System BIOS dla programujących językach w C i C++
(*Billy Taylor*, Wydawnictwo Nakom, Poznań 1996)

Wszystkie komputery osobiste IBM PC i inne komputery zgodne z nimi mają systemową pamięć ROM zawierającą *podstawowy system wejścia/wyjścia (BIOS - Basic Input/Output System)*. Zawiera on wiele użytecznych tablic i funkcji, pozwalających na obsługę napędów dysków komputera, drukarek, portów szeregowych, monitorów i innych urządzeń. Na początku pracy inicjalizuje komputer i uruchamia system operacyjny.

Kompilator języka C zawiera bibliotekę standardowych funkcji, pozwalających na wyświetlanie tekstu, czytanie i zapisywanie plików na dyskach oraz na wykonywanie innych zwyczajnych zadań. Funkcje te odwołują się do funkcji systemu operacyjnego lub BIOSU. Dlatego niezależnie od tego, jaką konfigurację i sprzęt komputerowy wybierzemy, wszystkie programy będą funkcjonowały normalnie.

Ponadto, producenci komputerów osobistych proponują nieustannie nowe rodzaje sprzętu. Wraz z postępem technicznym pojawiają się także nowe, całkowicie różne sposoby dostępu do poszczególnych elementów systemu. Jak to jest w ogóle możliwe, że przy takiej różnorodności elementów składowych wszystkie komputery osobiste mogą obsługiwać te same programy? W większości przypadków odpowiedzi należy szukać w systemowej pamięci ROM. Zawiera ona standardowe procedury, które:

- realizują operacje wejścia/wyjścia w komputerze na rozkaz systemu operacyjnego lub programów użytkowych;
- odpowiadają na sygnały komputera, gdy jest konieczna reakcja na nie;
- umożliwiają dostęp do wszystkich zaimplementowanych w pamięci ROM procedur i funkcji, do ich parametrów i zwracanych przez nic wartości, umożliwiając w ten sposób programom użytkowym i systemowi operacyjnemu dostęp do zasobów sprzętowych komputera, niezależnie od tego, jaki to jest komputer.

Ten zbiór procedur i funkcji standardowych to właśnie BIOS. Oryginalna architektura komputera osobistego IBM definiuje ten zbiór jako interfejs niskiego poziomu pomiędzy komputerem a programami użytkowymi lub systemami operacyjnymi. Zapewnia on jednolity i wygodny sposób wykonywania operacji wejścia/wyjścia komputera i zapewnia pełne sterowanie jego każdym elementem składowym. I tak, na przykład, system opera-

cyjny może poprosić BIOS, aby odczytał określony sektor na dysku bez względu na typ stacji dysków SCSI lub IDE. BIOS tłumaczy to żądanie na fizyczne rozkazy wejścia/wyjścia potrzebne do obsługi danego dysku.

Wszyscy producenci komputerów osobistych starają się zachować ten podstawowy element decydujący o kompatybilności. Każda firma, która projektuje systemowy ROM, musi naśladować oryginalny BIOS IBM i musi zapewnić wykonywanie udokumentowanych operacji w swoich warunkach technicznych. Postępując w ten sposób tworzy ona podstawowe ramy pozwalające programom na pracę na wszystkich komputerach osobistych kompatybilnych z IBM-em.

BIOS zawiera kod na poziomie sprzętu, potrzebny do obsługi głównych elementów systemu, takich jak monitor, klawiatura, napędy dyskietek, napędy dysków twardych, mysz, równoległe porty drukarki, porty szeregowe, zegar czasu rzeczywistego i głośnik. Możemy wyobrazić go sobie jako jedną warstwę systemu pokazanego na rysunku.

Aplikacja	Aplikacja nakazuje systemowi operacyjnemu wykonanie zadania, takiego jak odczytanie pliku lub wyświetlenie informacji. Przykład: Odczytaj plik z dysku lub wyświetl komunikat.
System operacyjny	Sterownik systemu operacyjnego przejmuje to polecenie, ustawia odpowiednie wartości rejestrów procesora i wywołuje funkcję BIOS, pozwalającą na uzyskanie dostępu do zasobów sprzętowych komputera.
BIOS	BIOS komunikuje się ze sprzętem wykonującym zadanie. Przykład: Wydanie polecenia odczytania danego sektora sterownikowi dysków.
Sprzęt	Sprzęt wykonuje rozkazy wydane przez BIOS.

Dla uzyskania dostępu do komputera programiści C najczęściej korzystają ze standardowych funkcji znajdujących się w bibliotece. Często łatwiej jest korzystać z tych funkcji bibliotecznych niż z funkcji BIOS, lecz z drugiej strony stwierdzić trzeba, że funkcje te nie stwarzają tak dużych możliwości, jak BIOS. W rzeczywistości wiele spośród tych funkcji ma ograniczony zakres zastosowania, nie ma uniwersalnego charakteru i nie odznacza się dużą skutecznością.

Jako przykład ilustrujący te cechy można podać funkcję **`printf()`**. Traktuje ona ekran tak, jak gdyby była to maszyna do pisania; powoduje pisanie znaków, poczynając od aktualnego położenia kursora, zupełnie niezależnie od tego, z jakimi kolorami tła i pióra mamy akurat do czynienia. Większe możliwości stwarza bezpośrednie wywoływanie funkcji obsługi karty graficznej BIOS z programu C. Pozwala ona bezpośrednio kontrolować obraz wyświetlany na ekranie monitora, tworzyć proste ciągi znaków lub skomplikowane obrazy graficzne, pisać w dowolnym miejscu ekranu, korzystając z wybranych kolorów, przesuwać kursor do dowolnego miejsca na ekranie i wykonywać inne ważne zadania stwarzające wielkie możliwości aplikacyjne.

Korzystanie z zasobów komputera

Adresy pamięci

Programiści w języku C odwołują się do zmiennych i funkcji, podając ich nazwę lub pośrednio, poprzez wskaźniki. Przy kompilacji i łączeniu programu przetwarzane są one na adresy pamięci, to jest na liczby, które opisują miejsce każdego elementu pamięci. Komputer odszukuje następnie poszczególne elementy korzystając z adresu, a nie posługując się nazwą.

Wiele funkcji BIOS wymaga podania na wejściu adresów pamięci lub przekazuje adresy jako wynik. Na przykład, chcąc odczytać poszczególne sektory dysku przy użyciu przerwania 13h, należy podać adres bufora, aby otrzymać dane. Trzeba nauczyć się, jak przetworzyć nazwy i wskaźniki C na adresy pamięci, które mogą być wykorzystane przez BIOS.

W komputerach opartych na procesorze 80x86 adresy pamięci składają się z dwóch części: szesnastobitowego numeru *segmentu* oraz szesnastobitowego *przesunięcia* (*offset*). Numer segmentu określa punkt wejścia w pamięci, a przesunięcie jest to odległość od tego punktu wejścia. Adres zapisuje się w postaci *segment:przesunięcie (segment:offset)*. Na przykład adres **`F000:0120h`** (**`h`** oznacza liczbę heksadecymalną czyli szesnastkową) określa element przesunięty o **`120h`** bajtów względem początku segmentu **`F000h`**. Zarówno numery segmentów, jak i przesunięcia rozpoczynają się od zera, przy czym adres **`0000:0000`** oznacza pierwszy bajt pamięci.

Najprostszy i najczęściej stosowany sposób tworzenia adresów pamięci w C

polega na użyciu *operatoa referencji* (&) i wskaźnika.

```
char buffer[] = „Błąd drukarki!”;
char *bufferStart;
char far *bufferEnd;
      bufferStart = &buffer[0];
      bufferEnd = &buffer[11];
```

Zmienne **bufferStart** (początek bufora) i **bufferEnd** (koniec bufora) to *wskaźniki* zawierające adresy pamięci - w tym przypadku adresy elementów w tablicy **buffer[]**. Przy posługiwaniu się BIOS należy zwrócić uwagę na dwa typy wskaźników: bliskie i dalekie *(near, far)*. Bliskie wskaźniki odnoszą się do adresów znajdujących się w obrębie aktualnego segmentu, a zatem takich, do których określenia wystarczy podanie przesunięcia. W stosowanych obecnie kompilatorach C tworzy się je w postaci liczb szesnastobitowych (o takim samym rozmiarze, jak **unsigned int**). Wskaźniki dalekie odnoszą się do adresów, które mogą znajdować się w dowolnym miejscu pamięci, a zatem takich, w przypadku których konieczne jest podanie zarówno numeru segmentu, jak i przesunięcia. W dzisiejszych kompilatorach C tworzy się je w postaci liczb trzydziestodwubitowych (o takiej samej wielkości, jak **unsigned long**). Numer segmentu zapisywany jest w starszych szesnastu bitach, a przesunięcie w młodszych szesnastu bitach. Tak więc, w programach C mamy do czynienia z dalekimi wskaźnikami mającymi postać liczb trzydziestodwubitowych. I tak, na przykład, adres **0xf000ffd9** oznacza przesunięcie **FFD9h** w segmencie **F000h**.

Uzyskiwanie numerów segmentów i przesunięć ze wskaźników

Chcąc przekazać daleki wskaźnik do BIOS. musimy rozłożyć go na numer segmentu i przesunięcie. Można tego dokonać za pomocą dwóch makrodefinicji znajdujących się w kompilatorach Borland i Microsoft C. **FP_SEG** i **FP_OFF** wybierają ze wskaźnika segment i przesunięcie podając je jako liczby całkowite bez znaku.

```
char string[] = "Błąd drukarki!";
char far *pCh = &string[0];
unsigned int segment, offset;
```

```
segment = FP_SEG( pCh );
offset  = FP_OFF( pCh );
```

Wersja **FP_SEG** firmy Microsoft działa (z definicji) tylko z dalekimi wskaźnikami. Wykorzystywanie jej z bliskimi wskaźnikami prowadzi do nieprawidłowych wyników. Makrodefinicje firmy Borland współdziałają prawidłowo ze wskaźnikami obu typów. Nowa implementacja firmy Borland może wytworzyć także numer segmentu i przesunięcie ze stałego adresu (w przeciwieństwie do zmiennej wskaźnikowej).

```
char string[] = „Błąd drukarki!”;
unsigned int segment, offset;

segment = FP_SEG(string);
offset = FP_OFF(string);
```

Rejestry segmentowe

Wszystkie procesory 80x86 przechowują numery segmentów w specjalnych, szesnastobitowych rejestrach segmentowych. Istnieją cztery rejestry:
CS — segment kodu,
DS — segment danych,
ES — dodatkowy segment (danych),
SS — segment stosu.

Zarówno kompilatory Microsoft, jak i Borland umożliwiają dostęp do rejestrów segmentowych poprzez strukturę, która zawiera wartości rejestrów przed i po przywołaniu funkcji. Aby utworzyć taką strukturę, należy zdefiniować zmienną typu **SREGS**, np.

```
struct SREGS segregs;
```

a następnie odwoływać się do poszczególnych rejestrów następująco:

```
segRegs.cs
segRegs.ds
segRegs.es
segkegs.ss
```

Rejestry ogólnego przeznaczenia (uniwersalne)

Komputer osobisty ma sześć rejestrów szesnastobitowych ogólnego przeznaczenia, używanych do wszystkich operacji, w tym także do przekazywania parametrów do funkcji BIOS i przekazywania wyników programowi wywołującemu. Dostęp do czterech rejestrów (AX, BX, CX i DX) można uzyskać albo w całości, albo jako dostęp do dwóch rejestrów ośmiobitowych. Dostęp do pozostałych dwóch rejestrów (SI i DI) można uzyskać tylko w taki sposób, jak do rejestrów szesnastobitowych.

Rejestry ogólnego przeznaczenia procesorów 80x86

16 bitów	starsze 8 bitów	młodsze 8 bitów
AX	AH	AL.
BX	BH	BL
CX	CH	CL
DX	DH	DL
SI		
DI		

Kompilatory Microsoft i Borland dają dostęp do rejestrów ogólnego przeznaczenia poprzez strukturę **`REGS`**. Pola struktury **`REGS`** przechowują zawartość rejestrów przed i po przywołaniu funkcji. Zmienna taka może być zdefiniowana następująco:

```
union REGS inregs;
```

Unia **`REGS`** umożliwia dostęp do rejestrów ogólnego przeznaczenia Należy pamiętać o tym, że zarówno nazwy ośmiobitowe, jak i szesnastobitowe odnoszą się do tych samych rejestrów. Stąd też zmiana jednej z nich powoduje również zmianę innych. Na przykład, **`inRegs.x.ax`** i połączenie **`inRegs.h.ah`** i **`inRegs.h.al`** oznacza fizycznie ten sam rejestr.

Elementy zmiennej **`REGS`**

Rejestry szesnastobitowe	Rejestry ośmiobitowe
`inRegs.x.ax`	**`inRegs.h.ah`** **`inRegs.h.al`**

`inRegs.x.bx`	`inRegs.h.bh` `inRegs.h.bl`
`inRegs.x.cx`	`inRegs.h.ch` `inRegs.h.cl`
`inRegs.x.dx`	`inRegs.h.dh` `inRegs.h.dl`
`inRegs.x.si`	
`inRegs.x.di`	
`inRegs.x.cflag`	(rejestr znaczników — *flag register*)

Znaczniki

Procesory 80x86 posiadają jednobitowe znaczniki odzwierciedlające wynik wykonania instrukcji maszynowej. Znacznik może być albo włączony (PRAWDA, wartość 1) lub wyłączony (FAŁSZ, wartość 0). Znacznikami najczęściej używanymi przez programistów są:

- znacznik *Carry* (Przeniesienie) pokazuje, czy działanie arytmetyczne spowodowało wystąpienie przeniesienia lub pożyczki. Oznacza on także bit usuwany z rejestru podczas operacji przesunięcia. BIOS wykorzystuje go do pokazania, czy funkcja została prawidłowo wykonana;
- znacznik *Zero* pokazuje, czy w wyniku otrzymano zero;
- znacznik *Sign* (znaku) zawiera najbardziej znaczący bit wyniku.

Struktura **`REGS`** umożliwia dostęp tylko do znacznika *Carry*. Zero oznacza fałsz (FALSE), a wartość różna od zera oznacza prawdę (TRUE).

Wywoływanie funkcji BIOS

Programy wywołują funkcje BIOS za pomocą przerwań programowych (*software interrupts*). Przerwanie sprzętowe powoduje zawieszenie czynności wykonywanych przez procesor, reakcję na sygnał, a następnie powrót do przerwanego działania. O przerwaniu programowym można powiedzieć, że następuje w wyniku celowego działania programu, a nie jest powodowane przez źródło zewnętrzne (sprzęt).

Najważniejsze przerwania programowe BIOS

Przerwanie	Obsługiwane urządzenie lub funkcja
05h	obsługa klawisza *print screen*

10h	funkcje obsługi karty graficznej
11h	informacja o zainstalowanym wyposażeniu
12h	podaje rozmiar pamięci podstawowej
13h	funkcje dyskietki i dysku twardego
14h	funkcje portu szeregowego
15h	obsługa systemu
16h	funkcje klawiatury
17h	funkcje obsługi portu równoległego
1Ah	obsługa *timera* i zegara systemowego
1Ch	obsługa licznika systemowego

Każde przerwanie ma do dypozycji funkcje, które spełniają określone zadania. I tak - na przykład - chcąc wyświetlić znak, wywołujemy funkcję 0Ah przerwania 10h. Chcąc odczytać określony sektor na dysku, przywołujemy funkcję 2 przerwania 13h.

Chcąc wywołać funkcję BIOS należy postępować w następujący sposób:

Krok 1: Załadować odpowiednimi wartościami rejestry procesora

Najpierw trzeba załadować rejestry ogólnego przeznaczenia (uniwersalne) i rejestry segmentowe (w razie potrzeby) wstawiając do nich wartości wymagane przez funkcję. Informacje o zawartości rejestrów są dostępne w różnych dostępnych opisach funkcji.
Na przykład — w jaki sposób możemy stwierdzić, czy zainstalowany został określony napęd dysku twardego? W tym celu trzeba sprawdzić przerwanie 13h (funkcje dyskowe) i zauważyć, że funkcja 15h zwraca interesującą nas informację. Tak jak to podano w dodatku A, mamy do czynienia z następującymi wartościami wejściowymi:

AH = 15h (numer funkcji)
DL = 80h (pierwszy twardy dysk)
Następnie możemy załadować wartości do zmiennej **REGS**:

```
union REGS inRegs, outRegs;

inRegs.h.ah = 0x15;    /* AH = Funkcja 15h  */
inRegs.h.dl = 0x80;    /* DL = Numer napędu */
```

Jeśli dana funkcja wymagała wykorzystania rejestrów segmentowych, możemy w podobny sposób załadować zmienną **SREGS**.

Krok 2: Wywołanie BIOS

W bibliotekach firm Microsoft i Borland znajdują się dwie funkcje wywoływania przerwań BIOS: **int86()** oraz **int86x()**.

Funkcja **int86()** przekazuje tylko rejestry ogólnego przeznaczenia AX, BX, CX, DX, SI i DI. Nie przekazuje rejestrów segmentowych. Składnia wywoływania jest następująca:

```
int86( interruptNumber, &inRegs, &outRegs );
```

Parametr **interruptNumber** (numer przerwania) oznacza przerwanie BIOS, **&inRegs** jest to adres zmiennej **REGS** zawierającej dane wejściowe, a **&outRegs** — adres zmiennej **REGS** potrzebny do otrzymania wartości wyjściowych. Po zakończeniu wykonywania funkcji **int86()** zwraca wszystkie rejestry ogólnego przeznaczenia i znacznik przeniesienia (prawda lub fałsz).

Początek funkcji wykrywającej zainstalowanie dysku twardego wygląda następująco:

```
unsigned char hardDriveIsInstalled( void )
{
 union REGS inRegs, outRegs;
 inRegs.h.ah = 0x15;  /* AH = Funkcja 15h  */
 inRegs.h.dl = 0x80;  /* DL = Numer napędu */
 int86( 0x13, &inRegs, &outRegs );
                /* Wywołanie funkcji BIOS */
}
```

Jeśli funkcja BIOS oczekuje lub zwraca rejestry segmentu, należy użyć **int86x()** zamiast **int86()**. Działanie obu tych funkcji jest takie samo - z tą różnicą, że **int86x()** przekazuje także rejestry DS i ES. Składnia wywołania jest następująca:

```
int86x(interruptNumber,&inRegs,&outRegs,&segRegs );
```

gdzie **interruptNumber**, **&inRegs** i **&outRegs** są takie same, jak dla **int86()**, a **&segRegs** jest adresem zmiennej typu **SREGS**. Funkcja zwraca rejestry ogólnego przeznaczenia, znacznik przeniesienia (*Carry*) i rejestry DS i ES.

Krok 3: Zbadanie, czy rejestry wyjściowe zwróciły informacje

Funkcje BIOS informują, czy operacja została wykonana, czy też nie albo za pomocą znacznika przeniesienia (fałsz, gdy operacja została wykonana; prawda, gdy zakończyła się niepowodzeniem), lub przez zwrot kodu stanu w rejestrze ogólnego przeznaczenia (zazwyczaj AH).

Analizując funkcję 15h przerwania 13h stwierdzamy, że zwraca ona rejestr AH=3, jeśli napęd jest zainstalowany. Zatem funkcja wykrywająca zainstalowanie dysku twardego ostatecznie wygląda następująco:

```
unsigned char hardDriveIsInstalled( void )
{
 union REGS inRegs, outRegs;
 inRegs.h.ah = 0x15;  /* AH = Funkcja 15h  */
 inRegs.h.dl = 0x80;  /* DL = Numer napędu */
 int86( 0x13, &inRegs, &outRegs );
                  /* Wywołanie funkcji BIOS */
 if (outRegs.h.ah == 3)
  return (1); /* AH=3 jeśli HDD zainstalowany */
 else
  return (0);
}
```

Tworzenie wskaźników z numerów segmentów i przesunięć

Niektóre przerwania zwracają adres pamięci, wykorzystując do tego rejestr DS lub ES (segmentu) i rejestr ogólnego przeznaczenia (przesunięcie). I tak - na przykład - funkcja 8 przerwania 13h zwraca adres tablicy parametrów napędu twardego dysku w ES:DI. Chcąc uzyskać dostęp do zawartych tam danych trzeba połączyć dwa rejestry tworząc daleki wskaźnik. Borland dostarcza makrodefinicję zwaną **MK_FP**, pozwalającą na wykonanie tej konwersji. Microsoft nie dostarcza takiej definicji.

```
/* Użycie funkcji 8 przerwania 13h do zademon-
strowania konwersji adresu w postaci se-
ment:przesunięcie do zmiennej za pomocą makrode-
finicji MK_FP  -  Borland */

unsigned char far *getDriveParameterTable
                    ( unsigned char whichDrive )

union REGS regs;
struct SREGS sregs;
unsigned char far *tablePtr;

/* Początkowo wskaźnik=NULL na wypadek niepowodzenia
funkcji 8 */
 tableptr = NULL;
 regs.h.ah = 8;              /* AH = funkcja 8 */
 regs.h.dl = whichDrive; /* DL = numer napędu */
 int86x( 0x13, &regs, &regs, &sregs );/* Wywołanie
funkcji BIOS */

/* Jeżeli operacja powiodła się, to sregs.es zawiera
numer segmentu, a regs.x.di zawiera przesunięcie.
MK_FP tworzy na ich podstawie daleki wskaźnik.
Pierwszym argumentem jest numer segmentu, drugim
przesunięcie. */

 if ( regs.x.cflag == 0 ) /* Jeżeli 0K, */
  tablePtr = MK_FP( sregs.es, regs.x.di );
/*
Zwracamy wskaźnik do funkcji wywołującej (lub NULL,
jeżeli funkcja przerwania 13h nie została poprawnie
wykonana).
*/
return( tablePtr );
}
```

Makrodefinicja **MK_FP** wymaga użycia jako pierwszego argumentu numeru segmentu, a jako drugiego - przesunięcia. Utworzenie zmiennej

wskaźnikowej bez makrodefinicji jest możliwe, lecz wymaga objaśnienia. Dalekie wskaźniki to liczby trzydziestodwubitowe. Numer segmentu zajmuje 16 bitów starszych (bardziej znaczących), a przesunięcie — 16 bitów młodszych (mniej znaczących). Gdy mamy numer segmentu i przesunięcie będące liczbami całkowitymi, możemy utworzyć zmienną wskaźnikową zachowując te człony na swoich miejscach:

```
/* Zapisz numer segmentu do młodszych 16 bitów */
(unsigned long)tablePtr = sregs.es;

/* Przesuń numer segmentu do starszych 16 bitów */
(unsigned long)tablePtr <<= 16;

/* Zapisz przesunięcie do młodszych 16 bitów */
(unsigned long)tablePtr |= regs.x.di;

/* tablePtr zawiera teraz daleki wskaźnik
zwrócony przez funkcję BIOS */
```

Metoda ta jest skuteczna w przypadku aktualnie używanych kompilatorów C, korzystających ze wskaźników trzydziestodwubitowych. W przypadku kompilatorów korzystających ze wskaźników sześćdziesięcioczterobitowych konieczne jest wprowadzenie pewnych zmian.

Rady i ostrzeżenia

- Funkcje BIOS pobierają wartości wejściowe z rejestrów ogólnego przeznaczenia i rejestrów segmentów, odpowiednio poprzez zmienne **REGS** i **SREGS**.
- Zwracają one dane i stan zakończenia operacji za pośrednictwem **REGS** i **SREGS**. Większość z nich zwraca znacznik przeniesienia ustawiony na fałsz i AH=0 w przypadku, gdy operacja powiedzie się i znacznik przeniesienia równy prawdzie i AH=kodowi błędu, jeśli operacja nie powiedzie się.
- Do podziału zmiennych wskaźnikowych na numer segmentu i przesunięcie należy wykorzystać makrodefinicje **FP_SEG** i **FP_OFF**.
- Do połączenia numerów segmentu i przesunięć tak, aby utworzyły one zmienne wskaźnikowe, należy wykorzystać makrodefinicję **MK_FP**

(dotyczy to użytkowników kompilatora Borland).

- Dostęp do rejestrów AX, BX, CX i DX można uzyskać również w postaci jednostek ośmiobitowych, nazywanych AH, AL, BH, BL, CH, CL, DH i DL. Nie wolno próbować używać obu tych form jednocześnie (na przykład, jeżeli napisze się AH, nie wolno jednocześnie napisać AX. AH i AL tworzą bowiem AX).
- Trzeba pamiętać, że niektóre modele pamięci kompilatorów tworzą bliskie wskaźniki automatycznie. Dalekie wskaźniki dla wartości wejściowych lub wyjściowych otrzymywanych z funkcji BIOS, trzeba specjalnie zadeklarować jako typ *far* (daleki).

Użyteczne funkcje języka C

```
void interrupt (far *getvect(int num))();
```

getvect() pobranie wektora przerwań: czyta i zwraca wartość wskaźnika do funkcji obsługi przerwania o numerze **num**

```
void setvect(int num, void interrupt (*isr)( ));
```

setvect() umieszcza w tablicy wektorów przerwań na pozycji **num** daleki wskaźnik do nowej funkcji obsługi przerwania **(*isr)( )**

```
int inport(unsigned portid);
```

inport() – czyta daną typu int z portu (urządzenia wejściowego) o numerze **portid**

```
unsigned char inportb(unsigned portid);
```

inportb() – czyta bajt z portu o numerze **portid**

```
void outport(int portid, int value);
```

outport() – wpisuje daną typu int reprezentowaną przez **value** do

portu (urządzenia wyjściowego) o numerze **portid**

```
void outportb(int portid, unsigned char value);
```

outportb() – wpisuje bajt reprezentowany przez **value** do portu o numerze **portid**

```
void disable(void);
```

disable() blokuje możliwość obsługi przerwań sprzętowych

```
void enable(void);
```

enable() odblokowuje możliwość obsługi przerwań sprzętowych