Лекция 2.3 Модель драйверов Linux. Шина PCI. Шина platform

Разработали: Максимов А.Н., Крапивный А.В.

Версия 3.3_6

Содержание.

- Модель устройств в Linux;
- Общая информация о шине PCI;
- Подсистема PCI в Linux;
- Структура PCI драйвера;
- Работа с MSI прерываниями.

Высокоуровневые структуры модели устройств.

Bus - Шина в систем. Может содержать устройства. (PCI, USB ...)

Devices - Аппаратное устройство обнаруженное ядром и подключенное к одной из шин.

Drivers - Драйвер зарегистрированный в ядре.

Class - Тип устройв в системе (audio cards, network cards, graphics cards, and so on). Класс может содержать устройства подключенные к различным шинам.

sysfs

Sysfs — виртуальная файловая система располагающаяся в памяти.

Основное назначение: обеспечивать явное предоставление инфомации о структурах ядра (шины, драйвера, устройства и т.д.) и их атрибутах в пользовательском пространстве.

Директории в в sysfs соответсвуют структура kobject в модели устройств.

Дополнительная информация может быть найдена в Documentation/sysfs.txt

Атрибуты

Атрибуты предстявляются в sysfs в виде обычных текстовых ASCII файлов.

Sysfs перенаправляет операции ввода/вывода над атрибутами и позволяет ядру читать и писать атрибуты.

Определиени атрибута:

```
struct attribute {
        char                    * name;
        struct module           *owner;
        mode_t                  mode;
};
```

Функции создания/удаления атрибута:

```
int sysfs_create_file(struct kobject * kobj, const struct attribute * attr);
void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);
```

(для абстрактоного атрибута функции чтения/записи не определены)

Атрибуты для устройства(device)

Возможно добавление атрибутов для объектов модели устройств типа device

Тип данных для атрибута устройства:

```
struct device_attribute {
   struct attribute   attr;
   ssize_t (*show)(struct device *dev, struct device_attribute *attr, char *buf);
   ssize_t (*store)(struct device *dev, struct device_attribute *attr, const char *buf,
     size_t count);
};
```

Функции для создания:

```
int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

Определенов в include/linux/device.h

Атрибуты для устройства(device). п.2

Для облегчения создания утрибутов устойств определен макрос:

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show,
    _store)
```

Пример.

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

Это эквивалентно:

```
static struct device_attribute dev_attr_foo = {
    .attr    = {
        .name = "foo",
        .mode = S_IWUSR | S_IRUGO,
        .show = show_foo,
        .store = store_foo,
    },
};
```

Пример.

```
// Определение функций чтения и записи параметров
int _port_reg;
static ssize_t foo_port_reg_show(struct device *dev,    struct device_attribute *attr, char *buf){
    return sprintf(buf, "%x\n", _port_reg);
}
static ssize_t foo_port_reg_set(struct device *dev,    struct device_attribute *attr,const char *buf, size_t
      count) {
    _port_reg=simple_strtoul(buf, NULL, 0);
    return count;
}
static DEVICE_ATTR(port_reg,0644,foo_port_reg_show,foo_port_reg_set);
int rtl8139_probe (struct pci_dev *pdev, const struct pci_device_id *id) {
   res = device_create_file(&pdev->dev,&dev_attr_port_reg);
  return 0;
}
```

simple_strtoul

simple_strtoul — преобразует string в unsigned long

Определение функции:

unsigned long simple_strtoul (const char *cp, char **endp, unsigned int base);

Аргументы:

cp - начало строки

endp - указатель на конец разбираемой строки

base - база числа

Атрибуты для драйвера устройства.

Структура для представления атрибута:

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *, char * buf);
        ssize_t (*store)(struct device_driver *, const char * buf,
                        size_t count);
};
```

Определение:

```
DRIVER_ATTR(_name, _mode, _show, _store)
```

Создание/уничтожение:

```
int driver_create_file(struct device_driver *, const struct driver_attribute *);
void driver_remove_file(struct device_driver *, const struct driver_attribute *);
```

Определено в include/linux/device.h

PCI обзор.

PCI (Peripheral component interconnect) - системная шина для подключения периферийных устройств.

Стандарт на шину PCI определяет:

- физические параметры (например, разъёмы и разводку сигнальных линий);
- электрические параметры (например, напряжения);
- логическую модель (типы циклов шины, адресацию на шине и т.д.);

Развитием стандарта PCI занимается организация PCI Special Interest Group. Дополнительную информаицию о PSI SIG можна найти на www.pcisig.com

Шины семейства PCI.

PCI это высокоскоростная шина для обмена между CPU и устройствами в.в. Спецификация PCI позвоялет перадавать 32 бита данных по параллельной шине с частотой 33 или 66 Мгц. Пиковая пропускная способность шины 266 Mbs.

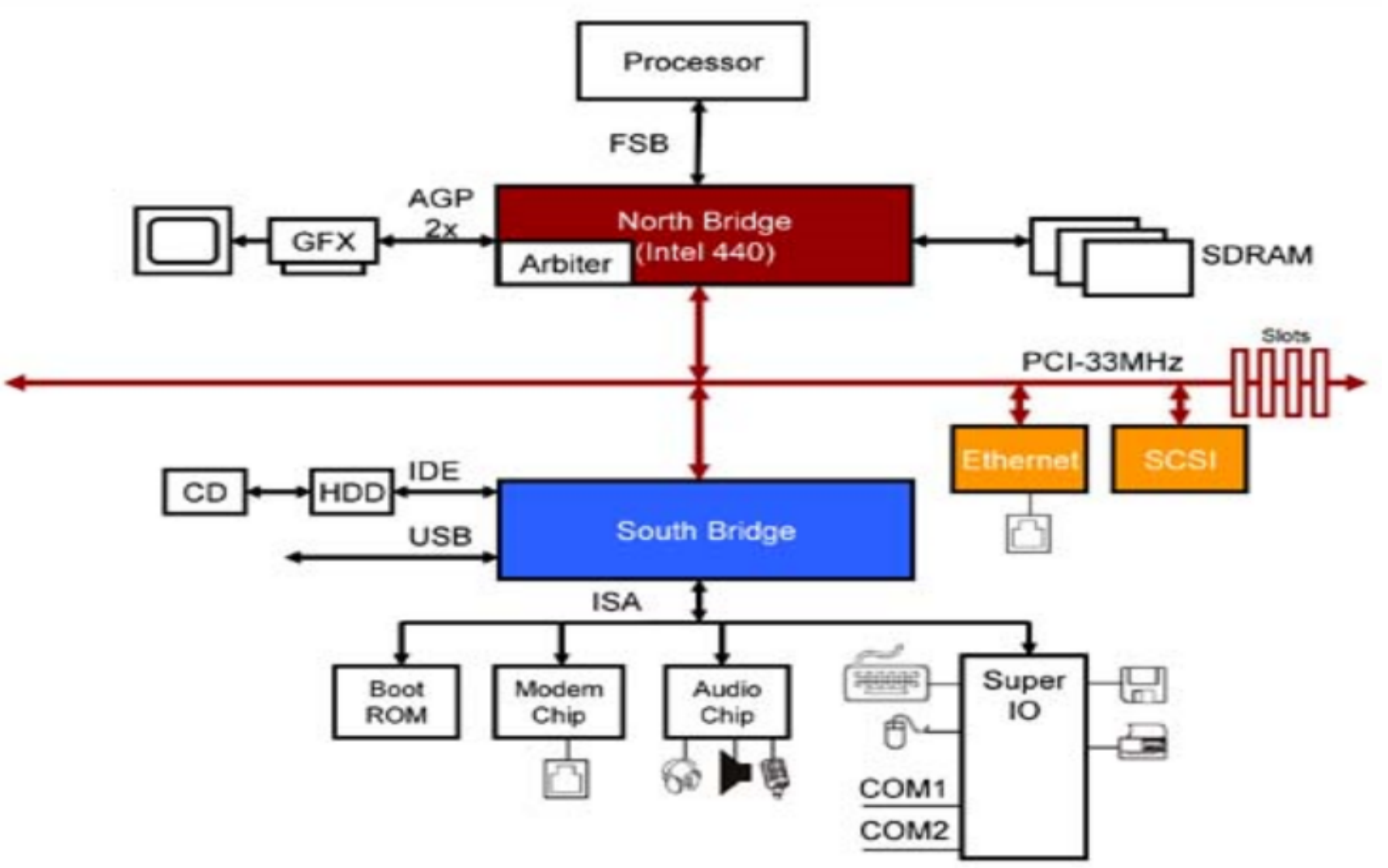
CardBus логически и электрически представляет собой полноценную 32-разрядную шину PCI, работающую на частоте 33 МГц, механические размеры и разъёмы позаимствованы у шины PCMCIA. Устройства CardBus могут поддерживать DMA.

Mini PCI версия шины PCI для использования в ноутбуках стандартизированная в рамках стандарта PCI версии 2.2. Использует 32-разрядную шину PCI, работающую на частоте 33 МГц. PCI устройства могут быть подключены к шине mini PCI через переходник.

PCI-X (PCI eXtended) 64 разрядное расширение шины PCI для использования в сервеных приложениях. Разработана в 1998 IBM, HP и Compaq. Частота шины может различноной от 66 (PCI-X версия 1.0) до 533 МГц(PCI-X версия 2.0). Пропускная способность от 1.06 Гб/с в первоначальной реализации до 4.3 Гб/с у версии 2.0.

PCI Express (PCIe или PCI-E) дальнейшее развитие технологии PCI. PCIe использует последовательную шину для передачи данных. PCIe поддерживает до 32 последовательных линков. Каждый PCIe лину имеет пропускную способность до 250Мб/с в каждом направлении обеспечивая до 8Гб/с в каждом направлении. Текущая ваеряис спецификации - PCIe 2.0.

Типичная архитектура системы с PCI



Работа с PCI

- Linux предоставляет специальные функции для определения куда физически отображается область памяти
- Функции шины PCI (Peripheral Component Interconnect) определены в стандарте на шину
- Устройство на шине идентифицируется парой значений:

```
#define VENDOR_ID    0x1039
#define DEVICE_ID 0x6325
```

Работа с PCI. Области памяти

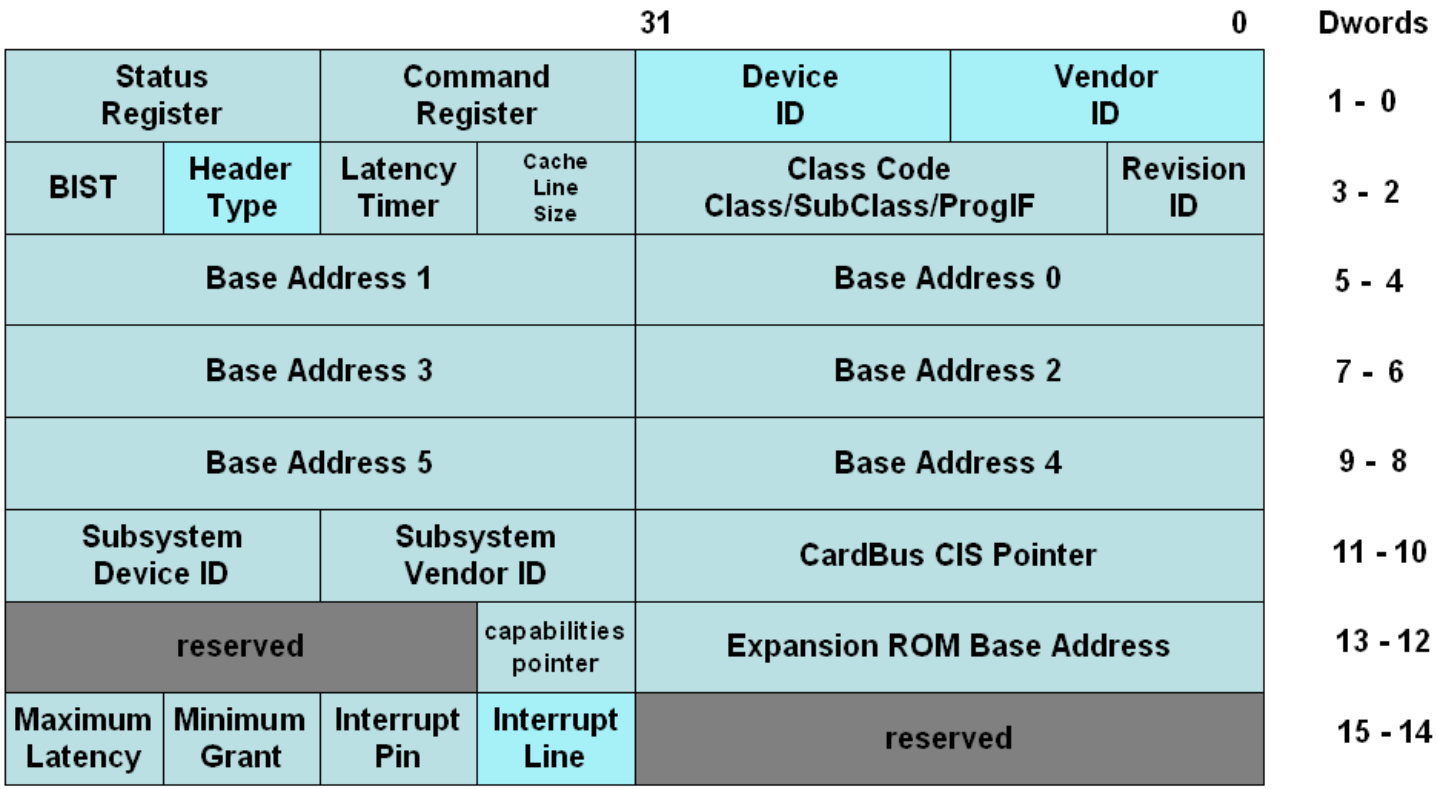
- CPU и устройство на шине PCI обмениваются информацией через общую память. Обычно разделяемая память содержит регистры команд и статусные регистры Переферийные устройства обладают собственной памятью. CPU может обращаться к этой памяти. Доступ устройства к памяти системы осуществляется при помощи механизма DMA

- Для шины PCI есть три области адресов:
  - PCI Configuration
  - PCI I/O
  - PCI Memory

Работа с PCI. Конфигурационная область

PCI Configuration

32 байта



Назначение некоторых полей

Stauts register — регистр статуса

Command Register — управляет способностью устройства генерировать и реагировать на PCI циклы

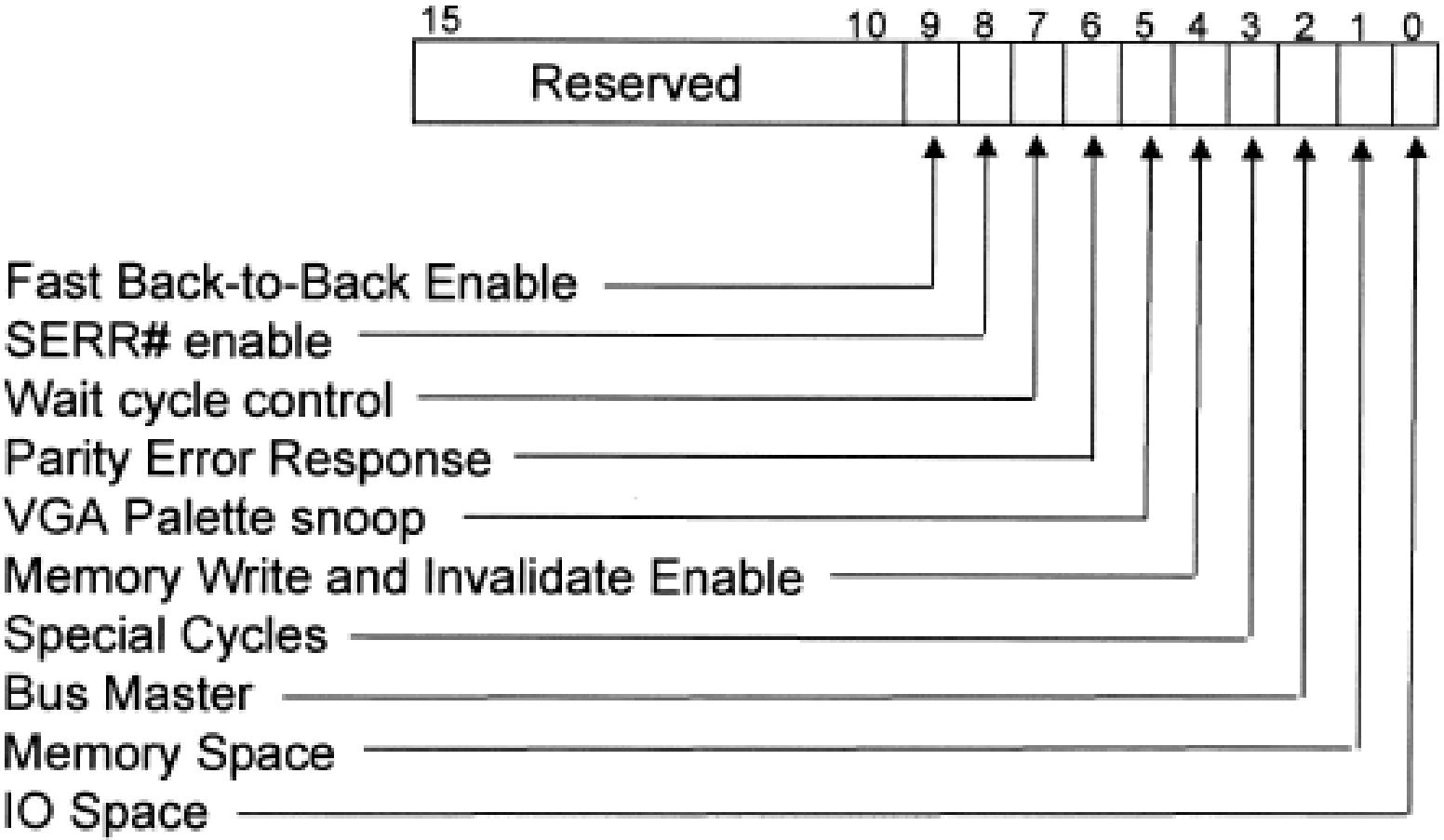
Vendor ID, Device ID — идентифицируют, тип устройства

Header Type — определяет тип заголовка. Type 0 — заголовок для большинства устройств. Type 1 — заголовок для устройств типа PIC бридж, Type 2 для устройств типа PC Card.

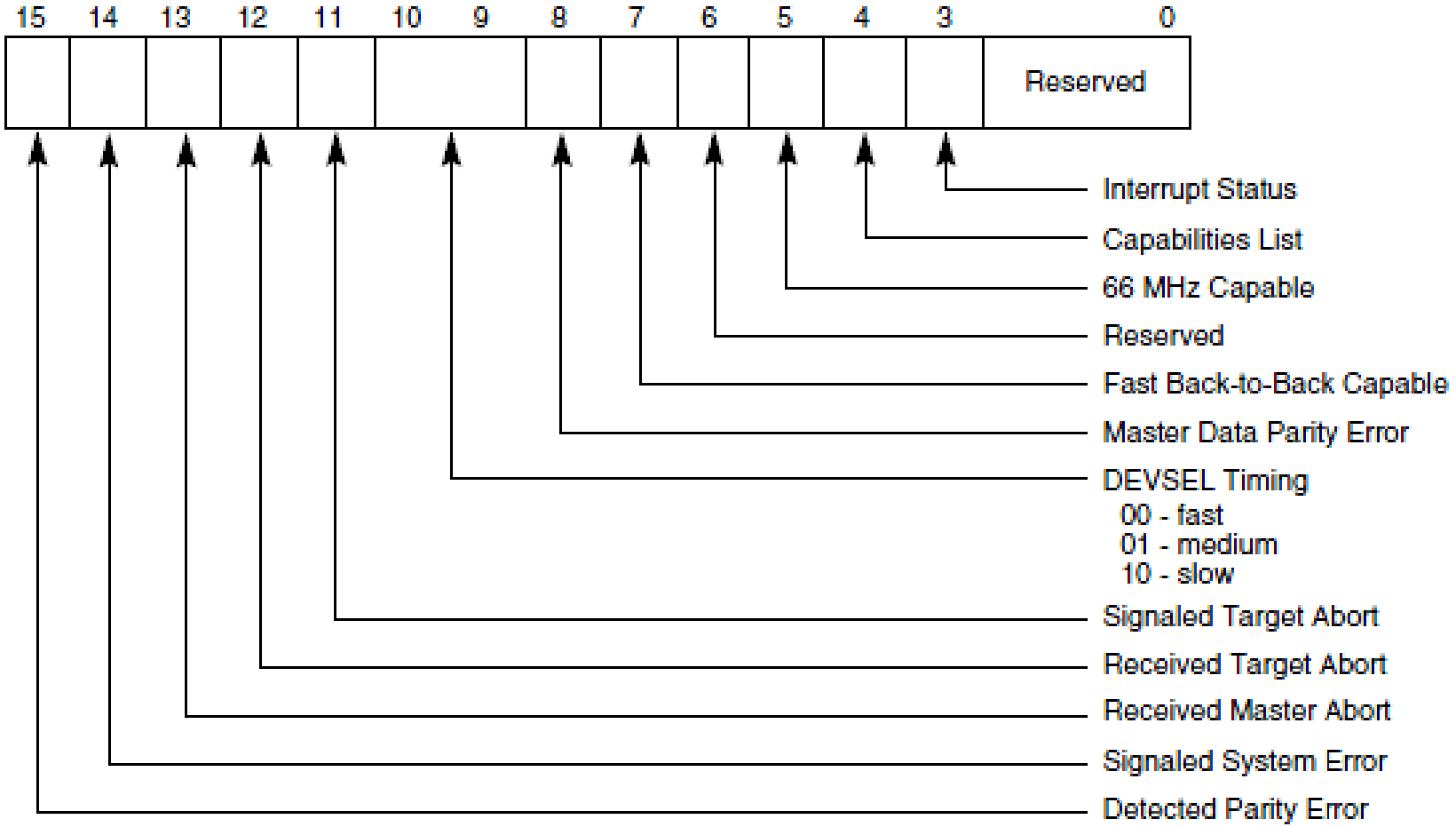
Base addres register (BAR) — адрес области памяти или портов в.в.

Interrupt line — номер прерывания

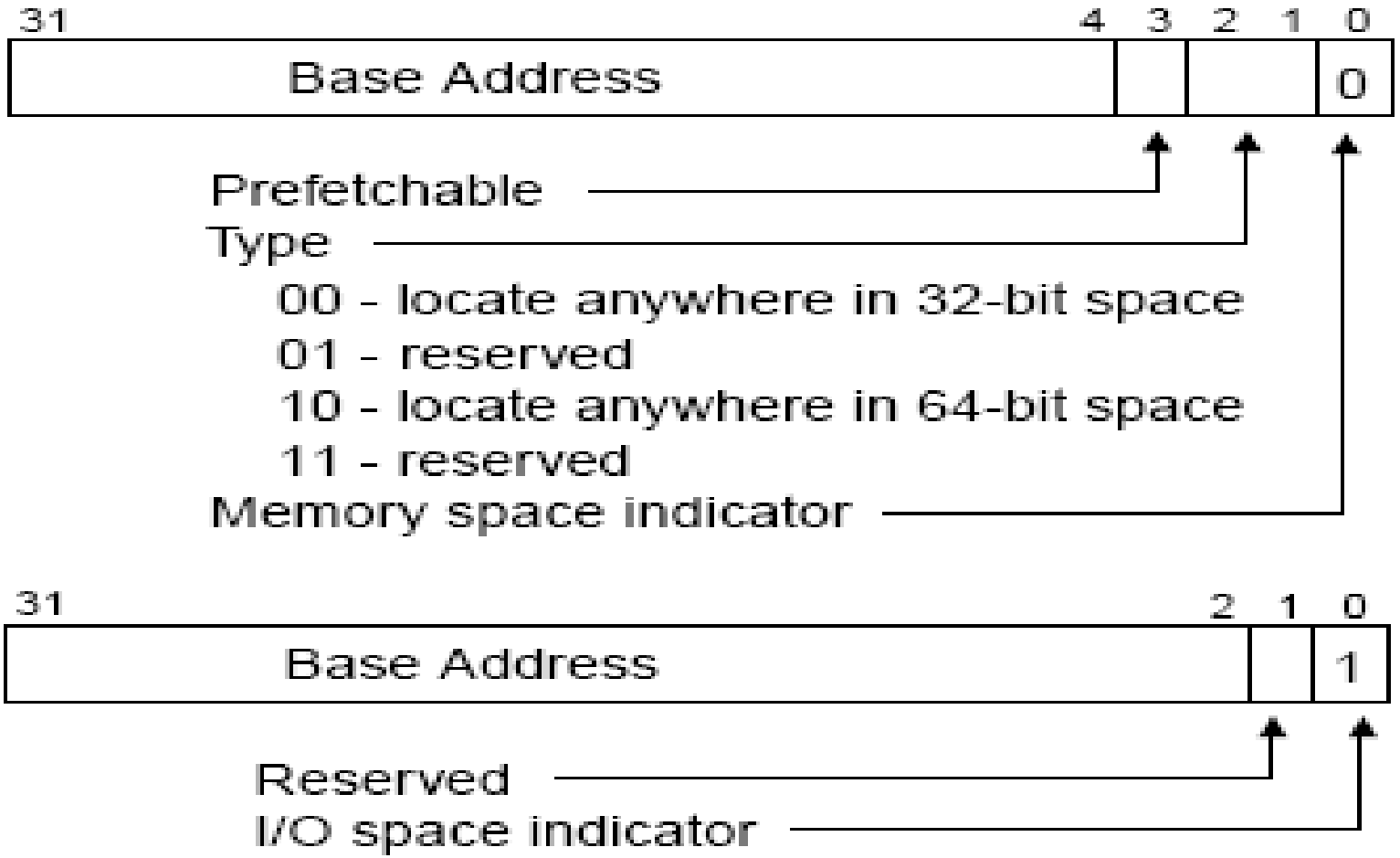
Command Register



Status Register



Base Address Register



Пример области конфигурации

PCI Config. Register Address (Hex)	Register Name			
0x00	PCI Device ID = 0x7820		PCI Vendor ID = 0x1435	
0x04	PCI Status		PCI Command	
0x08	PCI Class Code = 0x118000			PCI Rev. ID = 0x00
0x0C	PCI BIST	PCI Header Type	Bus Latency Timer	Cache Line Size
0x10	PCI Base Address Register 0: Memory Access to PLX9056 Registers			
0x14	PCI Base Address Register 1: I/O Access to PLX9056 Registers			
0x18	PCI Base Address Register 2: Memory Access to Digital I/O Registers			
0x1C	PCI Base Address Register 3: Reserved			
0x20	Reserved			
0x24	Reserved			
0x28	Reserved			
0x2C	PCI Subsystem ID = 0x9056		PCI Subsystem Vendor = 0x10B5	
0x30	Reserved			
0x34	Reserved			Reserved
0x38	Reserved			
0x3C	PCI Max Latency	PCI Min Grant	PCI Interrupt Pin	PCI Interrupt Line

RISoft

Нахождение устройства на шине PCI

```
struct pci_dev {
        // contains many fields
        };
struct pci_dev  *devp = NULL;
devp = pci_find_device( VID, DID, devp );
```

Необходимо подключить #include <linux/pci.h>

Нахождение устройства на шине PCI

```
int  probe  (struct pci_dev *dm7820_pci, const struct       pci_device_id *id){
...
    printk ("Get physics BAR0...");
  dm7820_PCIMem.real = pci_resource_start (dm7820_pci,0);
  dm7820_PCIMem.size = pci_resource_len   (dm7820_pci,0);
  if ((dm7820_PCIMem.real==0)||(dm7820_PCIMem.size==0)) {printk ("failed.\n"); return -1;} else  printk
("%u...OK.\n",(uint32)dm7820_PCIMem.real);
  printk ("Checks physics BAR0...");
  if (pci_resource_flags (dm7820_pci,0)&IORESOURCE_MEM) printk ("OK.\n"); else {printk
("failed.\n"); return -1;}
  printk ("Get virtual BAR0...");
  dm7820_PCIMem.virtual=ioremap_nocache (dm7820_PCIMem.real,dm7820_PCIMem.size);
  if (dm7820_PCIMem.virtual==0) {printk ("failed.\n"); return -1;} else  printk ("%u...OK.\n",
(uint32)dm7820_PCIMem.virtual);
  printk ("Request region BAR0...\t\t");
  if (request_mem_region (dm7820_PCIMem.real,dm7820_PCIMem.size,DM7820_NAME)) printk
("OK.\n"); else {printk ("failed.\n"); return -1;}
```

Нахождение устройства на шине PCI

```
  printk ("Get physics BAR1...");
  dm7820_PCIIO.real = pci_resource_start (dm7820_pci,1);
  dm7820_PCIIO.size = pci_resource_len   (dm7820_pci,1);
  if ((dm7820_PCIIO.real==0)||(dm7820_PCIIO.size==0)) {printk ("failed.\n"); return -1;} else  printk ("OK.\n");
  printk ("Checks physics BAR1...");
  if (pci_resource_flags (dm7820_pci,1)&IORESOURCE_IO) printk ("OK.\n"); else {printk ("failed.\n"); return -1;}
  printk ("Request region BAR1...");
  if (request_region (dm7820_PCIIO.real,dm7820_PCIIO.size,DM7820_NAME)) printk ("OK.\n"); else {printk
("failed.\n"); return -1;}

  printk ("Get physics BAR2...");
  dm7820_FPGAMem.real = pci_resource_start (dm7820_pci,2);
  dm7820_FPGAMem.size = pci_resource_len   (dm7820_pci,2);
  if ((dm7820_FPGAMem.real==0)||(dm7820_FPGAMem.size==0)) {printk ("failed.\n"); return -1;} else  printk
("OK.\n");
  printk ("Checks physics BAR2...");
  if (pci_resource_flags (dm7820_pci,2)&IORESOURCE_MEM) printk ("OK.\n"); else {printk ("failed.\n"); return
-1;}
  printk ("Get virtual BAR2...");
  dm7820_FPGAMem.virtual=ioremap_nocache (dm7820_FPGAMem.real,dm7820_FPGAMem.size);
  if (dm7820_FPGAMem.virtual==0) {printk ("failed.\n"); return -1;} else  printk ("OK.\n");
  printk ("Request region BAR2...");
  if (request_mem_region (dm7820_FPGAMem.real,dm7820_FPGAMem.size,DM7820_NAME)) printk ("OK.\n");
else {printk ("failed.\n"); return -1;}

  printk ("Set IRQ...");
  status=request_irq(dm7820_pci->irq,dm7820_handler,SA_SHIRQ,DM7820_NAME,dm7820_pci);
  if (status!=0) {printk ("failed.\n"); return -1;} else  printk ("%i...OK.\n",dm7820_pci->irq);
```

RISoft

Практический пример.

.

Опубликовать через файловую систему proc информацию о mmio_base и mmio_size.

Для realtek 8139

```
#define VENDOR_ID     0x10EC// ReakTek Semiconductors Corp
#define DEVICE_ID 0x8139              // RTL-8139 Network Controller
```

Модель устройств.

Модель устройств linux позволяет решать следующие задачи:

- Согласованное управление питанием;
- Поддержку plug-and-play;
- Поддержку Hot-plug support;
- Взаимодействеи с пространством пользователя;
- Контроль времени жизни объектов;

На верхнем уровне модели устройств можно выделить следующие структуры данных: bus, device, driver, class .

Структуры верхнего уровня базируются на низкоуровневых структурах:

kobject, ktype, kset, subsustem

Интерфейс модели устройств с пользовательским уровнем осуществляется при помощи файловой системы sysfs file system.

Модель устройств. Bus PCI

Шина PCI в каталоге /sys/bus/pci/ имеет две директории : 'devices' and 'drivers'.

PCI bus сопоставляет устройства сравнивая PCI Device ID для всех устройств и драйверов.

```
struct bus_type pci_bus_type = {
  .name       = "pci",
  .match      = pci_bus_match,
  .uevent     = pci_uevent,
  .probe      = pci_device_probe,
  .remove     = pci_device_remove,
  .shutdown   = pci_device_shutdown,
  .dev_attrs  = pci_dev_attrs,
  .pm         = PCI_PM_OPS_PTR,
};
```

(*linux-4.6.28\drivers\pci\pci-driver.c*)

Драйвер PCI

pci driver должен выполнить следующие задачи в init_module:

- Определить struct pci_driver
- Инициализировать struct pci_driver structure
- Зарегистрироватьstruct pci_driver pci_register_driver

Структура struct pci_driver включает структуру device_driver для регистрации в модели устройств .

```
struct pci_driver {
    struct list_head node;
    char *name;
    const struct pci_device_id *id_table;

    int  (*probe)  (struct pci_dev *dev, const struct
    pci_device_id *id);
     void (*remove) (struct pci_dev *dev);
      int  (*suspend) (struct pci_dev *dev,
pm_message_t state);
      int  (*suspend_late) (struct pci_dev *dev,
pm_message_t state);
      int  (*resume_early) (struct pci_dev *dev);
       int  (*resume) (struct pci_dev *dev);
          void (*shutdown) (struct pci_dev *dev);

      struct pm_ext_ops *pm;
      struct pci_error_handlers *err_handler;
      struct device_driver    driver;
      struct pci_dynids dynids;
};
```

(*linux-2.6.28\include\linux\pci.h*)

Функции pci_driver

int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);

Вставить устройство.

void (*remove) (struct pci_dev *dev);

Удалить устройство (NULL если устройство не HOT plug)

int (*suspend) (struct pci_dev *dev, pm_message_t state); /*Усыпить*/

int (*suspend_late) (struct pci_dev *dev, pm_message_t state);

int (*resume_early) (struct pci_dev *dev);

int (*resume) (struct pci_dev *dev); /*Пробудить*/

void (*shutdown) (struct pci_dev *dev);

Пример.

```
#include <linux/pci.h>
static struct pci_device_id rtl8139_pci_tbl[] = {
      {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
      {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
     {0,}
};
MODULE_DEVICE_TABLE (pci, rtl8139_pci_tbl)

static struct pci_driver rtl8139_pci_driver = {
      .name       = DRV_NAME,
      .id_table   = rtl8139_pci_tbl,
      .probe      = rtl8139_init_one,
      .remove     = __devexit_p(rtl8139_remove_one),
};


static int __init rtl8139_init_module (void) {
   return pci_register_driver(&rtl8139_pci_driver);
}
```

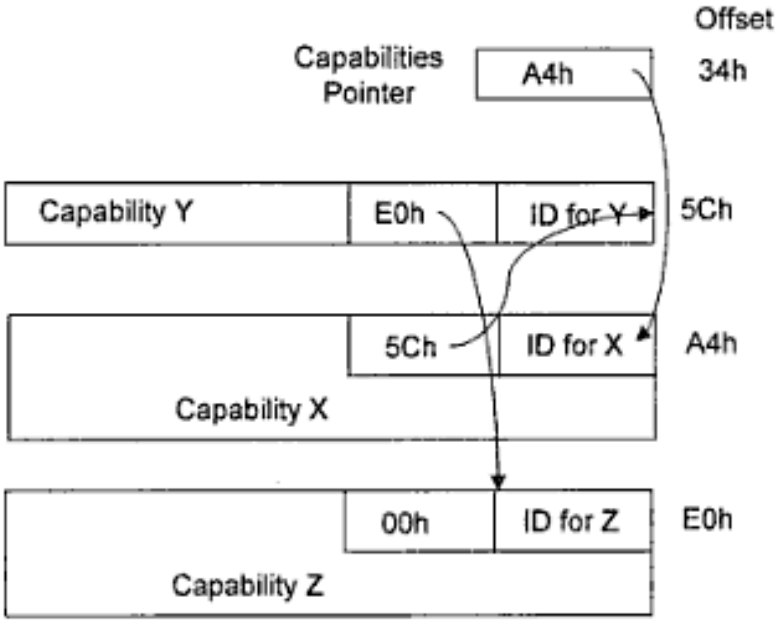
MSI прерывания

Message Signaled Interrupts (MSI) прерывания является опциональной функцией для устройст удовлетворяющих спецификации PCI версии 2.3 и более поздних и обязательной функцией для устройств стандарта PCI Express.

Для сигнализации запроса прерывания устройство запрашивает управление шиной и, получив его, посылает сообщение.
Сообщение выглядит как обычная запись двойного слова в ячейку памяти, адрес и шаблон сообщения на этапе конфигурирования устройств записываются в конфигурационные регистры устройства.
В сообщении старшие 16 бит всегда нулевые, а младшие 16 бит несут информацию об источнике прерывания.

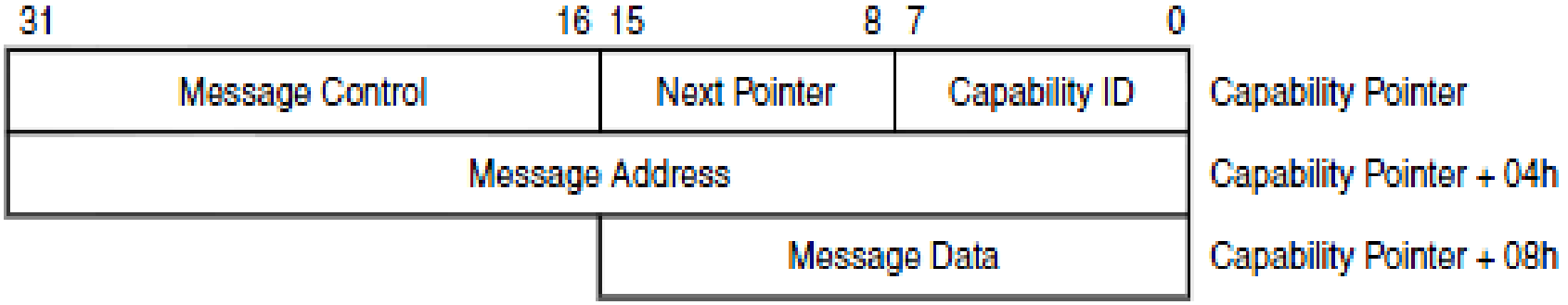
Как оределить наличе MSI прерывания.

Включение дополнительныех возможностей, в спецификации PCI 2.2 добавляются путев вставки групп регисторв в связанный список, который называется список свойств (Capabilities List).
Наличие этой структуры сигнализируется 4-м битом (Capabilities List bit) в регистре статуса PCI. Наличие этого бита указыает на существование указателя на список свойств по смещению 34h. Добавляемые «фичи» имеют ID и указатель на следующую. Для MSI ID=0x05

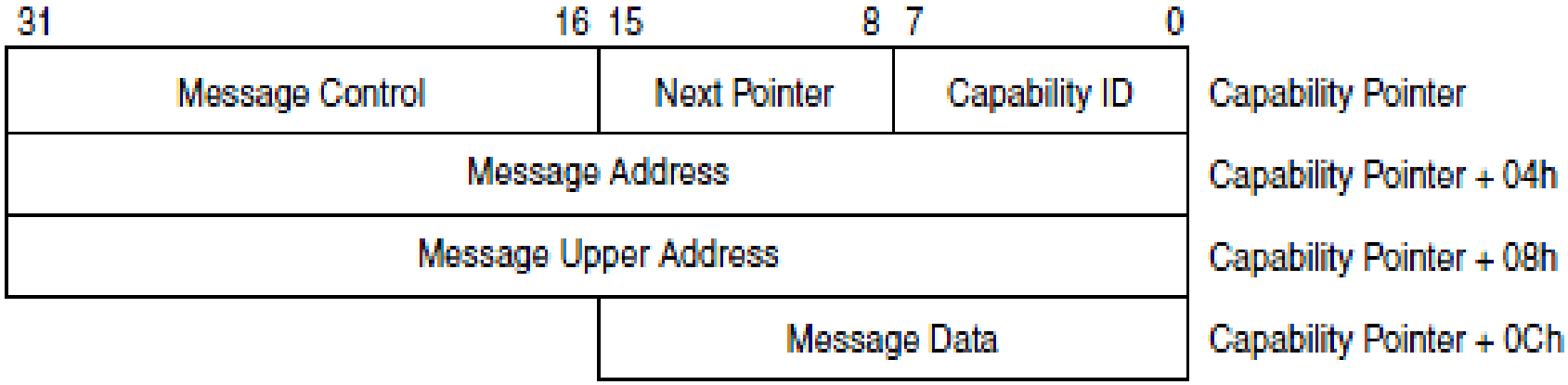


MSI Capability Structure

Capability Structure for 32-bit Message Address



Capability Structure for 64-bit Message Address



MSI прерывания

Message Signaled Interrupts (MSI) прерывания является опциональной функцией для устройст удовлетворяющих спецификации PCI версии 2.3 и более поздних и обязательной функцией для устройств стандарта PCI Express.

Для сигнализации запроса прерывания устройство запрашивает управление шиной и, получив его, посылает сообщение.
Сообщение выглядит как обычная запись двойного слова в ячейку памяти, адрес и шаблон сообщения на этапе конфигурирования устройств записываются в конфигурационные регистры устройства.
В сообщении старшие 16 бит всегда нулевые, а младшие 16 бит несут информацию об источнике прерывания.

Разрешение MSI прерывания в ядре Linux

Для того, чтобы разрешить Message Signaled Interrupts (MSI) прерывания в ядре linux при концигурации необходимо указать следующий ключ:

CONFIG_PCI_MSI (Bus options\Message signaled interrupts)

Разрешить генерацию MSI прерываний:
int pci_enable_msi(struct pci_dev *dev)

Запретить использвание MSI прерываний:
void pci_disable_msi(struct pci_dev *dev)

Разрешение MSI прерывания в ядре Linux

Разрешить генерацию MSI прерываний:
int pci_enable_msi(struct pci_dev *dev)

- Вызов выделяет одно MSI прерывание.
- Устройство настраивается на генерацию MSI прерываний.
- Меняется значение dev->irq на номер MSI прерывания.
- Необходимо вызвать до вызова request_irq().

(для выделения нескольких MSI прерываний следует использовать pci_enable_msi_block)

Запретить использвание MSI прерываний:
void pci_disable_msi(struct pci_dev *dev)

- До вызова этой функции должно быть вызвана free_irq()
- Меняется значение dev->irq на номер стандартного прерывания.

Проверка разрешениы ли MSI прерывания

Для того, чтобы проверить разрешены ли MSI прерывания можно воспользоваться следующей командой:

lspci -v

Существуют несколько возможности по которым MSI прерывания не доходят:
- материнская плата не работает с MSI корректно и они запрещены;
- MSI прерывания запрещены ниже PCI моста к которому подключено устройство;
- MSI прерывания запрещены на самом устройстве;

Назначение шины platform

Шина platform предназначена для выполненеия следующих задач:

- Интеграции устройств на встраиваемых системах в модель устройств не оснащенными

шинами с поддеркой технологий Plug&Play, Hot plug и идентификацию устройств

- Шина platform является псевдо шиной и предоставляет интерфейс platform driver / platform device во встраиваемых систьемах

- Устройства представляемые в виде platform device обычно напрямую подключены к центральному процессору

Ключевые сущности при работе с шиной platform

Основыне используемые структуры данных при работе с platform:

platform_bus

platfrom_driver

platform_device

resource

Пример реализации драйвера

Драйвер для последовательного порта Beagle Board (TI OMAP3530) определяется как platform_driver:

```
static struct platform_driver omap_mcbsp_driver = {
        .probe          = omap_mcbsp_probe,
        .remove             = __devexit_p(omap_mcbsp_remove),
        .driver         = {
                .name   = "omap-mcbsp",
        },
};
```

см. arch\arm\plat-omap\mcbsp.c

Регистрация и исключение platform_driver

Для регистрации и исключения из системы используются функции platform_driver_register и platform_driver_unregister определенные в linux/platform_device.h.

Пример.

```
int __init omap_mcbsp_init(void)
{
    /* Register the McBSP driver */
    return platform_driver_register(&omap_mcbsp_driver);
}
```

platform_device

- Структура данных platform_device описывает одно устроство;
- platform_device описывает конфигурацию устройств (для описания конфигурации используется структура resource);
- Определение platform_device производится в аппаратно зависимом коде инициализации платы (может быть статическим или динимическим);
- Шина platform производит сопоставление между устройством и дравером по полю name, которое должно быть уникальным.

platform_device

Структура данных platform_device описана в файле include\linux\platform_device.h

```
struct platform_device {
        const char      * name;
        int             id;
        struct device   dev;
        u32             num_resources;
        struct resource      * resource;
        struct platform_device_id    *id_entry;
        struct pdev_archdata   archdata; /* arch specific additions */
};
```

Пример определения platform_device

Пример статического определения platform_device:

```
static struct platform_driver omap_mpuio_driver = {
        .driver         = {
                .name   = "mpuio",
                .pm = &omap_mpuio_dev_pm_ops,
        },
};

static struct platform_device omap_mpuio_device = {
        .name           = "mpuio",
        .id         = -1,
        .dev = {
                .driver = &omap_mpuio_driver.driver,
        }
        /* could list the /proc/iomem resources */
};
```

Пример определения platform_device

Динамическое определение структуры platform_device производится при момощи следующих функции:

```
struct platform_device * platform_device_alloc (   const char *  name,   unsigned int      id);
```

Регистрация устройства платформы производится при помощи следующих функций:

```
int platform_device_register (struct platform_device *  pdev);     или
int platform_add_devices (struct platform_device ** devs,int num);
```

Пример:

```
// Декларирование устройств доступных на платформе
static struct platform_device *davinci_evm_devices[] __initdata = {
        &dm355evm_dm9000,
        &davinci_nand_device,
};
static __init void dm355_evm_init(void)  {
…
  platform_add_devices(davinci_evm_devices,ARRAY_SIZE(davinci_evm_devices));

}
```

Определени конфигурации машины

Пример определения конфигурации машины

```
MACHINE_START(DAVINCI_DM355_EVM, "DaVinci DM355 EVM")
        .phys_io      = IO_PHYS,
        .io_pg_offst  = (__IO_ADDRESS(IO_PHYS) >> 18) & 0xfffc,
        .boot_params  = (0x80000100),
        .map_io          = dm355_evm_map_io,
        .init_irq     = dm355_evm_irq_init,
        .timer           = &davinci_timer,
        .init_machine = dm355_evm_init,
MACHINE_END
```

Определени устройства.

Определение устройства:

```
static struct platform_device da8xx_edma_device = {
        .name                   = "edma",
        .id             = -1,
        .dev = {
                .platform_data   = da8xx_edma_info,
        },
        .num_resources          = ARRAY_SIZE(da8xx_edma_resources),
        .resource = da8xx_edma_resources,
};
```

Регистрация устройства:

```
platform_device_register(&da8xx_edma_device);
```

arch\arm\mach-davinci\devices-da8xx.c

Определени ресурсов устройства.

```
static struct resource da8xx_edma_resources[] = {
        {
                .name  = "edma_cc0",
                .start   = DA8XX_TPCC_BASE,
                .end     = DA8XX_TPCC_BASE + SZ_32K - 1,
                .flags   = IORESOURCE_MEM,
        },
        {
                .name  = "edma_tc0",
                .start   = DA8XX_TPTC0_BASE,
                .end     = DA8XX_TPTC0_BASE + SZ_1K - 1,
                .flags   = IORESOURCE_MEM,
        },
        {
                .name  = "edma_tc1",
                .start   = DA8XX_TPTC1_BASE,
                .end     = DA8XX_TPTC1_BASE + SZ_1K - 1,
                .flags   = IORESOURCE_MEM,
        },
        {
                .name  = "edma0",
                .start   = IRQ_DA8XX_CCINT0,
                .flags   = IORESOURCE_IRQ,
        },
        {
                .name  = "edma0_err",
                .start   = IRQ_DA8XX_CCERRINT,
                .flags   = IORESOURCE_IRQ,
        },
};
```

arch\arm\mach-davinci\devices-da8xx.c

Получение адресов и номера прерываний

Для получени адреса и номера прерывания могут быть использованы специальные функции:

int **platform_get_irq_byname** (struct platform_device * dev,char * name);

struct resource   * **platform_get_resource_byname** (struct platform_device * dev,

unsigned int type, char * name);

Пример получение номеро прерывания:

```
sprintf(irq_name, "edma%d_err", j);
err_irq[j] = platform_get_irq_byname(pdev, irq_name);
```

Пример получения адреса ввода вывода:

```
sprintf(res_name, "edma_cc%d", j);
r[j] = platform_get_resource_byname(pdev, IORESOURCE_MEM,  res_name);
len[j] = resource_size(r[j]); // Определена в arch\arm\mach-davinci\ioport.h
                              // Возвращаемое значение = res->end - res->start + 1;
r[j] = request_mem_region(r[j]->start, len[j],dev_name(&pdev->dev));
```

Литература.

1. Driver development. Kernel architecture for device drivers. http://free-electrons.com
2. Documentation / driver-model / device.txt
3. Documentation\driver-model\platform.txt
4. Documentation\filesystems\sysfs.txt

Литература

1. **The MSI Driver Guide HOWTO**
   http://devresources.linuxfoundation.org/dev/robustmutexes/src/fusyn.hg/Documentation/MSI-HOWTO.txt
2. **Шины PCI, USB и FireWire. Энциклопедия**
3. **Doug Abbott PCI bus demystified**
4. **The MSI Driver Guide HOWTO.**
   http://www.mjmwired.net/kernel/Documentation/PCI/MSI-HOWTO.txt
5. **Интересные факты об устройстве подсистемы PCI**
   http://www.linux-mips.org/wiki/PCI_Subsystem