

Jądro 2.6 – programowanie jądra i sterowników

TECHNIKA JĄDROWA

W ostatnim odcinku „Techniki jądrowej” omawialiśmy znakowe sterowniki urządzeń wirtualnych. W dzisiejszym odcinku pokazujemy, jak za pomocą jądra 2.6 obsługiwać prawdziwy sprzęt komputera PC, zwłaszcza urządzenia PCI.

EVA-KATHARINA KUNST I JÜRGEN QUADE

Składniki sprzętowe mogą się zepsuć, a ich zachowanie z czasem może także podlegać pewnym wahaniom. Dlatego w systemie operacyjnym, w którym stawia się na stabilność, takim jak Linux, dostęp do zasobów sprzętowych jest zawsze sprawą delikatną. Jednak również w jądrze 2.6, tak jak w wersji 2.4, programowanie własnych sterowników sprzętu pozostaje proste. Nawet wejście-wyjście i przestrzeń adresowa pamięci procesora są dostępne naprawdę w łatwy sposób dzięki zestawowi makr czy funkcji (zobacz Rysunek 1).

Procesor uzyskuje dostęp do adresów w obszarze wejścia-wyjścia, do portów, przy użyciu osobnych poleceń wejścia i wyjścia. Aby zachować przenośność jądra, polecenia te są zamknięte w makrach. Jeśli aplikacja żąda danych ze sprzętu, funkcja *read* najpierw uruchamia odpowiednią funkcję *DriverRead* (zobacz [1]) wewnątrz sterownika. Dopiero kiedy autor sterownika wprowadzi specjalne makro w funkcji *DriverRead*, dochodzi do faktycznego przepływu danych ze sprzętu.

Następnie makra *inb*, *inw* i *inl* odczytują z portu 8, 16 lub 32 bity, które dostały jako parametr. Zapis na wejściu-wyjściu sprzętu należy do makr *outb*, *outw* i *outl*; są one ulokowane w funkcji *DriverWrite*. Kierują one 8- („b”), 16- („w”) lub 32-bitową („l”) wartość, którą dostały jako pierwszy parametr,

pod adres portu przekazany jako drugi parametr.

Na Listingu 1 w wierszu 24 posługujemy się odmianą takiego makra: *outb_p*. Literka „p” oznacza pauzę. W tej postaci, a istnieje ona dla wszystkich poleceń portów, po każdym dostępie do sprzętu nastąpi krótka przerwa, aby dostęp nie następował zbyt szybko. Niektóre urządzenia nie mogą poradzić sobie ze zbyt szybką transmisją.

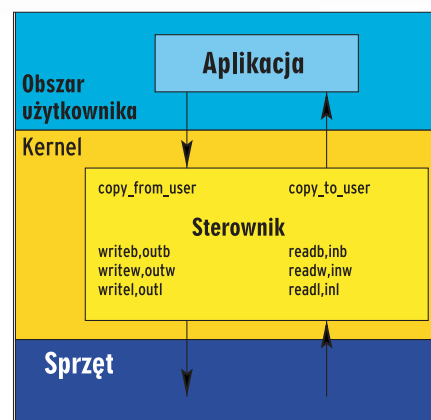
Jądro może uzyskać dostęp do obszaru pamięci urządzeń, które włączają się w normalną przestrzeń adresową, przy użyciu tych samych poleceń, za pomocą których obsługuje się zwykłą pamięć operacyjną. Aby nie zmniejszyć przenośności jądra, takie bezpośrednie operacje są zabronione. Zamiast nich występują funkcje *readb*, *readw* i *readl*. Odczytują one z przekazanego jako parametr adresu w pamięci jeden bajt lub jedno 16- lub 32-bitowe słowo. Analogicznie funkcje *writeb*, *writew* i *writel* zapisują wartość przekazaną jako pierwszy parametr w komórce pamięci, którą określa drugi parametr.

Bitowe typy danych

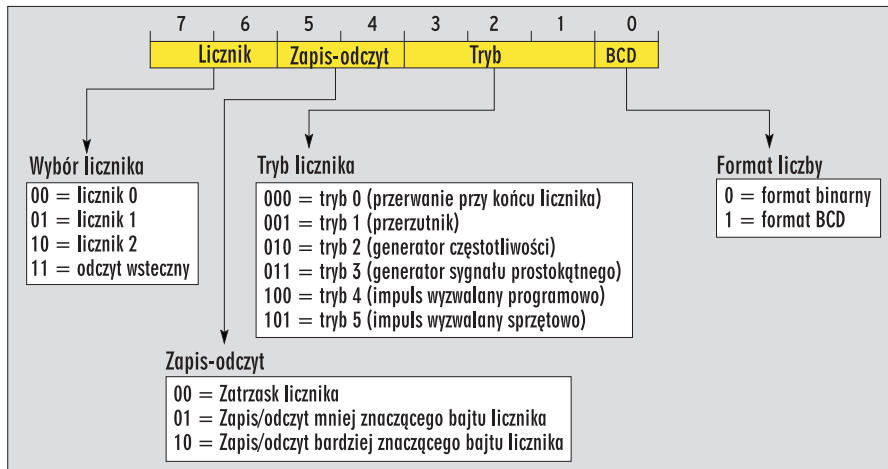
Nazewnictwo makr jest jasne: przy dostępie do sprzętu nie używa się typów *char*, *short*, *int* ani *long*, lecz danych o określonej w bitach długości. Jest tak dlatego, że typy danych w języku C zależą od typu komputera. Dla mikroprocesora typ danych *int* oznacza 16-

bitowe słowo, a dla najnowszego procesora może to być słowo 32-bitowe.

W Linuksie bitowe typy danych są definiowane przez dołączenie plików nagłówkowych *asm/types.h*. W ten sposób udostępniane są w nim definicje typów zmiennych *u8*, *u16* i *u32*, które oznaczają 8-, 16- i 32-bitowe zmienne bez znaku; z kolei typy *s8*, *s16* i *s32* oznaczają 8-, 16- i 32-bitowe zmienne ze znakiem. Bitowe typy danych są do dyspozycji w zmodyfikowanej postaci także w aplikacjach: typ *u8* z jądra w aplikacji ma formę *__u8*.



Rysunek 1: Jądro udostępnia sterownikowi cały szereg funkcji, służących zarówno do wymiany danych między aplikacjami a sterownikiem, jak i między sterownikiem a sprzętem.



Rysunek 2: Rejestr sterowania układem PIT składa się z wielu bitów [2]. W celu użycia głośniczka PC rejestr zostaje zainicjowany wartością 0xb6: stosowany jest licznik 2 oraz zapisywane są oba bajty licznika i wybrany tryb liczenia 3. Wówczas licznik pracuje w formacie binarnym.

Brzęczenie z głośnika

Na Listingu 1 widać, jak za pomocą przedstawionych makr można wydusić dźwięki ze starego, dobrego głośniczka w komputerze PC. Trzy adresy portów obsługują głośnik: 0x42, 0x43 i 0x61. Dwa pierwsze należą do układu PIT (Programmable Interval Timer – programowalny generator odstępów czasu). 0x42 to rejestr danych służący do zapisu wartości licznika układu PIT. Przez port 0x43 można ustawić sposób pracy tej kostki (zobacz Rysunek 2). Wartość 0xb6 włącza tryb pracy 3.

W tym trybie pracy układ PIT generuje sygnał prostokątny, który powstaje z wartości czasomierza ustawionej przez port 0x42 i CLOCK_TICK_RATE. Autentyczna wartość czasomierza jest 2-bajtowa, lecz musi być zapisana jako dwa pojedyncze bajty: najpierw bajt mniej znaczący, a potem bardziej znaczący. Na koniec makro przełącza wyjście układu PIT, z którego wychodzi sygnał prostokątny, na głośnik. W tym celu ustawia oba mniej znaczące bity z portu 0x61 na 1. Dokładniejsze informacje o programowaniu układu PIT znajdują się w [2].

Za pomocą znanego z [3] pliku Makefile można skompilować nowy sterownik głośniczka i załadować poleceniem *insmod*. Po utworzeniu pliku urządzenia:

```
mknod Speaker c 240 0
```

wreszcie usłyszymy przecudną muzykę. Gdy zapiszemy w nowym pliku urządzenia wartość:

```
echo "440" > Speaker
```

zostanie ona zinterpretowana jako częstotliwość i będzie słyszalna w głośniczku – może niezbyt to piękne, ale za to wyraźne. Polecenie „Zachować ciszę w bibliotece!” ma postać:

```
echo "0" > Speaker
```

Inteligentny Czytelnik na pewno dostrzeże, że przedstawiony sterownik co prawda działa, ale ma pewne braki. Pomijamy już to, że występuje tu niechroniona, krytyczna część, bo to historia na osobny artykuł. Jednak fakt, że sterownik korzysta z zasobów, czyli z portów, nie rezerwując ich wcześniej, jest w nowoczesnych systemach operacyjnych całkowicie niedopuszczalny. Należy to zaraz zmienić.

Listing 1: Dostęp do adresów portów

```
01 #include <linux/module.h>           26         outb((tone>>8) & 0xff,
02 #include <linux/version.h>         27         0x42);
03 #include <linux/init.h>             27         Save = inb( 0x61 );
04 #include <linux/fs.h>               28         outb( Save | 0x03, 0x61 );
05 #include <asm/uaccess.h>            29         } else {
06 #include <asm/io.h>                 30         outb(inb_p(0x61) &
07                                     31         0xFC, 0x61);
08 MODULE_LICENSE ("GPL");             31         }
09                                     32         return count;
10 static ssize_t DriverWrite          33     }
11 ( struct file *File,                34
12     __user const char *User,        35 static struct file_operations
13     size_t count, loff_t *Offs)     36     Fops = {
14     {                                 37         .owner=THIS_MODULE,
15         u16 tone;                    38         .write=DriverWrite,
16         u8 Save;                     39
17         char buffer[6];              40 static int __init BufInit(void)
18         if( count > sizeof          41     {
19         (buffer) )                   42     {
20         count = sizeof(buffer);       43         if(register_chrdev
21         copy_from_user( buffer,      44         (240, "PC-Speaker", &Fops) == 0)
22         User, count );               45         return 0;
23         buffer[sizeof(buffer)-1] =    46         return -EIO;
24         '\0';                         47
25         tone = (u16) simple_         47 static void __exit BufExit(void)
26         strtoul( buffer, NULL, 0 );   48     {
27         if( tone ) {                 49         outb(inb_p(0x61) & 0xFC,
28         tone = CLOCK_TICK_         50         0x61);
29         RATE/tone;                   51         unregister_chrdev(240,
30         outb( 0xb6, 0x43 ); //      52         "PC-Speaker");
31         Tryb 3: generowanie fali      53     }
32         prostokątnej                54
33         outb_p(tone & 0xff,         54 module_init( BufInit );
34         0x42);                       55 module_exit( BufExit );
```

Proszę jedną rezerwację!

Adresy portów są w takiej samej mierze zasobami, jak obszary pamięci operacyjnej, kanały DMA i przerwania. Aby jądro mogło poprawnie zarządzać zasobem, nie może z niego wielokrotnie korzystać. Dlatego też każdy składnik jądra musi rezerwować zasoby przed ich użyciem, a następnie – o ile tylko dostęp nie nastąpi w bezpośrednio dającej się przewidzieć przyszłości – szybko je zwalniać.

We właśnie opisanym sterowniku głośnika zignorowaliśmy z pewnych powodów rezerwację portów. Sterownik nie otrzymałby żądania zezwolenia, gdyż odnośne zasoby były już zajęte przez inne podsystemy, a więc zarezerwowane. Do rezerwacji w jądrze służy wiele funkcji:

> Porty

Prototypy znajdują się w pliku *linux/ioport.h*.

```
struct resource *request_region(
    unsigned long start,
    unsigned long count,
    const char *name );

void release_region(
    unsigned long start,
    unsigned long count );
```

Porty są rezerwowane na wyłączność dla jednego sterownika. Parametr *start* określa port, od którego jądro ma rezerwować *count* portów. Rezerwacja będzie przeprowadzona z użyciem nazwy *name*. Wartość zwrrotna funkcji *request_region* różna od NULL oznacza, że żądane zasoby są do dyspozycji sterownika.

> Adresy w pamięci

Prototypy są skatalogowane w pliku *linux/ioport.h*:

```
struct resource *request_mem_?
region(
    unsigned long start,
    unsigned long count,
    const char *name );

void release_mem_region(
    unsigned long start,
    unsigned long count );
```

Parametry odpowiadają funkcjom portów: *start* to adres początkowy rezerwowanego obszaru, *count* to liczba adresów, a *name* oznacza, przy użyciu którego jądro zarządza

rezerwacją i ją identyfikuje. Jeśli funkcja *request_mem_region* zwraca wartość różną od NULL, oznacza to, że sterownik może korzystać z żądanych zasobów.

> Kanały DMA

Prototypy znajdują się w pliku *asm/dma.h*:

```
int request_dma(
    unsigned int dmanr,
    const char *device_id);

void free_dma(
    unsigned int dmanr);
```

Parametr *dmanr* określa żądany kanał DMA, a *device_id* przysługujący mu identyfikator. Wartość zwrrotna różna od NULL wskazuje, że rezerwacja się udała.

> Przerwania

Prototypy znajdują się w pliku *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);

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

Przerwania to jedyne zasoby sprzętowe, których jądro nie przydziela z konieczności sterownikowi na wyłączność. Wspólne korzystanie z przerwania jest możliwe, gdy każda z odnośnych procedur obsługi przerwania może sprawdzić, czy to własne, czy też obce urządzenie wywołało przerwanie.

Parametr *irq* podaje numer przerwania. Nazwa sterownika jest zawarta w parametrze *devname*. Parametr *dev_id* musi być unikatowy w systemie. Służy on do rozróżniania wielokrotnych rezerwacji tego samego przerwania. Parametr *irqflags* to pole bitowe, które steruje - w pewnych granicach - zachowaniem procedury obsługi przerwania. Programista może przy tym łączyć następujące bity:

- **SA_SHIRQ**: wiele sterowników może korzystać z tego samego przerwania, gdyż nie jest ono dostępne na wyłączność dla jednego sterownika.

- **SA_INTERRUPT**: przerwania będą z powrotem zwalniane do lokalnego procesora

(tj. procesora, który właśnie obsługuje przetwarzanie). Wówczas procedura obsługi przetwarzania może być przerwana przez dalsze przetwarzanie.

- **SA_SAMPLE_RANDOM**: ustawiony bit oznacza, że moment wystąpienia przetwarzania służy do tworzenia liczby losowej.

Same adresy procedury obsługi przerwania są podawane w parametrze *handler*. W przeciwieństwie do wcześniejszych jąder, procedura obsługi przerwania ma wartość zwrrotną:

- Zwraca ona *IRQ_NONE*, o ile wyzwolone przerwanie nie było przeznaczone dla danego sterownika.

- Jeśli procedura obsługi przerwania obsłużyła przerwanie, zwraca *IRQ_HANDLED*.

- Wartość zwrrotna makra *IRQ_RETVAL(X)* zależy od tego, czy *X* ma wartość różną od NULL. Zwraca wówczas albo *IRQ_HANDLED*, albo *IRQ_NONE*.

Rezerwacja zasobów należy do fazy inicjacji urządzenia. Należą tutaj też:

- rozpoznanie urządzenia i
- inicjacja urządzenia.

Inicjacja sprzętu może tylko wtedy się udać, gdy sprzęt występuje w komputerze. Musi być znany jego adres i rozmiar zasobów. Głośnik w komputerze PC jest na przykład zawsze „zainstalowany” i jest zawsze dostępny pod tym samym adresem portu. Dlatego też te informacje są znane i można je uzyskać z każdej dokumentacji sprzętu komputera PC.

Nieuchwytny PCI

Wcześniejsza wiedza o standardowych składnikach komputera PC zawsze przydaje się przy każdego rodzaju sprzęcie. W wypadku dominujących już od dosyć dawna urządzeń PCI programista nie może liczyć, że występują one w komputerze podczas rozruchu systemu, gdyż są to, zgodnie z definicją linuxową, urządzenia podłączane podczas pracy (ang. hotpluggable devices). W ten sposób określa się urządzenia, które podczas pracy systemu (!) mogą być zainstalowane i wyjęte.

Jako że podsystem PCI jądra jest odpowiedzialny za stwierdzenie, czy urządzenie jest włożone, czy też wyjęte, procedura mogłaby też obsługiwać inicjację i dezinicjację urządzenia. Dlatego też w sterownikach PCI inicjacja sterownika (kod w funkcji *init_module*, względnie *modules_init*) jest oddzielona od inicjacji urządzenia. Podczas inicjacji sterownika zgłasza się on tylko w podsystemie PCI. Wywołuje on w tym celu funkcję

`pci_module_init`, której jedyny parametr jest typu `struct pci_driver`:

```
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);
    ...
};
```

Tu należy w istocie zapisać pola: nazwa sterownika `name`; tabela, przez którą sterownik stwierdza, za które urządzenie jest odpowiedzialny (`id_table`) i każdorazowo funkcja inicjacji oraz dezinicjacji urządzenia (`probe` i `remove`). Identyfikacja urządzenia PCI jest możliwa na podstawie łącznie sześciu cech: oznaczenia producenta, oznaczenia urządzenia, oznaczenia producenta i urządzenia dla podsystemu, klasy urządzenia i maski klasy urządzenia (wszystkie o długości 16 bitów).

Struktura danych typu `struct pci_device_id`, która przechowuje identyfikację,

wskazuje na kolejny element. Tak jak w wielu strukturach danych jądra, jest tu zwykle rezerwowane miejsce, które sterownik może wykorzystywać do własnych celów (zobacz Rysunek 3).

Na szczęście sterownik PCI nie musi podawać wszystkich cech w sposób wyczerpujący, aby później obsługiwać urządzenie. Zamiast wszystkich cech programista może zastosować oznaczenie wieloznaczne `PCI_ANY_ID` i `PCI_CLASS_NOT_DEFINED`. Najczęściej

wystarczy jawne ustawienie dwóch pierwszych cech.

Jeśli podsystem PCI znajdzie urządzenie, którego cechy zgadzają się z określonymi w strukturze sterownika, wywołuje funkcję detekcji – czyli funkcję sterownika służącą do inicjacji urządzenia. Z kolei sterownik na podstawie przekazanych przez podsystem PCI informacji rozstrzyga, czy może obsługiwać urządzenie, czy też nie. Jeśli tak, musi po pierwsze włączyć urządzenie za pomocą

Siedem składników sterownika PCI

- `struct pci_device_id`: struktura danych określa urządzenia, za które odpowiada sterownik.
- `struct pci_dev`: struktura danych przekazuje podsystemowi PCI opis sterownika. Należą tu tabele urządzeń (`struct pci_device_id`) i adresy funkcji służących do inicjacji i dezinicjacji urządzenia.
- `pci_modul_init`: za pomocą tej funkcji sterownik zgłasza się w podsystemie PCI.
- inicjacja urządzenia: sterownik musi udostępnić funkcję inicjacji urządzenia. Jej nazwę można wybrać dowolnie. Podsystem PCI uzyskuje adres tej funkcji za pośrednictwem struktury danych `struct pci_dev`. W ramach tej funkcji są także rezerwowane potrzebne zasoby.
- `pci_enable_device`: funkcja jest wywoływana w ramach inicjacji urządzenia. Włącza ona urządzenie.
- dezinicjacja urządzenia: sterownik musi mieć funkcję dezinicjacji urządzenia. Jej nazwę można wybrać dowolnie. Podsystem PCI uzyskuje adres tej funkcji za pośrednictwem struktury danych `struct pci_dev`. W ramach tej funkcji wcześniej zarezerwowane zasoby są zwalniane.
- `pci_unregister_driver`: gdy sterownik zostanie zwolniony z pamięci, musi się wyrejestrować z podsystemu PCI za pomocą tej funkcji.

Listing 2: Fragment kodu własnego sterownika

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/pci.h>
06 #include <linux/interrupt.h>
07
08 MODULE_LICENSE("GPL");
09
10 // TODO: Tu muszą zostać
11 wpisane własne identyfikatory.
12 #define MY_VENDOR_ID 0x10b7
13 #define MY_DEVICE_ID 0x5157
14
15 static unsigned long ioport=0L,
16 iolen=0L, memstart=0L, memlen=0L;
17
18 static irqreturn_t pci_isr( int
19 irq, void *dev_id, struct pt_regs
20 *regs )
21 {
22     return IRQ_HANDLED;
23 }
24
25 static int DeviceInit(struct
26 pci_dev *dev, const struct pci_
27 device_id *id)
28 {
29     printk("0x%4.4x|0x%4.4x:
30 %s\n", dev->
31 vendor, dev->device,
32 dev->dev.name );
33 if( request_irq (dev->
34 irq, pci_isr, SA_
35 INTERRUPT|SA_SHIRQ, "pci_drv", NULL)
36 ) {
37     printk(
38 KERN_ERR "pci_drv: IRQ %d not
39 free.\n", dev->irq );
40     return -EIO;
41 }
42
43 ioport = pci_
44 resource_start( dev, 0 );
45 iolen = pci_
46 resource_len( dev, 0 );
47 if( request_region(
48 ioport, iolen, dev->dev.
49 name )!=NULL ) {
50     release_region(
51 ioport, iolen );
52     printk(KERN_ERR
53 "Memory address conflict for device
54 %s\n",
55 dev->dev.name);
56     goto
57 cleanup_ports;
58 }
59 }
```

```

static struct pci_device_id pci_drv_tbl[] __devinitdata = {
  { 0x10b7, ← Oznaczenie producenta
    0x5950, ← Oznaczenie urządzenia
    PCI_ANY_ID, ← Oznaczenie producenta dla podsystemu
    PCI_ANY_ID, ← Oznaczenie urządzenia dla podsystemu
    PCI_CLASS_NOT_DEFINED, ← Klasa urządzenia
    0, ← Maska klasy urządzenia
    NULL }, ← Dane niestandardowe sterownika
  {0,} ← Koniec tabeli
};

```

Rysunek 3: Sterownik określa w strukturze danych typu `struct pci_device_id` wszystkie urządzenia, które jest w stanie obsłużyć. Tu zostaje wybrane urządzenie oznaczone jako `0x5950` przez producenta `3com 0x10b7` (`PCI_VENDOR_ID_3COM`).

funkcji `pci_enable_dev`, a po drugie zwrócić wartość `0` podsystemowi PCI. Każda inna wartość oznacza dla podsystemu PCI „proszę udać się do innego sterownika”.

Podsystem PCI wywołuje funkcję usuwania, kiedy urządzenie zostało wyjęte oraz kiedy sterownik zgłasza się w podsystemie PCI przy użyciu funkcji `pci_unregister_driver`.

Zbieranie informacji o PCI

Pozostaje jeszcze pytanie, w jaki sposób sterownik uzyskuje dostęp do zasobów rezerwowanych dla urządzenia. Ilekcioć podsystem

PCI wywołuje funkcję detekcji, jądro przekazuje sterownikowi strukturę danych typu `struct dev_id`, która zawiera żądane informacje.

Każde urządzenie PCI może posiadać maksymalnie sześć różnych obszarów pamięci lub wejścia-wyjścia. Adresy, rodzaj i wielkość tych obszarów są zapisane w komórkach oznaczonych w pamięci konfiguracyjnej sprzętu PCI (zobacz Rysunek 4) nazwami od `Base Address 0` do `Base Address 5`. Ponieważ informacje o adresie, rodzaju i wielkości są zakodowane w wartości 32-bitowej, sterownik musi ją odpowiednio odkodować.

Autorowi sterownika służą do tego makra zdefiniowane w pliku nagłówkowym `linux/pci.h`. Każdy z nich oczekuje dwóch parametrów: po pierwsze wskaźnika do przekazanych danych PCI o urządzeniu (`struct pci_dev`), a po drugie numeru komórki w pamięci konfiguracyjnej (a więc numeru od 0 do 5). Makro `pci_resource_start` zwraca adres początkowy obszaru pamięci. Każdy obszar ma minimalny rozmiar 16 bajtów. Konkretną długość podaje makro `pci_resource_len`.

Ponadto autor sterownika może uzyskać

koniec obszaru pamięci za pomocą makra `pci_resource_end`. Do określenia typu zasobu, pamięci lub adresu wejścia-wyjścia, istnieje także makro: `pci_resource_flag`. Plik nagłówkowy `linux/ioport.h` definiuje odpowiednie flagi `IORESOURCE_IO`, `IORESOURCE_MEM`, `IORESOURCE_IRQ` i `IORESOURCE_DMA`.

Także kiedy za pomocą tych mechanizmów można automatycznie zarezerwować zasoby urządzenia PCI, programista musi znać sposób wykorzystania tych zasobów. Najczęściej w dokumentacji sprzętu można znaleźć na przykład informacje, pod jakim adresem w pamięci konfiguracyjnej znajduje się dany zasób.

Na Listingu 2 widać szkielet prostej inicjacji urządzenia PCI. W przykładzie założono, że urządzenie wymaga przerwania, a w pamięci konfiguracyjnej `Base Address 0` ma adres wejścia-wyjścia, zaś w pamięci konfiguracyjnej `Base Address 1` adresy w przestrzeni adresowej pamięci. Autorzy własnych sterowników PCI muszą w szczególności dopasować identyfikację zasobów `MY_VENDOR_ID` i `MY_DEVICE_ID` do własnego sprzętu.

Moduł można skompilować i załadować nawet bez obsługiwanego przez niego sprzętu. Osoby, które wcześniej przetestowały moduł głośniczka, nie powinny zapomnieć

Listing 2: Fragment kodu własnego sterownika

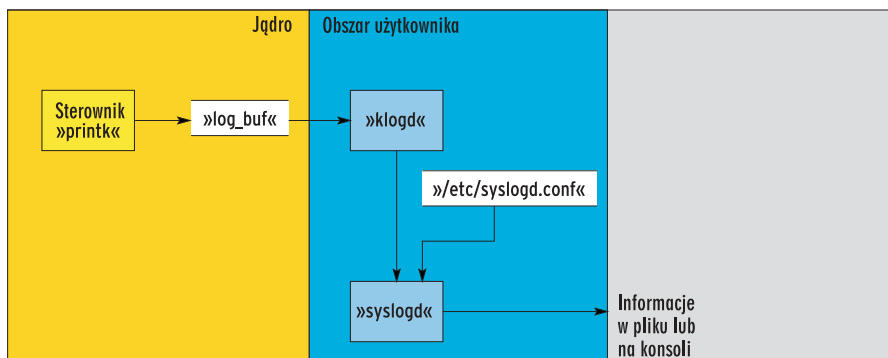
```

44     pci_enable_device( dev );
45     return 0;
46 cleanup_ports:
47     release_region( ioport, ↵
iolen );
48 cleanup_irq:
49     free_irq( dev->irq, ↵
"pci_drv" );
50     return -EIO;
51 }
52
53 static void DeviceDeInit ↵
( struct pci_dev *pdev )
54 {
55     free_irq( pdev->irq, ↵
"pci_drv" );
56     if( ioport )
57         release_region ↵
( ioport, iolen );
58     if( memstart )
59         release_mem ↵
region( memstart, memlen );
60 }
61
62 static struct file_operations ↵
PCIFops;
63
64 static struct pci_device_id ↵
pci_drv_tbl[] __devinitdata = {
65     { MY_VENDOR_ID, MY ↵
DEVICE_ID, PCI_ANY_ID, PCI_ANY_ID, ↵
0, 0, 0 },
66     { 0, }
67 };
68
69 static struct pci_driver ↵
pci_drv = {
70     .name= "pci_drv",
71     .id_table= pci_drv_tbl,
72     .probe= DeviceInit,
73     .remove= DeviceDeInit,
74 };
75
76 static int __init pci_drv ↵
init(void)
77 {
78     if(register_chrdev(240, ↵
"PCI-Driver", &PCIFops)==0) {
79
80         if( pci ↵
module_init(&pci_drv) == 0 )
81             return
82             unregister ↵
chrdev(240, "PCI-Driver");
83     }
84     return -EIO;
85 }
86 static void __exit ↵
pci_drv_exit(void)
87 {
88     pci_unregister_driver ↵
( &pci_drv );
89     unregister ↵
chrdev(240, "PCI-Driver");
90 }
91
92 module_init(pci_drv_init);
93 module_exit(pci_drv_exit);

```

	0x00	0x01		0x07	0x08					0x0f
0x00	Vendor ID	Device ID	Command Register	Status Register	Revision ID	Class code	Cache Line	Latency Timer	Header Type	BIST
0x10	Base Address 0		Base Address 1		Base Address 2		Base Address 3			
0x20	Base Address 4		Base Address 5		CardBus CIS pointer		Subsystem Vendor ID		Subsystem Device ID	
0x30	Expansion ROM Base Addr		Capab. List	Reserved			IRQ Line	IRQ Pin	Min GNT	Max LAT

Rysunek 4: Każde urządzenie PCI ma pamięć konfiguracyjną o wielkości 256 bajtów, w której przechowywane są najważniejsze parametry sprzętu – częściowo zakodowane. Podsystem PCI jądra przygotowuje istotne parametry sterownika przy użyciu struktury danych typu *struct pci_dev*.



Rysunek 5: Za pomocą funkcji *printk* kod jądra może przekazać informacje użytkownikowi. Demon dziennika jądra przekazuje dane normalnemu mechanizmowi *syslogd* w Linuksie.

o zwolnieniu starego modułu poleceniem *rmmmod*, gdyż oba moduły mają zakodowaną tę samą liczbę główną. Podczas ładowania modułu należy podać kompletną nazwę obiektu, lecz podczas zwalniania wystarczy podać tylko właściwą część nazwy: *insmod speaker.ko*, ale *rmmmod speaker*.

Na Listing 2 w oczy rzucają się dwie charakterystyczne rzeczy: użycie konstrukcji *goto* i dyrektywa *__devinit*. Instrukcji *goto* można byłoby łatwo uniknąć. Ponieważ jest ona jednak zgodna ze stylem programowania jądra, została tutaj umyślnie zastosowana (zo-

bacz Ramkę „Goto OR Not goto”). Dyrektywa *__devinitdata* należy do opisywanych w pierwszym odcinku [3] typu dyrektyw *__init* i *__exit*. Jest ona używana w deklaracji i definicji globalnych zmiennych, których z kolei wymaga inicjacja urządzenia podłączanego w czasie pracy komputera. Jeśli jądro nie obsługuje tej klasy urządzeń, umiejscawia ono tak wyróżnione zmienne w segmencie inicjacyjnym. Ten fragment pamięci po rozruchu zostaje zwolniony.

Na koniec jeszcze coś do wypróbowania: gdyby wiersze od 25 do 49 na Listing 2 zostały

skasowane, a obie definicje *MY_DEVICE_ID* i *MY_VENDOR_ID* zastąpione przez *PCI_ANY_ID* (*#define MY_DEVICE_ID PCI_ANY*), jądro wywoła funkcję detekcji *DeviceInit* dla każdego urządzenia PCI, dla którego wcześniej nie znalazło sterownika. Funkcja *printk* w wierszu 23 generuje odpowiednie dane w dzienniku *syslog* (zobacz Rysunek 5). To, czy wyświetlą się one na konsoli, czy też w pliku (na przykład */var/log/messages*), zależy od konfiguracji demona *syslog* (często */etc/syslogd*).

W następnym odcinku

Jeśli zawsze chcieliście się dowiedzieć, jak jest programowany wątek jądra i co to jest tasklet, a wstydziście się zapytać, to nie prze-gapcie następnego odcinka „Techniki jądrowej” w Linux Magazine. Ponieważ Linux Torvalds zrobił duże porządki w obszarze *Soft-IRQ* i kolejek roboczych w jądrze 2.6, temat będzie też fascynujący dla starych jądrowych wyjadaczy. (jk) ■

INFO

- [1] Eva-Katharina Kunst i Jürgen Quade „Technika jądrowa”, odcinek 2, Linux Magazine nr 13 z lutego 2005
- [2] „Assembler Page” (TU-Chemnitz): http://www.tu-chemnitz.de/informatik/RA/educ/mop/assembler_hints/ports.html
- [3] Eva-Katharina Kunst i Jürgen Quade „Technika jądrowa”, odcinek 1, Linux Magazine nr 12 ze stycznia 2005
- [4] Lista dyskusyjna jądra Linuksa – „zastosowanie instrukcji GOTO w jądrze”: <http://www.lkml.org/archive/2003/1/12/128/index.html>

AUTOR

Eva-Katharina Kunst, dziennikarka i Jürgen Quade, profesor w Hochschule Niederrhein, należą od czasu powstania Linuksa do fanów oprogramowania Open Source.

Goto OR Not goto

Uczeni są co do tego jednomyślni: *goto* nie pasuje do strukturalnego programowania, a więc należy unikać tej instrukcji. A jednak Linus Torvalds przeciwstawia się tutaj opinii profesorów (i to nie pierwszy raz). W bardzo ciekawym wątku na liście dyskusyjnej poświęconej jądru [4] omówiono wszystkie „za” i „przeciw”. Można więc podsumować, że instrukcja *goto* wewnątrz jądra systemu operacyjnego jest mile widziana, gdyż ma dwie podstawowe zalety: kod staje się po pierwsze bardziej przejrzysty, zatem czy-

telny, a poza tym instrukcje *goto* przyczyniają się do zwiększenia wydajności. Błędem byłoby używać instrukcji *goto* zawsze i wszędzie. Tego nie popierają także Torvalds i jego koledzy.

Jądro Linuksa korzysta z instrukcji *goto* przede wszystkim przy kończeniu nieudanych inicjacji. Tutaj to słowo kluczowe ułatwia zwolnienie zasobów w kolejności odwrotnej do tej, w jakiej je wcześniej zarezerwowano (zobacz Listing 2, funkcja *DeviceInit*).