调度以后，只要有机会就会尽可能早的执行，如果在执行之前又调度了一次，仍然只会执行一次。

## 3、工作队列

工作队列与软中断和tasklet有些不同，它是将拖后的任务交给一个内核线程去执行，因此总是会在进程上下文中执行。这样工作队列可以利用进程上下的优势，可以重新调度甚至睡眠。

工作队列子系统是一个用于创建内核线程的接口，通过它创建的线程负责执行由内核其他部分排到队列里的任务。这些线程称为工作者线程。

（1）数据结构

工作队列线程通过struct workqueue\_struct结构表示：

struct workqueue\_struct {

…

struct pool\_workqueue \_\_percpu \*cpu\_pwqs; /\* I: per-cpu pwqs \*/

struct pool\_workqueue \_\_rcu \*numa\_pwq\_tbl[]; /\* FR: unbound pwqs indexed by node \*/

}

数据成员struct pool\_workqueue \_\_percpu \*cpu\_pwqs代表每个CPU的工作队列池

struct pool\_workqueue {

…

struct work\_struct unbound\_release\_work;

struct rcu\_head rcu;

}

struct work\_struct unbound\_release\_work代表工作队列里的每一个工作项。

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

（2）工作队列使用

使用默认的工作队列：

A．创建工作项

#define DECLARE\_WORK(n, f)

or

#define INIT\_WORK(\_work, \_func)

B．工作项处理函数

void work\_handler(void \*data)

默认情况下，允许响应中断并且不持有任何锁，如果需要函数可以睡眠。使用锁的机制和其他进程上下文中使用锁机制一样。

C．调度执行工作项

schedule\_work(&work)

or

schedule\_delay\_work(&work, delay)延时一定时间后才会执行

cancel\_delayed\_work(&work)取消延迟

D．刷新操作

flush\_scheduled\_work()函数会一直等待，直到队列中所有对象都被执行才会返回，该函数会进入是休眠状态。

创建自己的工作队列：

如果缺省的队列不能满足你的需要，创建一个新的工作队列和与之相应的工作者线程，但是会在每个处理器上都创建一个工作者线程。

struct workqueue\_struct \*create\_workqueu(const char \*name)

创建之后可以调用下面列举的函数

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work)

int queue\_delayed\_work(struct workqueue\_struct \*wq, struct work\_struct \*work, unsigned long delay)

flush\_workqueue(struct workqueue\_struct \*wq)

功能和上面相似。

## 4、三者选择

软中断、tasklet和工作队列，前两者相似，工作队列机制不同靠内核线程实现：

|  |  |  |
| --- | --- | --- |
| 下半部 | 上下文 | 顺序执行保障 |
| 软中断 | 中断 | 无 |
| tasklet | 中断 | 同类型不能同时执行 |
| 工作队列 | 进程 | 无（同进程一样调度） |

软中断：

执行序列化保障最少。这要求软中断处理函数必须格外小心的采取一些步骤确保共享数据的安全，两个甚至更多相同类别的软中断有可能在不同处理器上同时执行，因此，最好可重入。它实时性最好。

tasklet：

两个相同类型的tasklet不能同时执行，实现简单。一种软中断而又不并发运行，因此驱动程序开发者应当尽可能选择tasklet而不是软中断。

工作队列：

如果需要把下半部推到进程上下文运行即可以睡眠，就只能工作队列。由于是内核线程，涉及进程切换，开销比上面两个大很多。

## 5、下半部控制

为了保证共享数据的安全，先得到一个锁，然后再禁止下半部的处理。

|  |  |
| --- | --- |
| 函数 | 描述 |
| local\_bh\_disable | 禁止本地处理器的软中断和tasklet的处理 |
| local\_bh\_enable | 激活本地处理器的软中断和tasklet的处理 |

需要成对调用，这些函数只影响软中断和tasklet，并不能禁止工作队列的执行。因为工作队列是在进程上下文中运行的，它保护共享数据所做的工作和其他任何进程上下文中做的都差不多。

# 四、中断源码分析

## 1、中断初始化

### （1）中断向量表映射

在Documentation/arm/memory.txt文档里说明了CPU中断向量映射后地址：00000000 ~ 00000fff 或 ffff0000 ~ fff0fff，具体哪一个有CP15寄存器的V bit决定，如果是1在ffff0000 ~ fff0fff，否则在00000000 ~ 00000fff。

在start\_kernel -> setup\_arch -> paging\_init -> devicemaps\_init中完成中断向量映射。

static void \_\_init devicemaps\_init(struct machine\_desc \*mdesc)

{

struct map\_desc map;

unsigned long addr;

void \*vectors;

//申请向量页

vectors = early\_alloc(PAGE\_SIZE);

//拷贝向量到申请的向量页

early\_trap\_init(vectors);

….

//完成向量页的地址映射

map.pfn = \_\_phys\_to\_pfn(virt\_to\_phys(vectors));

map.virtual = 0xffff0000;

map.length = PAGE\_SIZE;

map.type = MT\_HIGH\_VECTORS;

create\_mapping(&map, false);

//如果Vbit=0，就映射到low\_vector地址00000000 ~ 00000fff，否则就是上面的ffff0000 ~ fff0fff

if (!vectors\_high()) {

map.virtual = 0;

map.type = MT\_LOW\_VECTORS;

create\_mapping(&map, false);

}

…

}

//将向量表拷贝到申请的向量页

void \_\_init early\_trap\_init(void \*vectors\_base)

{

unsigned long vectors = (unsigned long)vectors\_base;

// vectors\_start:0xc05c5024, stubs\_start:0xc05c4e00，在arch/arm/kernel/entry-armv.S中定义，其中有个偏移计算（未弄明白）

extern char \_\_stubs\_start[], \_\_stubs\_end[];

extern char \_\_vectors\_start[], \_\_vectors\_end[];

extern char \_\_kuser\_helper\_start[], \_\_kuser\_helper\_end[];

int kuser\_sz = \_\_kuser\_helper\_end - \_\_kuser\_helper\_start;

vectors\_page = vectors\_base;

/\*

\* Copy the vectors, stubs and kuser helpers (in entry-armv.S)

\* into the vector page, mapped at 0xffff0000, and ensure these

\* are visible to the instruction stream.

\*/

memcpy((void \*)vectors, \_\_vectors\_start, \_\_vectors\_end - \_\_vectors\_start);

memcpy((void \*)vectors + 0x200, \_\_stubs\_start, \_\_stubs\_end - \_\_stubs\_start);

memcpy((void \*)vectors + 0x1000 - kuser\_sz, \_\_kuser\_helper\_start, kuser\_sz);

/\*

\* Do processor specific fixups for the kuser helpers

\*/

kuser\_get\_tls\_init(vectors);

/\*

\* Copy signal return handlers into the vector page, and

\* set sigreturn to be a pointer to these.

\*/

memcpy((void \*)(vectors + KERN\_SIGRETURN\_CODE - CONFIG\_VECTORS\_BASE),

sigreturn\_codes, sizeof(sigreturn\_codes));

flush\_icache\_range(vectors, vectors + PAGE\_SIZE);

modify\_domain(DOMAIN\_USER, DOMAIN\_CLIENT);

}

### （2）通用中断处理函数初始化

上面完成映射后，产生中断后就会跳转到相应的向量表入口，下面以irq为例。

\_\_stubs\_start:

/\*

\* Interrupt dispatcher

\*/

vector\_stub irq, IRQ\_MODE, 4

.long \_\_irq\_usr @ 0 (USR\_26 / USR\_32)

.long \_\_irq\_invalid @ 1 (FIQ\_26 / FIQ\_32)

.long \_\_irq\_invalid @ 2 (IRQ\_26 / IRQ\_32)

.long \_\_irq\_svc @ 3 (SVC\_26 / SVC\_32)

.long \_\_irq\_invalid @ 4

红色即为有效入口（为了对齐，其他都是非法入口），分别代表在用户空间和内核空间产生irq中断的入口，这里以\_\_irq\_svc为例。

.align 5

\_\_irq\_svc:

svc\_entry

irq\_handler //中断处理函数

#ifdef CONFIG\_PREEMPT

get\_thread\_info tsk

ldr r8, [tsk, #TI\_PREEMPT] @ get preempt count

ldr r0, [tsk, #TI\_FLAGS] @ get flags

teq r8, #0 @ if preempt count != 0

movne r0, #0 @ force flags to 0

tst r0, #\_TIF\_NEED\_RESCHED

blne svc\_preempt

#endif

svc\_exit r5, irq = 1 @ return from exception

UNWIND(.fnend )

ENDPROC(\_\_irq\_svc)

/\*

\* Interrupt handling.

\*/

.macro irq\_handler

#ifdef CONFIG\_MULTI\_IRQ\_HANDLER

ldr r1, =handle\_arch\_irq //最终的处理函数

mov r0, sp

adr lr, BSYM(9997f)

ldr pc, [r1]

#else

arch\_irq\_handler\_default

#endif

9997:

.endm

handle\_arch\_irq初始化过程如下：

start\_kernel -> init\_IRQ -> machine\_desc->init\_irq()(hi3536\_gic\_init\_irq) -> gic\_init -> gic\_init\_bases -> set\_handle\_irq(gic\_handle\_irq):

void \_\_init set\_handle\_irq(void (\*handle\_irq)(struct pt\_regs \*))

{

if (handle\_arch\_irq)

return;

handle\_arch\_irq = handle\_irq;

}

因此，handle\_arch\_irq就是上面的gic\_handle\_irq，最终就是跳转到该函数进行中断处理，具体执行见小节3。

### （3）中断域初始化

handle\_arch\_irq是任何体系结构产生中断的统一入口地址，中断域根据中断号又进行了中断划分，其初始化如下：

start\_kernel -> init\_IRQ -> machine\_desc->init\_irq()(hi3536\_gic\_init\_irq) -> gic\_init -> gic\_init\_bases -> irq\_domain\_add\_legacy:

gic->domain = irq\_domain\_add\_legacy(node, gic\_irqs, irq\_base, hwirq\_base, &gic\_irq\_domain\_ops, gic);

const struct irq\_domain\_ops gic\_irq\_domain\_ops = {

.map = gic\_irq\_domain\_map,

.xlate = gic\_irq\_domain\_xlate,

};

static int gic\_irq\_domain\_map(struct irq\_domain \*d, unsigned int irq,

irq\_hw\_number\_t hw)

{

if (hw < 32) {

irq\_set\_percpu\_devid(irq);

irq\_set\_chip\_and\_handler(irq, &gic\_chip, handle\_percpu\_devid\_irq);

set\_irq\_flags(irq, IRQF\_VALID | IRQF\_NOAUTOEN);

} else {

irq\_set\_chip\_and\_handler(irq, &gic\_chip, handle\_fasteoi\_irq);

set\_irq\_flags(irq, IRQF\_VALID | IRQF\_PROBE);

}

irq\_set\_chip\_data(irq, d->host\_data);

return 0;

}

当中断号<32时使用handle\_percpu\_devid\_irq；>=32时使用handle\_fasteoi\_irq进行处理，该分类个人认为是根据第二章小节2描述一节的PPI和SPI划分的。

## 2、中断注册

以下网卡驱动注册中断为例：

ret = request\_irq(SYNOP\_GMAC\_IRQNUM, stmmac\_interrupt, IRQF\_SHARED, STMMAC\_RESOURCE\_NAME, pdev)：

static inline int \_\_must\_check

request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags,

const char \*name, void \*dev)

{

return request\_threaded\_irq(irq, handler, NULL, flags, name, dev);

}

/\*

申请一个中断line，如果thread\_fn不等于NULL，则会创建一个线程。

申请中断资源，使能中断line和中断处理

\*/

int request\_threaded\_irq(unsigned int irq, irq\_handler\_t handler,

irq\_handler\_t thread\_fn, unsigned long irqflags,

const char \*devname, void \*dev\_id)

{

struct irqaction \*action;

struct irq\_desc \*desc;

int retval;

/\*

\* Sanity-check: shared interrupts must pass in a real dev-ID,

\* otherwise we'll have trouble later trying to figure out

\* which interrupt is which (messes up the interrupt freeing

\* logic etc).

\*/

if ((irqflags & IRQF\_SHARED) && !dev\_id)

return -EINVAL;

desc = irq\_to\_desc(irq); //根据中断号获取相应的中断域

if (!desc)

return -EINVAL;

if (!irq\_settings\_can\_request(desc) ||

WARN\_ON(irq\_settings\_is\_per\_cpu\_devid(desc)))

return -EINVAL;

if (!handler) {

if (!thread\_fn)

return -EINVAL;

handler = irq\_default\_primary\_handler;

}

action = kzalloc(sizeof(struct irqaction), GFP\_KERNEL);

if (!action)

return -ENOMEM;

//填充中断处理结构体

action->handler = handler;

action->thread\_fn = thread\_fn;

action->flags = irqflags;

action->name = devname;

action->dev\_id = dev\_id;

//将相应中断号的中断处理结构体添加到相应的中断域里

chip\_bus\_lock(desc);

retval = \_\_setup\_irq(irq, desc, action);

chip\_bus\_sync\_unlock(desc);

if (retval)

kfree(action);

return retval;

}

## 3、中断执行

这里主要讲解一下以irq中断产生的正常处理流程。中断产生后，CPU跳转到相应的入口地址：

\_\_stubs\_start:

/\*

\* Interrupt dispatcher

\*/

vector\_stub irq, IRQ\_MODE, 4

.long \_\_irq\_usr @ 0 (USR\_26 / USR\_32)

.long \_\_irq\_invalid @ 1 (FIQ\_26 / FIQ\_32)

.long \_\_irq\_invalid @ 2 (IRQ\_26 / IRQ\_32)

.long \_\_irq\_svc @ 3 (SVC\_26 / SVC\_32)

.align 5

\_\_irq\_svc:

svc\_entry

irq\_handler

#ifdef CONFIG\_PREEMPT

get\_thread\_info tsk

ldr r8, [tsk, #TI\_PREEMPT] @ get preempt count

ldr r0, [tsk, #TI\_FLAGS] @ get flags

teq r8, #0 @ if preempt count != 0

movne r0, #0 @ force flags to 0

tst r0, #\_TIF\_NEED\_RESCHED

blne svc\_preempt

#endif

svc\_exit r5, irq = 1 @ return from exception

UNWIND(.fnend )

ENDPROC(\_\_irq\_svc)

.macro irq\_handler

#ifdef CONFIG\_MULTI\_IRQ\_HANDLER

ldr r1, =handle\_arch\_irq（见初始化= gic\_handle\_irq）

mov r0, sp

adr lr, BSYM(9997f)

ldr pc, [r1]

#else

arch\_irq\_handler\_default

#endif

9997:

.endm

static asmlinkage void \_\_exception\_irq\_entry gic\_handle\_irq(struct pt\_regs \*regs)

{

u32 irqstat, irqnr;

struct gic\_chip\_data \*gic = &gic\_data[0];

void \_\_iomem \*cpu\_base = gic\_data\_cpu\_base(gic);

do {

irqstat = readl\_relaxed(cpu\_base + GIC\_CPU\_INTACK);

irqnr = irqstat & ~0x1c00;

if (likely(irqnr > 15 && irqnr < 1021)) {

irqnr = irq\_find\_mapping(gic->domain, irqnr); //获取中断号

handle\_IRQ(irqnr, regs);

continue;

}

if (irqnr < 16) {

writel\_relaxed(irqstat, cpu\_base + GIC\_CPU\_EOI);

/\*Call sig proccess func\*/

if ((irqnr == ipc\_irq\_handle.irq)

&& (ipc\_irq\_handle.handle)) {

ipc\_irq\_handle.handle(((irqstat >> 10) & 0x7),

irqnr, regs);

continue;

}

#ifdef CONFIG\_SMP

handle\_IPI(irqnr, regs);

#endif

continue;

}

break;

} while (1);

}

void handle\_IRQ(unsigned int irq, struct pt\_regs \*regs)

{

struct pt\_regs \*old\_regs = set\_irq\_regs(regs);

irq\_enter();

/\*

\* Some hardware gives randomly wrong interrupts. Rather

\* than crashing, do something sensible.

\*/

if (unlikely(irq >= nr\_irqs)) {

if (printk\_ratelimit())

printk(KERN\_WARNING "Bad IRQ%u\n", irq);

ack\_bad\_irq(irq);

} else {

generic\_handle\_irq(irq);

}

irq\_exit(); //退出中断，如果可能会调用软中断，处理下半部

set\_irq\_regs(old\_regs);

}

int generic\_handle\_irq(unsigned int irq)

{

struct irq\_desc \*desc = irq\_to\_desc(irq);

if (!desc)

return -EINVAL;

generic\_handle\_irq\_desc(irq, desc);

return 0;

}

static inline void generic\_handle\_irq\_desc(unsigned int irq, struct irq\_desc \*desc)

{

desc->handle\_irq(irq, desc);

}

这里的handle\_irq就是小节1中断域初始化的handle\_percpu\_devid\_irq或handle\_fasteoi\_irq，这里以handle\_fasteoi\_irq为例讲解：

void

handle\_fasteoi\_irq(unsigned int irq, struct irq\_desc \*desc)

{

raw\_spin\_lock(&desc->lock);

if (unlikely(irqd\_irq\_inprogress(&desc->irq\_data)))

if (!irq\_check\_poll(desc))

goto out;

desc->istate &= ~(IRQS\_REPLAY | IRQS\_WAITING);

kstat\_incr\_irqs\_this\_cpu(irq, desc);

/\*

\* If its disabled or no action available

\* then mask it and get out of here:

\*/

if (unlikely(!desc->action || irqd\_irq\_disabled(&desc->irq\_data))) {

desc->istate |= IRQS\_PENDING;

mask\_irq(desc);

goto out;

}

if (desc->istate & IRQS\_ONESHOT)

mask\_irq(desc);

preflow\_handler(desc);

handle\_irq\_event(desc);

if (desc->istate & IRQS\_ONESHOT)

cond\_unmask\_irq(desc);

out\_eoi:

desc->irq\_data.chip->irq\_eoi(&desc->irq\_data);

out\_unlock:

raw\_spin\_unlock(&desc->lock);

return;

out:

if (!(desc->irq\_data.chip->flags & IRQCHIP\_EOI\_IF\_HANDLED))

goto out\_eoi;

goto out\_unlock;

}

irqreturn\_t handle\_irq\_event(struct irq\_desc \*desc)

{

struct irqaction \*action = desc->action;

irqreturn\_t ret;

desc->istate &= ~IRQS\_PENDING;

irqd\_set(&desc->irq\_data, IRQD\_IRQ\_INPROGRESS);

raw\_spin\_unlock(&desc->lock);

ret = handle\_irq\_event\_percpu(desc, action);

raw\_spin\_lock(&desc->lock);

irqd\_clear(&desc->irq\_data, IRQD\_IRQ\_INPROGRESS);

return ret;

}

irqreturn\_t

handle\_irq\_event\_percpu(struct irq\_desc \*desc, struct irqaction \*action)

{

irqreturn\_t retval = IRQ\_NONE;

unsigned int flags = 0, irq = desc->irq\_data.irq;

do {

irqreturn\_t res;

trace\_irq\_handler\_entry(irq, action);

res = action->handler(irq, action->dev\_id); //此处执行的就是驱动中断注册的处理函数。

trace\_irq\_handler\_exit(irq, action, res);

…

retval |= res;

action = action->next; //个人认为：当该中断是共享的时候，遍历一遍所有的中断处理函数；如果不是共享的，则next应该为NULL。

} while (action);

add\_interrupt\_randomness(irq, flags);

if (!noirqdebug)

note\_interrupt(irq, desc, retval);

return retval;

}

通过是中断执行的整个完成代码流程，还有很多细节待进一步了解。