

## Обслуживание прерываний

### 1. Аппаратные прерывания

Схема обработки аппаратных прерываний — это принципиально архитектурно зависимое действие, связанное с непосредственным взаимодействием с контроллером прерываний. Но схема в основных чертах остаётся неизменной, независимо от архитектуры. Это связано со спецификой решаемой прерываниями задачи информирования процессора о асинхронном событии, возникающем в системе.

Аппаратные прерывания возникают от внешних устройств, являются в системе асинхронными событиями, которые возникают независимо от какой-либо выполняемой в системе работы, и их принято делить на следующие группы:

- Прерывание от системного таймера, которое возникает в системе периодически.
- Прерывания от устройств ввода-вывода. Возникают по инициативе устройства, когда устройству нужно сообщить процессору о завершении операции ввода-вывода.
- Прерывания от действий оператора, например, в ОС Windows при нажатии клавишей `ctrl_alt_del` для вызова task manager.

Аппаратные прерывания освобождают процессор от необходимости опрашивать внешние устройства с целью определения их готовности передать запрошенные процессом данные. Но требуют от системы выполнения последовательности действий по их обслуживанию. Пока данные не готовы процессор может выполнять какую-то другую работу. Но, когда поступает сигнал прерывания, процессор должен переключиться на его обслуживание.

В персональных компьютерах на базе процессоров Intel обработка аппаратных прерываний поддерживается аппаратно. Первой была микросхема Intel 8259 PIC, которая имела 8 входных линий (IRQ0-7), и одну выходную линию INTR (или просто INT). Сигналы прерывания от устройств ввода-вывода поступают на входы IRQ (Interrupt Request), а контроллер прерывания формирует сигнал прерывания, который по шине управления (линии INTR) поступает на соответствующую ножку (pin) процессора. Сигнал прерывания будет передан процессору, если он не замаскирован, т.е. его обработка разрешена. Для увеличения числа обрабатываемых прерываний контроллеры стали подключать в виде каскада: ведущий и ведомый контроллеры (всего 15 линий IRQ, одна линия используется для каскадного соединения). Этого было достаточно для систем с шиной ISA.

Обычно раскладка прерываний под устройства была более менее стандартная, например:  
IRQ 0 — system timer

IRQ 1 — keyboard controller  
 IRQ 2 — cascade (прерывание от slave контроллера)  
 IRQ 3 — serial port COM2  
 IRQ 4 — serial port COM1  
 IRQ 5 — parallel port 2 and 3 or sound card  
 IRQ 6 — floppy controller  
 IRQ 7 — parallel port 1  
 IRQ 8 — RTC timer  
 IRQ 9 — ACPI  
 IRQ 10 — open/SCSI/NIC  
 IRQ 11 — open/SCSI/NIC  
 IRQ 12 — mouse controller  
 IRQ 13 — math co-processor  
 IRQ 14 — ATA channel 1  
 IRQ 15 — ATA channel 2

На смену шине ISA пришла шина PCI. И количество устройств, требующих подключения, стало больше. Кроме того, в отличие от статической шины ISA шина PCI позволяла добавляться устройства в систему динамически. В данной шине прерывания могут быть разделяемыми (то есть к одной линии IRQ можно подсоединить несколько устройств). В итоге чтобы решить проблему нехватки линий IRQ, прерывания от PCI устройств стали группировать в линии PIRQ (Programmable Interrupt Request).

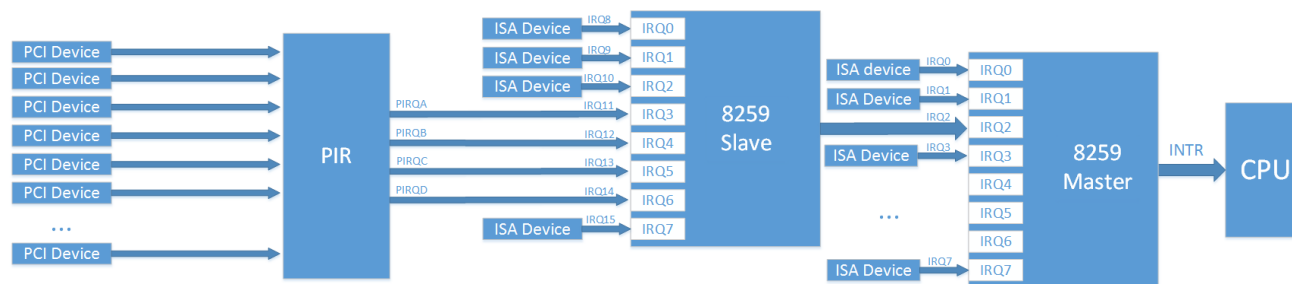


Рис.1

С появлением многопроцессорных систем PIC был заменен APIC (Advanced PIC), чтобы сбалансировать нагрузку по обработке прерываний на процессоры. Для каждого процессора добавляется специальный контроллер LAPIC (Local APIC) и для маршрутизации прерываний от устройств добавляется контроллер I/O APIC. Все эти контроллеры объединяются в общую шину с названием APIC (новые системы сейчас уже соединяются по стандартной системной шине).

Для обеспечения обратной совместимости со старыми системами первый 16 линий прерывания отводятся под старые прерывания ISA.

Шина PCI была заменена PCI express. Линии прерываний решили убрать. Чтобы сохранить совместимость, сигналы о возникновении прерываний (INTx#) эмулируются отдельными видами сообщений. В этой схеме логическое сложение линий прерываний, которое раньше производилось физическим соединением проводов, легло на плечи PCI мостов. Однако поддержка legacy INTx прерываний — это лишь поддержка обратной совместимости с шиной PCI. В PCI express был предложен новый метод доставки сообщений о прерываниях — MSI (Message Signaled Interrupts). В этом методе для сигнализации о прерывании устройство просто производит запись в MMIO область отведённую под LAPIC процессора.

Файл /proc/interrupts предоставляет таблицу о количестве прерываний на каждом из процессоров в следующем виде:

- Первая колонка: номер прерывания
- Колонки CPUx: счётчики прерываний на каждом из процессоров
- Следующая колонка: вид прерывания:
  - IO-APIC-edge — прерывание по фронту на контроллер I/O APIC
  - IO-APIC-fasteoi — прерывание по уровню на контроллер I/O APIC
  - PCI-MSI-edge — MSI прерывание
  - XT-PIC-XT-PIC — прерывание на PIC контроллер
  - Последняя колонка: устройство, ассоциированное с данным прерыванием

Так вот, как и положено в современной системе, для устройств и драйверов, поддерживающих прерывания MSI/MSI-X, используются именно они. Остальные прерывания «роутятся» ([англ. routing](#) - маршрутизация) через I/O APIC. Упрощённая схема роутинга прерываний показана на рис.2.

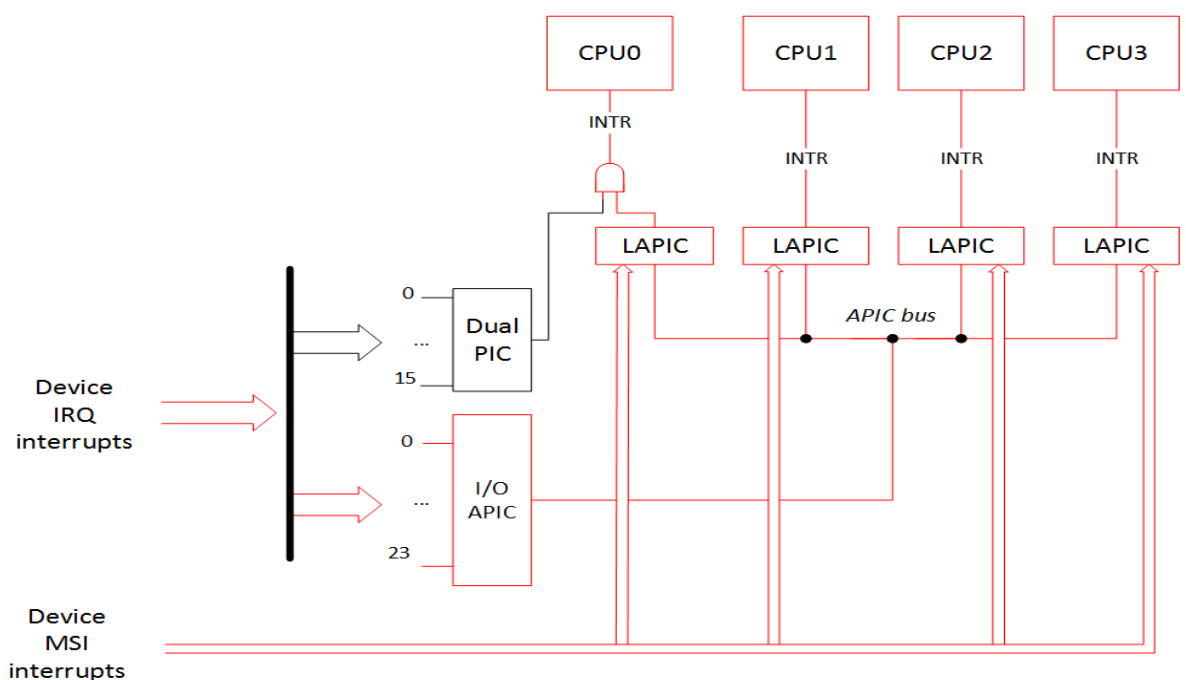


Рис.2

В многопроцессорных и многоядерных системах устройства получают возможность самостоятельно выбирать процессор/ядро для обработки конкретного прерывания, причем это делается полностью на уровне аппаратуры без программной поддержки. Это позволяет оптимизировать работу системы путем размещения большей части структур драйвера устройства и связанного с ним ПО, например, сетевых протоколов, в кэше конкретного процессора или же в его «ближней» NUMA-памяти<sup>1</sup>

## 2. Адресация обработчиков прерываний в защищенном режиме

Схема, представленная на рис.3, может рассматриваться как абстракция, позволяющая лаконично продемонстрировать адресацию обработчиков прерываний в защищенном режиме. Операции, выполняемые на единственном процессоре, рассматриваются только как тривиальный, вырожденный случай SMP (SMP - Symmetric MultiProcessing — симметричное мультипроцессирование).

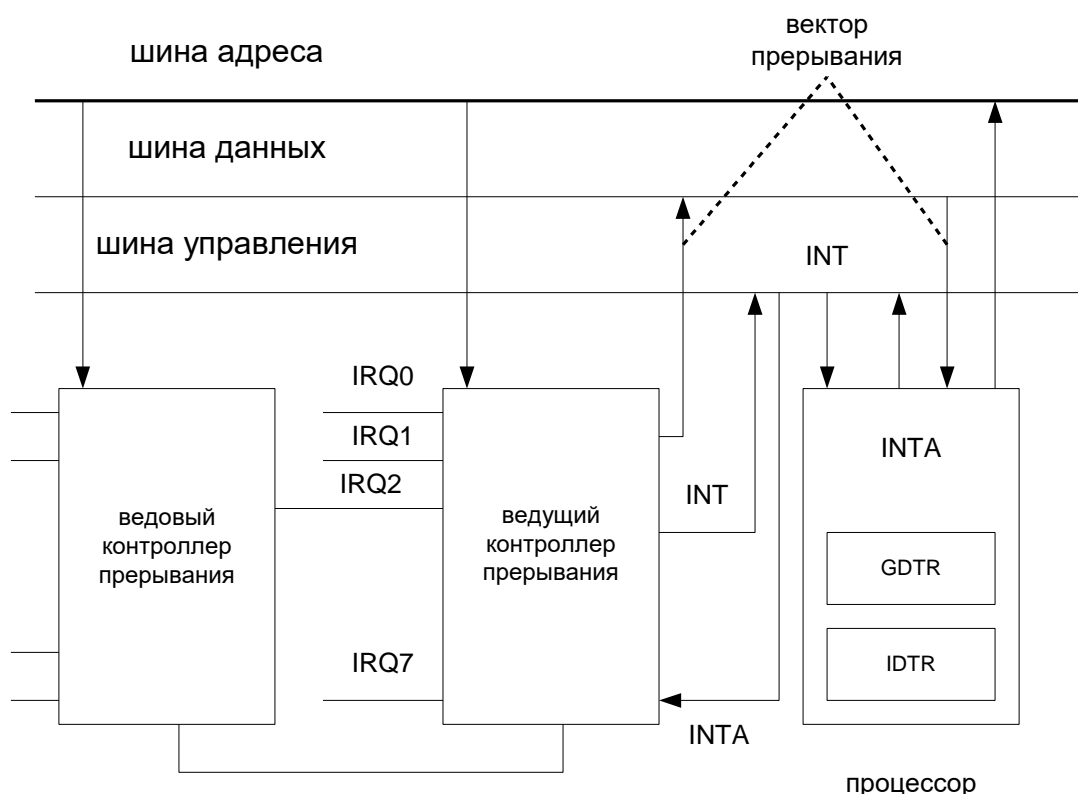


Рис.3

<sup>1</sup> NUMA – Non-Uniform Memory Access – «неравномерный доступ к памяти» или Non-Uniform Memory Architecture – архитектура с неравномерной памятью – схема реализации оперативной памяти, которая используется в мультипроцессорных системах, когда время доступа к памяти определяется ее расположением относительно процессора. Практически все архитектуры процессоров используют кэши. В NUMA поддержка когерентности кэша приводит к тому, что более один кэш может хранить содержимое одной и той же ячейки памяти. Системы с когерентностью кэша называются ccNUMA. Kernel Linux начиная с версии 2.5 содержит базовые NUMA. Версия 3.8 поддерживает новую NUMA и позволяет более эффективно развивать NUMA политику в следующих версиях.

Регистр GDTR содержит начальный адрес таблицы GDT – Global Descriptor Table – таблицы глобальных дескрипторов, которая в защищенном режиме содержит дескрипторы сегментов физической памяти.

Регистр IDTR содержит начальный адрес таблицы дескрипторов прерываний, которая содержит дескрипторы шлюзов прерываний, ловушек и задач.

Дескриптор глобальной таблицы дескрипторов имеет размер 8 байт и имеет следующие поля (рис.4):

1. Размер – limit – 2 байта
2. Начальный адрес сегмента – младшая часть – base\_low – 2 байта
3. Начальный адрес сегмента – средняя часть – base\_midle – 1 байт
4. Байт атрибутов – младший байт атрибутов – attr1
5. Байт атрибутов – старший байт атрибутов – attr2
6. Начальный адрес сегмента – старшая часть – base\_high – 1 байт

Таким образом, начальный адрес сегмента составляет 32 разряда в защищенном режиме.

Биты атрибутов 1 имеют следующее назначение:

Бит 0 - A – бит доступа к сегменту; устанавливается аппаратно при обращении к сегменту

Следующие три бита определяют тип сегмента:

Бит 1 – для сегмента кода, если бит сброшен, то чтение из сегмента запрещено, но это не относится к выборке команд, если установлен, то чтение разрешено;  
для сегмента данных, если бит сброшен, то модификация данных запрещена, если установлен, то модификация данных разрешена.

Бит 2 – для сегмента кода это – бит подчинения: если 0, то сегмент подчиненный, если 1, то сегмент кода обычный;  
для сегментов данных и стека: если 0, то это сегмент данных, если 1, то это -сегмент стека;

Бит 3 – бит предназначения: если он равен 0, то это – сегмент данных или стека, если равен 1, то это – сегмент данных.

-----  
Бит 4 - S - указывает, является ли дескриптор системным (0 - дескриптор системный).

Биты 5-6 - DPL – descriptor privilege level - привилегии дескриптора – определяет уровень привилегий (0 - ядро системы, 3 - пользовательский). В архитектуре x86 поддерживается 4 кольца защиты: от 0 до 3.

Бит 7 - P - present - признак присутствия сегмента в памяти (сегмент может быть выгружен менеджером виртуальной памяти).

Биты байта атрибутов 2 имеют следующие значения:

Биты 0-3 - limit – размер и с учетом двух первых байтов дескриптора – 0-го и 1-го – размер сегмента ограничен 20 битами ( $2^{20}=1024$  Кб или 1Мб)

Бит 4 - AVL –

Бит 5 - 0 – зарезервирован, а в 64-разрядных системах это – флаг L, который теперь служит признаком 64-разрядности сегмента. Если он установлен, флаг D/B должен быть сброшен.

Бит 6 - D/B - флаг, указывающий разрядность сегмента: 0 – сегмент 16-разрядный, 1 – сегмент 32 разрядный (этот флаг ещё называют BIG).

Бит 7 - G - бит гранулярности (Granularity) - указывает, в чём измеряется лимит (0 - в байтах, 1 - в страницах по 4 килобайта).

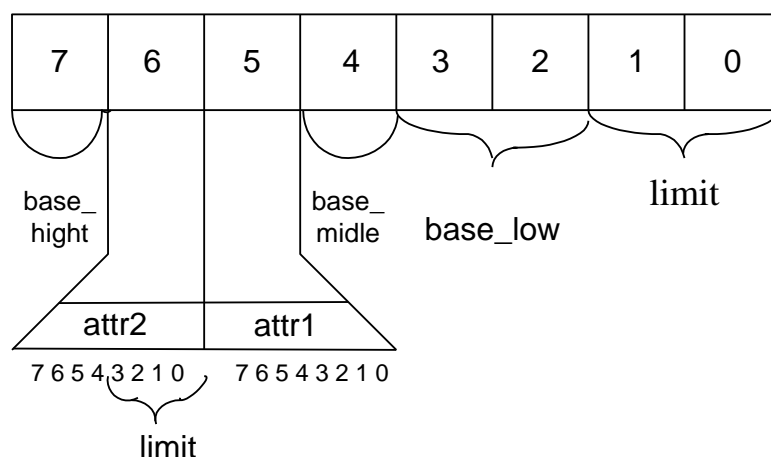


Рис.4 Дескриптор таблицы GDT

Дескриптор таблицы дескрипторов прерываний имеет следующий формат: размер дескриптора 8 байт, которые распределяются следующим образом (рис.5):

- Первые два байта и последние два байта – смещение – 32 бита.
- Байты 2 и 3 – селектор – 16 бит – используется для доступа к дескриптору сегмента, содержащего обработчик прерывания, в таблице GDT (смещение). Получив из дескриптора сегмента его начальный адрес и добавив смещение из дескриптора прерывания, получается адрес точки входа в обработчик прерывания.

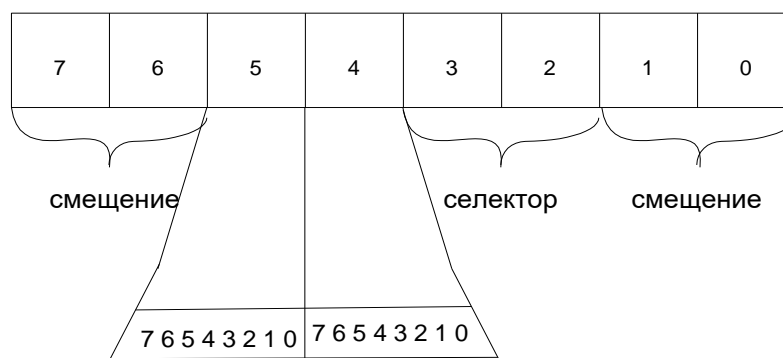


Рис.5 Дескриптор таблицы IDT

- Байт 4 – резерв
- Байт 5 – байт атрибутов:  
Биты 0, 1, 2, 3 определяют тип дескриптора

| №  | Значение<br>типа | Назначение   |
|----|------------------|--|
| 1  | 0                | Не определено  |
| 2  | 1                | Свободный сегмент задачи (TSS) 80286                           |
| 3  | 2                | LDT  |
| 4  | 3                | Занятый сегмент состояния задачи (TSS) 80286                   |
| 5  | 4                | Шлюзы вызова 80286   |
| 6  | 5                | Шлюз задачи (Task Gate)  |
| 7  | 6                | Шлюз прерываний 80286 (Interrupt Gate)                         |
| 8  | 7                | Шлюз ловушки 80286 (Trap Gate)                                 |
| 9  | 8                | Не определено  |
| 10 | 9                | Свободный сегмент состояния задачи (TSS) 80386, 80486 и старше |
| 11 | Ah               | Не определено  |
| 12 | Bh               | Занятый сегмент состояния задачи (TSS) 80386, 80486 и старше   |
| 13 | Ch               | Шлюз вызова 80386, 80486 и старше                              |
| 14 | Dh               | Не определено  |
| 15 | Eh-1110          | Шлюз прерываний (Interrupt Gate) 80386, 80486 и старше         |

|    |         |  |
|----|---------|--|
| 16 | Fh-1111 | Шлюз ловушки (Trap Gate) 80386, 80486 и старше |
|----|---------|--|

4 бит – 0 – не используется

Биты 5-6 – DPL - Уровень привилегий сегмента (**DPL** англ. - **Descriptor Privilege Level**) соответствует значению поля **DPL** в дескрипторе сегмента. Текущий уровень привилегий (CPL англ. ... **DPL** вызываемого сегмента не должен быть численно больше CPL;

**DPL** шлюза не должен быть численно меньше **DPL**, указанного в нём сегмента;

В шлюзе должен быть указан только сегмент кода или TSS (если это шлюз задачи).

Бит 7 – P – present – бит присутствия, если 0 – неиспользуемый вектор.

Для доступа к дескриптору прерывания в качестве смещения в таблице IDT используется вектор прерывания (рис. 3).

### 3. Обработчики прерываний

Обратимся к man(9).

```
#include <asm/irq.h>
#include <linux/signal.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
int
request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char *devname, void *dev_id);
```

Функция request\_irq() предназначена для регистрации обработчика прерывания на определенной линии прерывания.

Именно обработчики прерываний в драйверах устройств ответственны за взаимодействие с внешними устройствами на этапе передачи данных от устройств, полученных, например, в результате операции ввода-вывода. Драйвер устройства регистрирует в системе один обработчик прерывания. Делается это с помощью функции request\_irq(), которой в качестве параметра передается указатель на обработчик, обслуживающий конкретное прерывание: **irqreturn\_t (\*handler)(int, void \*)**. Эта функция вызывается, когда операционная система получает прерывание. Прототип обработчика принимает три параметра и возвращает значение типа **irqreturn\_t**. Тип **irqreturn\_t** определяется следующим образом:

include/linux/irqreturn.h, line 17 (as a typedef)



```

/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_IRQRETURN_H
#define _LINUX_IRQRETURN_H

/**
 * enum irqreturn
 * @IRQ_NONE      interrupt was not from this device or was not
handled
 * @IRQ_HANDLED   interrupt was handled by this device
 * @IRQ_WAKE_THREAD handler requests to wake the handler thread
 */
enum irqreturn {
    IRQ_NONE           = (0 << 0),
    IRQ_HANDLED        = (1 << 0),
    IRQ_WAKE_THREAD    = (1 << 1),
};

typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x) ((x) ? IRQ_HANDLED : IRQ_NONE)

#endif

```

Например,

```

static irqreturn_t intr_handler (int irq, void *dev) {
    if(! /* проверка того, что обслуживаемое устройство запросило
прерывание */)
        return IRQ_NONE;
    /* код обслуживания устройства */
    return IRQ_HANDLED;
}

```

Первым параметром функции **request\_irq()** является **unsigned int irq**, который определяет номер прерывания. Для некоторых устройств, например унаследованных (legacy) PC устройств таких, как таймер или клавиатура, это значение обычно жестко определено. Для большинства других устройств эта величина подбирается или назначается динамически и программно. Функция вызывается всякий раз, когда возникает прерывание с соответствующим значением irq.

Третий параметр – irqflags – может быть или нулем, или битовой маской одного или нескольких следующих флагов, которые используются только ядром как часть IRQ-обработчиков:

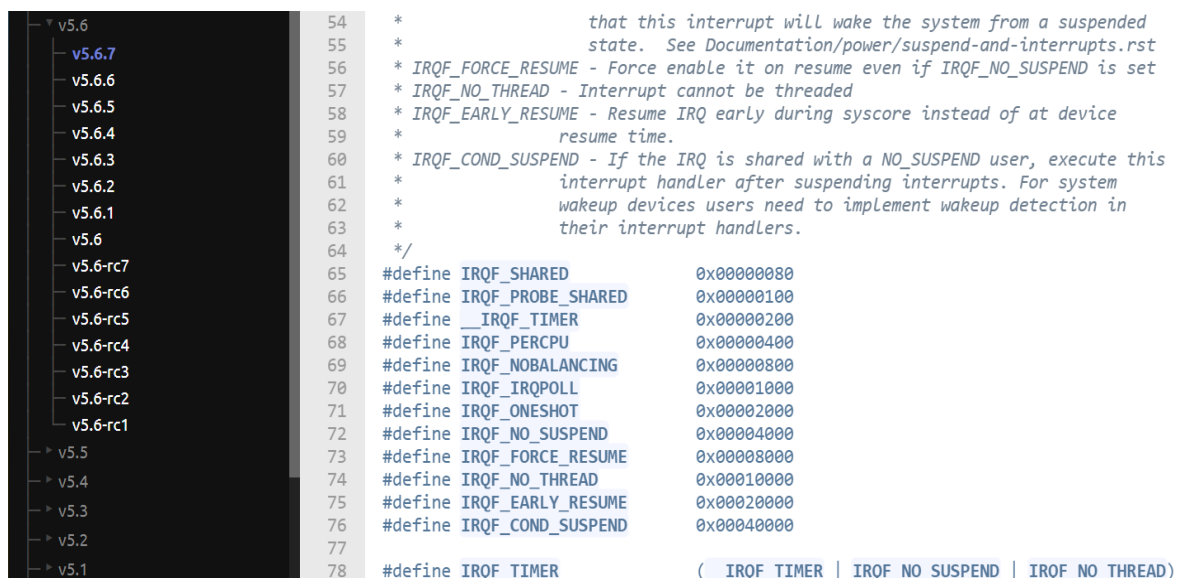
```

#define IRQF_SHARED 0x00000080 /*разрешить разделение IRQ несколькими
устройствами*/
#define IRQF_PROBE_SHARED 0x00000100 /*устанавливается вызывающими, когда они
ожидают, что произойдет несовпадение
при обмене*/
#define _IRQF_TIMER 0x00000200 /* прерывание помечается как прерывание по
таймеру.*/
#define IRQF_PERCPU 0x00000400 /*прерывание устанавливается на процессор*/
#define IRQF_NOBALANCING 0x00000800 /*флаг, чтобы исключить это прерывание из
балансировки irq*/

```

```
#define IRQF_IRQPOLL 0x00001000 /*прерывание используется для опроса (только
                                сообщение производительности, которое
                                зарегистрировано первым в общем прерывании,
                                рассматривается)*/
#define IRQF_ONESHOT 0x00002000 /*Прерывание не включается после завершения
                                работы обработчика hardirq. Используется потоковыми
                                прерываниями, которые должны держать линию irq
                                отключенной до тех пор, пока не будет запущен
                                потоковый обработчик*/
#define IRQF_NO_SUSPEND 0x00004000 /*не отключайте этот IRQ во время приостановки.
                                Не гарантирует, что это прерывание выведет систему из
                                приостановленного состояния. См. Documentation / power /
                                suspend-and-interrupts.txt*/
#define IRQF_FORCE_RESUME 0x00008000 /* принудительно включить его при
                                возобновлении, даже если установлен
                                IRQF_NO_SUSPEND*/
#define IRQF_NO_THREAD 0x00010000 /*Прерывание не может быть связано*/
#define IRQF_EARLY_RESUME 0x00020000 /* Возобновить IRQ на ранней стадии во время
                                syscore, а не во время возобновления работы
                                устройства*/
#define IRQF_COND_SUSPEND 0x00040000 /*если IRQ используется совместно с
                                пользователем NO_SUSPEND, запустите этот обработчик
                                прерываний после приостановки прерываний. Для
                                системных устройств пробуждения пользователи должны
                                реализовать обнаружение пробуждения в своих
                                обработчиках прерываний*/

#define IRQF_TIMER      ( __IRQF_TIMER | IRQF_NO_SUSPEND | IRQF_NO_THREAD )
```



```
54 *
55 *      that this interrupt will wake the system from a suspended
56 *      state. See Documentation/power/suspend-and-interrupts.rst
57 * IRQF_FORCE_RESUME - Force enable it on resume even if IRQF_NO_SUSPEND is set
58 * IRQF_NO_THREAD - Interrupt cannot be threaded
59 * IRQF_EARLY_RESUME - Resume IRQ early during syscore instead of at device
60 *      resume time.
61 * IRQF_COND_SUSPEND - If the IRQ is shared with a NO_SUSPEND user, execute this
62 *      interrupt handler after suspending interrupts. For system
63 *      wakeup devices users need to implement wakeup detection in
64 *      their interrupt handlers.
65 */
66 #define IRQF_SHARED          0x00000080
67 #define IRQF_PROBE_SHARED    0x00000100
68 #define __IRQF_TIMER         0x00000200
69 #define IRQF_PERCPU         0x00000400
70 #define IRQF_NOBALANCING     0x00000800
71 #define IRQF_IRQPOLL        0x00001000
72 #define IRQF_ONESHOT         0x00002000
73 #define IRQF_NO_SUSPEND      0x00004000
74 #define IRQF_FORCE_RESUME    0x00008000
75 #define IRQF_NO_THREAD       0x00010000
76 #define IRQF_EARLY_RESUME    0x00020000
77 #define IRQF_COND_SUSPEND    0x00040000
78 #define IRQF_TIMER           ( __IRQF_TIMER | IRQF_NO_SUSPEND | IRQF_NO_THREAD )
```

Следует отметить, что прототипы `irq_handler_t` и флаги установки обработчика существенно меняются от версии к версии ядра, например, радикально поменялись после 2.6.19 все флаги. Старое именованное `SA_*` изменено на `IRQF_*`.

`SA_INTERRUPT` был заменен на `__IRQF_TIMER`.

Четвертый параметр – `devname` – ASCII текстовое представление устройства, связанного с прерыванием. Например, для клавиатуры это – "keyboard". Это имя используется в `/proc/irq` и `/proc/interrupt`.

Пятый параметр – `dev_id` – используется прежде всего для разделения линий прерывания. Когда обработчик прерывания освобождается `dev_id` предоставляет уникальный файл cookie, позволяющий удалить с линии `irq` только соответствующий обработчик прерывания. Ядро без этого параметра не может узнать какой обработчик удалять с данной линии прерывания. Можно установить значение `NULL`, если линия прерывания не разделяется.

Следует отметить, что функция `request_irq()` может блокироваться и поэтому не может быть вызвана из контекста прерывания или в ситуации, когда блокировка небезопасна.

Когда драйвер выгружается, необходимо отменить регистрацию соответствующего обработчика прерывания и потенциально отключить линию прерывания. Это делается функцией:

```
void free_irq(unsigned int irq, void *dev_id);
```

Если указанная линия прерывания не является разделяемой, то функция `free_irq()` удаляет обработчик и отключает линию прерывания. Если линия прерывания разделяется, то обработчик, идентифицированный через `dev_id` удаляется, но сама линия прерывания будет отключена только при удалении последнего обработчика.

#### **4. Быстрые и медленные прерывания**

В ОС Linux принято различать быстрые и медленные прерывания. В ядрах до версии 2.6.19 для обозначения быстрых прерываний использовался флаг `SA_INTERRUPT`. В современных версиях такого флага нет и флаги обозначаются, как было показано ранее, `IRQF_*` и единственным быстрым прерыванием осталось прерывание от таймера `__IRQF_TIMER 0x00000200`.

Если говорить об аппаратных прерываниях, то они выполняются на высоких уровнях привилегий и их выполнение нельзя прервать. Быстрые прерывания это такие, которые можно обрабатывать очень быстро, тогда как обработка медленных прерываний занимает значительно больше времени. Быстрые прерывания выполняются при запрете всех прерываний на текущем процессоре. На других процессорах прерывания могут обрабатываться, но при запрете прерываний по линии `IRQ`, относящейся к выполняемому быстрому прерыванию. Таким образом, выполнение быстрого прерывания не может быть прервано.

Медленные прерывания могут потребовать значительных затрат процессорного времени. Очевидно, что запрет прерываний на длительное время может привести к целому ряду негативных явлений, связанных с отзывчивостью системы. Например, задачи, требующие быстрого внимания, могут не выполняться слишком долго.

После инициирования прерывания его обработчик должен выполняться как можно быстрее, чтобы не прерывать текущую активность на длительное время. Не все задачи по обработке прерывания можно выполнить за несколько инструкций. Например, обработка поступившего в сетевой адаптер пакета требует нескольких тысяч тактов прежде, чем пакет сможет быть передан в соответствующее адресное пространство режима пользователя. Очевидно, что такая обработка не может быть выполнена обработчиком прерывания. Чтобы сократить время выполнения обработчиков прерываний обработчики медленных аппаратных прерываний делятся на две части, которые традиционно называются верхняя (top) и нижняя (bottom) половины (half). Верхними половинами остаются обработчики, устанавливаемые функцией `request_irq()` на определенных IRQ. Выполнение нижних половин иницируется верхними половинами, т.е. обработчиками прерываний.

С современных ОС Linux имеется три типа нижних половин (bottom half):

- `softirq` – отложенные прерывания;
- `tasklet` – тасклеты;
- `workqueue` – очереди работ.

## 5. Нижние половины (bottom half)

### *Отложенные прерывания (softirq)*

Буквальный перевод `soft irq` – мягкие или гибкие `irq`, более соответствует их назначению, чем принятое – отложенные прерывания, так как все нижние половины выполняются как отложенные действия.

Гибкие прерывания определяются статически во время компиляции ядра. В файле `linux/interrupt.h` определена структура `struct softirq_action`, которая представляет одно отложенное действие:

```
enum action {Fail = 0, Ok = 1, Hold, Wait};
```

```
/* softirq mask and active fields moved to irq_cpustat_t in  
 * asm/hardirq.h to get better cache usage. KAO  
 */
```

```
struct softirq_action  
{
```

```
void (*action)(struct softirq_action *);
};
```

В старых версиях ядра в структуре было поле данных. В современных ядрах его нет.

Функция `open_softirq` заботится об инициализации `softirq`. Функция `open_softirq`, определенная в `kernel / softirq.c`:

```
static struct softirq_action
softirq_vec[NR_SOFTIRQS]
__cacheline_aligned_in_smp;
```

```
DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
```

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
    "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "IRQ_POLL",
    "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

Как видно из описания функции она получает два параметра: индекс элемента массива `softirq_vec` и указатель на выполняемую функцию `softirq`.

Определено перечисление типов гибких прерываний:

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq –
                 не используется, но сохраняется как последний
                 тип для сохранения нумерации*/

```

```
NR_SOFTIRQS /* количество softirq */
```

```
};
```

Все идентификаторы типов `softirqs` представлены в следующем массиве:

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
    "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK",
    "BLOCK_IOPOLL",
    "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

Или мы можем увидеть это в выводе / proc / softirqs:

```
~$ cat /proc/softirqs
```

|               | CPU0   | CPU1   | CPU2   | CPU3   | CPU4   | CPU5   | CPU6   |    |
|---------------|--------|--------|--------|--------|--------|--------|--------|----|
| HI:           | 5      | 0      | 0      | 0      | 0      | 0      | 0      |    |
| TIMER:        | 332519 | 310498 | 289555 | 272913 | 282535 | 279467 | 282895 | 27 |
| NET_TX:       | 2320   | 0      | 0      | 2      | 1      | 1      | 0      |    |
| NET_RX:       | 270221 | 225    | 338    | 281    | 311    | 262    | 430    |    |
| BLOCK:        | 134282 | 32     | 40     | 10     | 12     | 7      | 8      |    |
| BLOCK_IOPOLL: | 0      | 0      | 0      | 0      | 0      | 0      | 0      |    |
| TASKLET:      | 196835 | 2      | 3      | 0      | 0      | 0      | 0      |    |
| SCHED:        | 161852 | 146745 | 129539 | 126064 | 127998 | 128014 | 120243 | 11 |
| HRTIMER:      | 0      | 0      | 0      | 0      | 0      | 0      | 0      |    |
| RCU:          | 337707 | 289397 | 251874 | 239796 | 254377 | 254898 | 267497 | 25 |

Гибкие прерывания (softirqs) являются повторно входимыми - они могут выполняться параллельно на любом процессоре.

## Использованные и дополнительные источники

1. Linux Device Drivers, 2nd Edition [By Alessandro Rubini & Jonathan Corbet](#) 2nd Edition June 2001  
0-59600-008-1, Order Number: 0081, 586 pages
2. Цирюлик О.И. Модули ядра Linux. Внутренние механизмы ядра  
<http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-29.html>
3. Обслуживание периферии в коде модулей ядра: Часть 57. Регистрация обработчика прерываний  
Олег Цирюлик Опубликовано 30.05.2013  
[https://www.ibm.com/developerworks/ru/library/l-linux\\_kernel\\_57/](https://www.ibm.com/developerworks/ru/library/l-linux_kernel_57/)
4. Обслуживание периферии в коде модулей ядра: Часть 58. Создание "верхней половины" обработчика прерываний. **Олег Цирюлик** Опубликовано 04.06.2013  
[https://www.ibm.com/developerworks/ru/library/l-linux\\_kernel\\_58/](https://www.ibm.com/developerworks/ru/library/l-linux_kernel_58/)
5. Обслуживание периферии в коде модулей ядра: Часть 56. Обработка прерываний  
**Олег Цирюлик** Опубликовано 28.05.2013  
[https://www.ibm.com/developerworks/ru/library/l-linux\\_kernel\\_56/](https://www.ibm.com/developerworks/ru/library/l-linux_kernel_56/)

