V. Kod wynikowy

Niniejszy rozdział różni się trochę od reszty książki. Rozważania w nim zawarte "leżą na niższym poziomie abstrakcji", nie dotyczą bowiem składni ani bibliotek języka C lecz procesu powstawania programu wynikowego i sposobów ingerencji w ten proces. Podczas gdy w pozostałych rozdziałach starałem się opisywać język C w sposób niezależny zarówno od kompilatora jak i sprzętu, ten rozdział będzie dotyczył tylko komputerów kompatybilnych z IBM PC, pracujących pod nadzorem systemu DOS.

Każdy, kto zetknął się z językiem C, zastanawiał się zapewne, dlaczego najprostszy program:

```
main()
{}
```

daje tak duży kod wynikowy. Otóż dzieje się tak nie dlatego, że kompilatory C dają bardzo nieoptymalny kod, tylko dlatego, że bardzo poważnie traktują każdy program. W wyniku takiego poważnego podejścia, do każdego programu dołączany jest spory fragment kodu (w dalszej części rozdziału będę określał go mianem Startup), który po pierwsze przygotowuje pewne dane dla programu głównego a po drugie robi wszystko co możliwe, żeby zabezpieczyć system przed zmianami jakie może poczynić program. Startup zachowuje szereg informacji o stanie systemu w chwili uruchomienia programu: adres zmiennych środowiskowych, wersję DOS-u, wektory niektórych przerwań, itp. Kolejną wykonywaną czynnością jest przygotowanie pewnych danych i funkcji wykorzystywanych przez program główny. Instalowana jest na przykład standardowa funkcja obsługi błędu dzielenia przez zero, funkcje arytmetyki zmiennoprzecinkowej, zapamiętywana jest wartość zegara BIOS-u w celu ewentualnego użycia przez funkcję **clock**.

Jedną z ważniejszych funkcji wykonywanych przez Startup jest przygotowanie argumentów dla funkcji **main**. Funkcja ta może mieć trzy argumenty:

```
main(int argc, char *argv[], char *envp[])
```

Oczywiście, nie każdy program musi używać argumentów, a nie każdy, który ich używa, musi używać wszystkich. Poprawne są również następujące nagłówki funkcji **main**:

```
main()
main(int argc)
main(int argc,char *argv[])
```

W pierwszym argumencie, oznaczonym tutaj argc, Startup przekazuje do programu głównego liczbę argumentów w wierszu wywołania programu plus jeden.

Drugi argument, oznaczony argv, jest wskaźnikiem do tablicy wskaźników wskazujących kolejne argumenty wywołania. Na przykład, jeżeli program został wywołany w następujący sposób:

```
program -u book a:\*.c b:\*.c
```

to argumenty funkcji **main** będą miały następujące wartości:

```
argc=5
 argv[0]="program"        /* DOS 3.0 i wyzsze */
 argv[1]="-u"
 argv[2]="book"
 argv[3]="a:\*.c"
 argv[4]="b:\*.c"
```

Ostatni argument funkcji **main**, oznaczony envp, jest wskaźnikiem do tablicy wskaźników do zmiennych środowiskowych. Ostatnim elementem tablicy jest wskazanie puste NULL.

Poniższy program używa trzeciego argumentu funkcji **main**, do wypisywania wartości zmiennych środowiskowych w chwili uruchomienia programu.

```
#include <stdio.h>

void main(int c, char *v[], char *env[])
{
 while(*env)
  printf("%s\n",*env++);
}
```

Po wykonaniu wszystkich opisanych czynności, Startup wywołuje program główny, czyli funkcję **main**. Po powrocie z funkcji **main**, czyli po zakończeniu działania programu, Startup odtwarza zapamiętane przed wywołaniem programu informacje (na przykład wektory przerwań) i wraca do DOS-u.

1. Zmieniamy Startup

Często pisze się proste programu, nie wymagające zachowywania wektorów przerwań, instalowania procedur obsługi błędów i wykonywania wszystkich tych operacji, które robi Startup. Chcielibyśmy, żeby takie programy były krótkie a mogą one być krótkie, a nawet bardzo krótkie. Żeby tak się stało, trzeba pozbyć się zbędnego kodu, a więc trzeba usunąć oryginalny Startup.

Spróbujmy stworzyć własną wersję Startup-u, która będzie ograniczała się jedynie do wywołania funkcji **main**, umożliwiała zrobienie z prostego programu małego COM-a.

W tym miejscu trzeba kilka słów poświęcić sposobowi w jaki kompilator (i konsolidator) dołączają Startup do programu.

W przypadku kompilatorów firmy Borland jest to realizowane w sposób bardzo prosty i przejrzysty. Kod Startup zawarty jest w osobnych plikach o nazwach **c0?.obj** (? symbolizuje tu literę oznaczającą model pamięci np. s - model *small*). Dla każdego modelu pamięci istnieje osobny kod Startup i osobny moduł **c0?.obj**.

Konsolidacja programu w środowiskach firmy Borland wygląda następująco:

```
tlink c0s.obj program ... , program , , cs.lib
```

(kropki oznaczają ewentualne kolejne moduły programu).

Dzięki takiemu rozwiązaniu zastąpienie Startup-a sprowadza się do zastąpienia modułu **c0s.obj** z powyższego przykładu innym, np.:

```
tlink tsr16.obj program ... , program , , cs.lib
```

W środowisku Microsoft C nie ma oddzielnych modułów Startup. Kod startowy "zaszyty" jest w bibliotekach i automatycznie dołączany do każdego programu łączonego z biblioteką. Aby standardowy kod Startup nie został dołączony do programu i mógł być zastąpiony innym, w programie należy zdefiniować zmienną o nazwie _acrtused (dlatego we wszystkich programach przedstawionych w tym rozdziale taka zmienna jest zdefiniowana):

```
int _acrtused=0;
```

i dokonać konsolidacji programu z opcją /NOE, np. tak:

```
link /NOE tsr16.obj program ... , program , , slibce.lib
```

Wszystkie opisane w tym rozdziale przykłady kodów startowych służą do otrzymywania programów typu COM.

Punkt wejścia programu typu COM musi mieć przemieszczenie 100h. Nasz pierwszy, najprostszy Startup od razu wywoła funkcję **_main**, a po powrocie z tej funkcji wróci do DOS-u. Przypominam, że identyfikatory globalne w języku C są automatycznie poprzedzane podkreśleniem. Dlatego funkcja **main** nazywa się w rzeczywistości _main.

```
; plik c0.asm

.MODEL SMALL
extrn _main:near
.CODE
 ORG 100h
 start:      call _main        ; wywolaj funkcje main
             mov ah,4Ch
             int 21h           ; wróc do DOS-a
             end start
```

Proszę zwrócić uwagę, że przed powrotem do DOS-u ustawiamy tylko zawartość rejestru AH, natomiast rejestr AL będzie zawierał wartość zwróconą przez funkcję **main**. Ta właśnie będzie zwrócona przez program.

Po asemblacji pliku c0.asm otrzymamy moduł c0.obj.

Możemy teraz napisać pierwszy program, w którym oryginalny Startup zastąpimy naszą minimalną wersją. Program będzie zamieniał ze sobą porty drukarki, a więc po jego wykonaniu port pierwszy stanie się drugim i vice versa. Progra-

mik taki jest przydatny np. gdy mamy uszkodzony port LPT1 i chcemy "podstawić" go portem LPT2.

```
/* plik lptport.c */

int _acrtused=0;

void main()   /* program wymienia adresy portów LPT1 i
LPT2 */
{
 int x;
 x=*(int far *)0x408;                /* 40h:08h adres
LPT1 */          *(int far *)0x408=*(int far *)0x40A; /*
40h:0Ah adres LPT2 */
 *(int far *)0x40A=x;
}
```

Program zamienia ze sobą adresy portów drukarki umieszczone w obszarze danych BIOS-u. Załóżmy, że program znajduje się w pliku **lptport.c**. Po skompilowaniu w modelu *Small*[1)] otrzymamy moduł **lptport.obj**.

Możemy teraz utworzyć program wykonywalny:

```
tlink /t c0.obj lptport.obj , lptport.com
```

lub dla Microsoft C

```
link /NOE c0.obj lptport.obj , lptport , , slibce
exe2bin lptport.exe
```

W bieżącym katalogu zostanie utworzony program **LPTPORT.COM** . Jego długość w zależności od kompilatora i ustawionych opcji wyniesie około 50 bajtów!

Wprawdzie nie robi on wiele, ale ten sam program skompilowany "standardowo" zajmuje przecież prawie 4 tysiące bajtów.

Następny programik będzie służył do wyboru opcji w programach wsadowych (typu .bat). Będzie on pobierał z klawiatury odpowiedź na zadane pytanie i zwracał odpowiednią wartość. Wartość zwracaną przez program można w programach wsadowych testować przy pomocy warunku:

```
if ERRORLEVEL liczba
```

Warunek ten jest spełniony, gdy wartość zwrócona przez ostatnio wywołany program jest równa lub większa od wartości liczba.

Ponieważ funkcje obsługi wejścia w standardowych bibliotekach C są dość obszerne, będziemy czytać klawiaturę bezpośrednio przy pomocy funkcji BIOS-u. Odpowiednią do naszego programu będzie funkcja 0 przerwania 16h. Funkcję

[1] Wszystkie programy należy kompilować bez informacji dla debuggera, a w kompilatorach firmy Microsoft z opcją /Gs

_getch czytającą znak z klawiatury zdefiniujemy w osobnym pliku **getch.asm**, gdyż będziemy jej używać także w innych programach[2)].

```
; plik getch.asm

.MODEL SMALL
.CODE

 proc __getch       ; funkcja pobiera znak z bufora
klawiatury
       mov ah,0     ; i zwraca jako rezultat
       int 16h      ; jeżeli bufor jest pusty czeka na
znak
       ret
 endp

 public __getch
 end
```

Wykorzystana funkcja 0 przerwania 16h czeka na naciśnięcie klawisza i zwraca w rejestrze AX kod naciśniętego klawisza. Wartości tej nie trzeba nigdzie przepisywać, ponieważ program w języku C spodziewa się przekazania wartości funkcji typu **int** właśnie w rejestrze AX.

Program ma pobierać odpowiedź na pytanie. Załóżmy, że po naciśnięciu klawisza T (odpowiedź twierdząca) program będzie zwracał wartość 116 (kod znaku 't') a w przeciwnym wypadku wartość 0.

```
/* plik tak_nie.c  */
int _acrtused=0;
main()
{
 char c=_getch();              /* pobierz znak z
klawiatury */
 if(c=='t'||c=='T')
       return 't';             /* jezeli 't' lub 'T'
*/
    else
       return 0;               /* w przeciwnym razie
*/
}
```

Teraz kompilujemy nasz program w modelu *Small* i tworzymy program wykonywalny.

```
tlink /t c0.obj getch.obj tak_nie.obj , tak_nie
```

lub

```
link /NOE c0.obj getch tak_nie , tak_nie , , slibce
exe2bin tak_nie.exe
```

Programik ma tylko 50 bajtów długości. Można go wykorzystać w programach wsadowych w następujący sposób:

---

[2] Moduły asemblerowe, w których definiowane są symbole używane w programie w C należy kompilować programem TASM z opcją /ml

```
@echo "Zainstalować cache-a (t/n) ? "
@tak_nie
@if ERRORLEVEL 116 SUPERPCK /EP /T+
```

Po uruchomieniu powyższego programu na ekranie zostanie wyświetlone pytanie:

```
Zainstalować cache-a (t/n) ?
```

Jeżeli użytkownik naciśnie klawisz **T**, zostanie uruchomiony program SUPERPCK, w przeciwnym wypadku uruchomienie programu nie nastąpi.

2. Programy rezydentne

W rozdziale tym zaprezentuję "zastępczy" Startup, który nie będzie, jak poprzednio, uproszczeniem Startup-a oryginalnego, lecz będzie pełnił całkowicie odmienną funkcję. Zamiast uruchamiać program główny (funkcję **main**), będzie on instalował go jako program rezydentny, przechwytujący wybrane przerwanie. W ten sposób zamienienie zwykłego programu na program rezydentny będzie bardzo proste i będzie sprowadzać się do jego powtórnej konsolidacji.

Zanim przejdę do meritum, chciałbym przypomnieć po krótce czym są programy rezydentne, do czego służą i jak się je tworzy.

Programy rezydentne to programy, które po zakończeniu działania pozostawiają część swojego kodu w pamięci i ustawiają wektor wybranego przerwania tak, żeby wskazywał na ten kod. Po wygenerowaniu tego przerwania następuje uruchomienie rezydującej w pamięci części programu.

2.1. Jakie są zastosowania programów rezydentnych ?

Program rezydentny może być rozszerzeniem systemu operacyjnego, dostarczającym innym programom usług, których nie zapewnia system. Przykładem takiego programu jest sterownik myszki. Programy, które chcą skorzystać z myszki (np. odczytać położenie kursora myszki, stan klawiszy) wywołują przy pomocy odpowiedniego przerwania rezydentny program sterownika myszki i przekazują mu w rejestrach procesora informację o jaką usługę proszą.

Innym zastosowaniem programów rezydentnych jest zmiana obsługi standardowego przerwania. Program taki może na przykład przejąć przerwanie klawiatury w celu niestandardowej interpretacji niektórych klawiszy (np. ALT+A jako ą).

Przy pomocy programów rezydentnych realizuje się także w systemie DOS namiastkę procesów działających w tle. Program przejmuje w tym celu jakieś często wywoływane przez system przerwanie (np. przerwanie zegara 1Ch lub przerwanie obsługi klawiatury 16h), dzięki czemu jest odpowiednio często wywoływany. Takie "procesy" mogą albo wykonywać w tle jaką prostą pracę albo tylko czuwać i ujawniać się dopiero po zaistnieniu określonego warunku (upływie określonego czasu, naciśnięciu jakiegoś klawisza itp.).

2.2. Tworzenie programów rezydentnych

Część rezydująca programu jest wywoływana jako procedura obsługi przerwania. Przerwanie, a w szczególności przerwanie sprzętowe, może pojawić się praktycznie w dowolnym momencie, a więc w dowolnym momencie może przerwać działanie innego programu. Aby po powrocie z procedury obsługi przerwania (programu rezydentnego) przerwany program mógł wznowić poprawnie działanie, stan procesora musi pozostać niezmieniony. Dlatego też każdy program rezydentny powinien zachowywać zawartość wszystkich rejestrów procesora.

Ponadto, jeżeli program rezydentny korzysta z usług systemu DOS (za pomocą przerwanie 21h), musi sprawdzać, czy w momencie jego wywołania nie była wykonywana jakąś funkcja DOS-u. Wynika to z faktu, że do funkcji DOS-u nie można wchodzić dwukrotnie (DOS nie jest wielobieżny (ang. *reentrant*)). Kłopotu tego można uniknąć albo przez nieużywanie usług systemowych w programie rezydentnym, albo wykorzystując przerwanie, które jest wywoływane tylko wtedy, gdy aktywacja programu rezydentnego jest "bezpieczna" (idealne jest tu przerwanie BIOS-u 16h).

Przerwania, przejmowane przez programy rezydentne bardzo często pełnią ważną funkcję w systemie. Jeżeli tak jest, program rezydentny powinien również wywoływać oryginalną (tzn. zastaną w momencie instalacji) procedurę obsługi przerwania.

2.3. Funkcja main jako program rezydentny

Funkcja **main** nie zachowuje oczywiście zawartości rejestrów procesora. Dlatego wektor przerwania nie może wskazywać bezpośrednio na nią, lecz na fragment kodu, który zachowa rejestry na stosie, wywoła funkcję **_main**, a po jej zakończeniu odtworzy zawartość rejestrów ze stosu. Kod ten powinien także zapewniać wykonanie oryginalnej procedury obsługi przerwania.

Żeby zabezpieczyć się przed zapętleniem wywołań programu rezydentnego (wywołaniem programu w trakcie jego działania), powinien on posiadać jakiś wskaźnik aktywności. Przed aktywacją program sprawdzałby czy wskaźnik jest wyzerowany i tylko w takim przypadku uaktywniałby się ustawiając jednocześnie wskaźnik na 1.

Należy jeszcze zastanowić się nad zawartością rejestrów segmentowych w chwili wywołania programu rezydującego w pamięci. Ponieważ obsługa przerwania przez procesor polega na wykonaniu dalekiego wywołania procedury, rejestr segmentowy kodu CS będzie wskazywał segment, w którym znajduje się program rezydentny. Rejestry segmentowe danych DS i ES, będą oczywiście wskazywać na segment(-y) danych programu przerwanego. Zakładamy, że program w C będziemy kompilować w modelu *Small* (tworzenie rezydentów większych niż 64K nie ma sensu). W takim przypadku dane znajdują się w tym

samym segmencie co kod a więc do rejestrów DS oraz ES należy przed wywołaniem funkcji **_main** przepisać zawartość rejestru CS.

Poniżej przedstawiam fragment kodu, realizujący wszystkie opisane powyżej działania.

```
Interrupt16   proc    far
              cli               ; zablokuj przerwania dopóki
wskaźnik
                                ; aktywności nie jest ustawiony
              cmp     cs:TSR, 1 ; czy program aktywny ?
              je      end
              mov     cs:TSR, 1 ; ustaw wskaźnik aktywności
              sti               ; można odblokować przerwania
              push    ax        ; zachowaj rejestry na stosie
              push    bx
              push    cx
              push    dx
              push    di
              push    si
              push    ds
              push    es
              push    bp
              push    cs        ; inicjuj rejestry segmentowe
danych
              pop     ds
              push    cs
              pop     es

              call _main        ; wywołaj program główny

              pop     bp        ; odtwórz zawartość rejestrów
              pop     es
              pop     ds
              pop     si
              pop     di
              pop     dx
              pop     cx
              pop     bx
              pop     ax

              mov     cs:TSR, 0      ; wyzeruj wskaźnik
aktywności
end:          sti
              jmp     cs:Original16  ; skocz do oryginalnej
procedury
              endp                   ; obsługi przerwania 16h
```

Odwołania do zmiennych (TSR i Original16) na początku i końcu programu są prefiksowane rejestrem CS, gdyż w tych miejscach rejestr DS, używany domyślnie do adresowania pamięci, wskazuje na segment danych programu przerwanego.

Powyższy kod będzie jedną z części nowego Startup-a.

Druga część Startup-a musi zapamiętać zastany wektor przerwania 16h w zmiennej Original16, ustawić wektor tego przerwania tak, żeby wskazywał na zdefiniowaną powyżej procedurę Interrupt16, i powrócić do DOS-u pozostawiając program w pamięci.

Do pozostawienia programu rezydującego w pamięci, użyjemy funkcji DOS-u 31h (*Keep*). Funkcji tej należy podać w rejestrze DX liczbę paragrafów 16-bajtowych które będzie zajmować program rezydujący. Wartość ta jest definiowana przez programistę w zmiennej globalnej size w programie w C.

```
extrn _main:near
extrn _size:word
extrn _autor:near
.model TINY

.CODE
org 100h                                    ; offset kodu 100h
gdyż                                            ; program typu COM
Start:

;--- zachowaj oryginalny wektor przerwania 16h

     mov     ax, 3516h                 ;pobierz wektor 16h
     int     21h
     mov     word ptr Original16, bx   ;zachowaj offset wektora
     mov     word ptr Original16+2, es ;zachowaj segment wektora

;--- załaduj nowy wektor

   mov     ax, 2516h                 ;funkcja 25h i wektor 16h
   mov     dx, offset Interrupt16    ;offset w dx, segment w ds
        int     21h

;--- terminate and stay resident

   mov     ax, 3100h                 ;funkcja 31h kod powrotu 0
   mov     dx,_size                  ;ile paragrafów zostawić

   int     21h
```

Moglibyśmy już połączyć obie części i utworzyć nowy Startup. Zanim jednak to zrobimy dodajmy jeszcze dwie proste i pożyteczne funkcje.

Przed uruchomieniem każdego programu system operacyjny tworzy kopię środowiska. Kopia taka zostanie utworzona także dla programu rezydentnego; co gorsza, pozostanie ona, zupełnie niepotrzebnie, w pamięci. Dopiszmy więc do naszego Startup-a żądanie zwolnienia pamięci zajętej przez kopię środowiska. Adres zawierającego ją segmentu umieszczony jest w słowie przesuniętym o 2Ch bajtów względem początku prefiksu segmentu programu (PSP).

```
mov      bx,2Ch              ;zwolnij pamięć zajmowaną przez
mov      ax,[word ptr bx]     ;kopię środowiska
mov      es,ax                ;adres segmentu środowiska do ES
mov      ah,49h               ;zwolnij segment pamięci
int      21h                  ;wskazywany przez ES
```

Drugim zadaniem kodu startowego będzie wypisanie wizytówki programu rezydentnego w czasie jego instalacji. Załóżmy, że będzie to łańcuch wskazywany przez globalną zmienną programu w C o nazwie autor. Do wypisania tego łańcucha użyjemy funkcji 09h DOS-u (funkcja ta oczekuje, że łańcuch będzie zakończony znakiem $).

```
     mov       ah,09h
```

```
    mov       dx,[word PTR _autor]       ;wypisz łańcuch
    int       21h
```

Oto pełna wersja Startup-a instalującego funkcję **main** jako program rezydentny przechwytujący przerwanie 16h :

```
; plik TSR16.asm
extrn _main:near
extrn _size:word
extrn _autor:near
.model TINY

.CODE
org 100h

Start:

;--- zachowaj oryginalny wektor przerwania 16h

    mov     ax, 3516h                ;funkcja 35h i wektor 16h
        int     21h
    mov     word ptr Original16, bx   ;zachowaj offset wektora
    mov     word ptr Original16+2, es ;zachowaj segment wektora

;--- załaduj nowy wektor

    mov     ax, 2516h                ;funkcja 25h i wektor 16h
    mov     dx, offset Interrupt16    ;offset w dx, segment w ds
        int     21h

;--- wypisz wizytówkę

        mov     ah,09h
        mov     dx,[word PTR _autor]      ;wypisz łańcuch
        int     21h

;--- zwolnij pamięć środowiska

     mov     bx,2Ch                   ;zwolnij pamięć zajmowaną
     mov     ax,[word ptr bx]         ;przez kopię środowiska
        mov     es,ax
        mov     ah,49h                   ;funkcja zwolnij pamięć
        int     21h

;--- terminate and stay resident

      mov     ax, 3100h              ;funkcja 31h kod powrotu 0
      mov     dx,_size                  ;ile paragrafów zostawić
        int     21h

Interrupt16 proc    far
            cli             ; zablokuj przerwania dopóki wskaźnik
                            ; aktywności nie jest ustawiony
            cmp     cs:TSR, 1  ; czy aktywny ?
            je      end
            mov     cs:TSR, 1  ; ustaw wskaźnik aktywności
            sti                ; można odblokować przerwania
            push    ax         ; rejestry na stos
            push    bx
            push    cx
            push    dx
            push    di
            push    si
            push    ds
```

```
                push    es
                push    bp

                push    cs          ; inicjuj rejestry segmentowe
danych
                pop     ds
                push    cs
                pop     es

                call    _main       ; wywołaj program główny

                pop     bp          ; odtwórz zawartość rejestrów
                pop     es
                pop     ds
                pop     si
                pop     di
                pop     dx
                pop     cx
                pop     bx
                pop     ax

                mov     cs:TSR, 0   ; zaznacz, że nieaktywny

end:            sti
                jmp     cs:Original16      ; skocz do oryginalnej
procedury
                endp                       ; obsługi przerwania 16h
TSR                     label BYTE
                        db 0
Original16              label Dword
                        dw ?               ;offset
                        dw ?               ;segment
END Start
```

W powyższej wersji funkcja **main** zostaje "podpięta" do przerwania 16h. Na dyskietce dołączonej do książki znajdują się także wersje wykorzystująca przerwanie zegarowe 1Ch i przerwanie sprzętowe klawiatury 09h. Wersja dla przerwania zegarowego jest dokładnym odpowiednikiem wersji dla przerwania 16h. W przypadku przerwania klawiatury 09h, przed wywołaniem funkcji **main** wywoływana jest oryginalna procedura obsługi tego przerwania. Inaczej funkcja **main** nie mogłaby odczytać kodu klawisza, którego naciśnięcie spowodowało jej uaktywnienie.

Dodatkowo dyskietka zawiera wszystkie wyżej wspominane programy rozbudowane o możliwość deinstalacji programu rezydentnego. Deinstalacja następuję po wywołaniu programu z opcją `/u` lub /U i jest możliwa tylko wtedy, gdy po zainstalowaniu programu jegoprzerwanie nie zostało przejęte (np. przez inny program rezydentny).

Żeby ułatwić korzystanie ze zdefiniowanych wyżej modułów przedstawiam poniżej dwa programiki wsadowe tworzące automatycznie program rezydentny korzystający z wybranego przerwania.

```
REM plik TSRMS.BAT dla środowiska Microsoft C
@if %1==1c goto 1c
@if %1==16 goto 16
@if %1==9 goto 9
```

```
@echo Syntax : tsr 1c|16|9 plik.obj [ pliki.obj ]
@goto koniec
:16
link /NOE tsr16 %2 %3 %4 %5 %6 %7 %8 %9, %2 , , slibce.lib
@goto end
:1c
link /NOE tsr1c %2 %3 %4 %5 %6 %7 %8 %9, %2 , , slibce.lib
@goto end
:9
link /NOE tsr9  %2 %3 %4 %5 %6 %7 %8 %9, %2 , , slibce.lib
@goto end
:end
exe2bin %2
:koniec
REM program TSRTC.BAT dla środowiska Turbo C
@if %1==1c goto 1c
@if %1==16 goto 16
@if %1==9 goto 9
@echo Syntax : tsr 1c|16|9  plik.obj [ pliki.obj ]
@goto end
:16
tlink /t tsr16 %2 %3 %4 %5 %6 %7 %8 %9 , %2 , , cs.lib
@goto end
:1c
tlink /t tsr1c %2 %3 %4 %5 %6 %7 %8 %9 , %2 , , cs.lib
@goto end
:9
tlink /t tsr9  %2 %3 %4 %5 %6 %7 %8 %9 , %2 , , cs.lib
@goto end
:end
```

Nazwy plików należy uzupełnić ewentualnymi ścieżkami dostępu. Przykładowe wywołanie powyższych programów może wyglądać następująco (zakładam ogólną nazwę TSR zamiast TSRMS czy TSRTC):

```
tsr 16 program
```

3. Przykładowe programy rezydentne

Używając zdefiniowanego powyżej kodu Startup, programista nie musi martwić się o żadne kwestie wynikające z faktu, że program ma działać jako rezydent. Program napisany w normalny sposób i uruchomiony przy pomocy debuggera, np. w środowisku zintegrowanym kompilatora, wystarczy jedynie ponownie skonsolidować, np.:

```
tlink /t tsr16.obj program , program
```

W ten sposób powstanie program typu COM (użyto opcji `/t` programu TLINK) o nazwie **program.com**. Po uruchomieniu tego programu na ekranie zostanie wypisany komunikat przypisany zdefiniowanej w programie zmiennej globalnej autor, a program zostanie zainstalowany w pamięci jako program rezydentny wywoływany przerwaniem 16h. Od tego momentu każde wygenerowaniu przerwania 16h spowoduje wywołanie funkcji **main** programu.

Tak więc teoretycznie pisanie programów rezydentnych nie różni się niczym od pisania innych programów. W praktyce, programom takim stawia się trochę inne wymagania.

Przede wszystkim zajmują one stale miejsce w pamięci komputera, a więc powinny być małe. Z tego powodu niedopuszczalne jest użycie wielu funkcji z bibliotek standardowych, które nie były projektowane do zastosowania w programach rezydentnych i dają duży kod wynikowy. Dotyczy to szczególnie funkcji wejścia/wyjścia, które trzeba zastąpić własnymi. Prostą funkcję wejściowa zdefiniowaliśmy już we wcześniejszej części tego rozdziału; była to funkcja **_getch** zwracająca kod znaku wprowadzonego z klawiatury. Funkcja ta w razie potrzeby czeka na naciśnięcie klawisza. Poniżej przedstawiam drugą przydatną funkcję wejścia.

```
 ; plik checkch.asm
.model SMALL
.CODE

proc _checkch      ; funkcja zwraca znak z bufora
                     klawiatury
        mov ah,1   ; lub -1 jezeli bufor jest pusty
        int 16h    ; nie usuwa znaku z bufora
        jnz koniec
        mov ax,-1
koniec: ret
        endp

public _checkch
end
```

Funkcja **checkch** zwraca kod znaku znajdującego się buforze klawiatury (nie usuwając go z bufora) lub wartość -1 gdy w buforze nie ma żadnego znaku.

Jako mechanizm wyjścia najlepiej używać w programach rezydentach bezpośrednich odwołań do pamięci ekranu. Aby program był uniwersalny, powinien on samodzielnie ustalać adres pamięci ekranu w zależności od karty graficznej. Można to zrobić umieszczając na początku funkcji **main** następującą sekwencję:

```
#define MK_FP(segment,offset) ((void far*)\
           (((unsigned long)(segment) << 16 ) |
(unsigned)(offset)))

   /* ... */
              main()
              {
               if(*(char far *)0x449==7)
                scr=MK_FP(0xb000,0x0000);
               else
                scr=MK_FP(0xb800,0x0000);
   /* ... */
              }
```

Wartością makrodefinicji MK_FP jest daleki wskaźnik do miejsca w pamięci o adresie podanym w postaci segment:offset. W warunku **if** sprawdzana jest wartość zmiennej z obszaru danych BIOS-u określającej bieżący tryb graficzny.

W trybie o numerze 7, odpowiadającym kartom monochromatycznym, pamięć ekranu zaczyna się od adresu 0xb000:0000; w pozostałych trybach - od adresu 0xb800:0000.

Jeżeli program rezydentny ma pełnić funkcję procesu działającego w tle, należy także zadbać by był on wystarczająco szybki i nie spowalniał pracy komputera.

Przykład pierwszy

Z takimi założeniami możemy napisać pierwszy, prosty program rezydentny. Jak przystało na książkę poświęconą językowi C, program będzie pomocny przy programowaniu w tym języku. Będzie to nakładka na edytor, podświetlająca słowa kluczowe języka C w tekście wyświetlanym na ekranie. Program, wykorzystujący przerwanie zegarowe 1Ch będzie regularnie przeglądał ekran w poszukiwaniu słów kluczowych C, a w przypadku znalezienia takiego słowa będzie odpowiednio zmieniał atrybuty ekranu.

Zacznijmy od definicji globalnych. Dla potrzeb kodu startowego musimy zdefiniować zmienną size, określającą rozmiar programu w paragrafach i zmienną autor, wskazującą na łańcuch wyświetlany podczas instalacji.

```
int size=0x50;
char *autor="Adam Sapek    C Key Words$";
```

Należy pamiętać, by łańcuch wskazywany przez zmienną autor kończył się znakiem $.

Wartość zmiennej size dobiera się w następujący sposób. Gotowy program należy skompilować i dołączyć do niego odpowiedni kod startowy. Do długości otrzymanego programu ***.COM** należy dodać 256, do tego dodać sumaryczną długość (w bajtach) zmiennych **globalnych**, którym nie jest **jawnie** nadawana w definicji wartość (wygodniej i bezpieczniej jest jawnie nadać wartość wszystkim zmiennym). Tak otrzymaną wartość należy podzielić przez 16 (zaokrąglając w górę).

Jeżeli po zainstalowaniu programu następuje zawieszenie systemu, prawdopodobne jest, że wartość zmiennej size jest za mała. Jeżeli program rezydentny po zainstalowaniu działa niepoprawnie, ale nie powoduje załamania systemu, należy raczej poszukać błędu w programie, niż zwiększać wartość zmiennej size. Częstym powodem niepoprawnego działania programu rezydentnego, podczas gdy wersja nierezydentna działa bez zarzutu, jest pominięcie inicjacji zmiennych globalnych w funkcji **main**. W normalnym programie zmienne globalne mają na początku funkcji **main** wartość nadaną w definicji lub zero, natomiast w programie rezydentnym zmienne te mogą zawierać wartości przypadkowe pozostałe po poprzednim wywołaniu.

Słowa kluczowe języka C są zgromadzone w tablicy wskaźników znakowych.

```
char *listtab[]={"for","int","if","char","while","case","do",
                 "else","return","void","default","struct",
                 "switch","far","extern","break","goto",
                 "float","const","continue","double","union",
```

```
                    "unsigned","static","sizeof","long","short",
                    "auto","register","typedef",0},**list;
```

Kolejność słów w tablicy jest taka, by słowa częściej występujące znajdowały się bliżej początku tablicy. Dzięki temu zmniejsza się średni czas przeszukiwania tablicy.

Żeby odnaleźć na ekranie słowa kluczowe języka C, program będzie szukał małych liter, sprawdzał czy są one początkiem słowa (czy nie są poprzedzone literą lub podkreśleniem) i porównywał kolejne słowa kluczowe z tablicy ze znalezionym słowem.

Jeżeli jedno ze znajdujących się w tablicy słów będzie odpowiadało słowu znalezionemu na ekranie, program sprawdzi jeszcze, czy słowo na ekranie **kończy się w tym miejscu**. Jest to konieczne aby nie wyróżnić na przykład początku nazwy zmiennej:

```
char  inteligencja;
```

jako słowa kluczowego **int**.

Jak widać z powyższego skróconego opisu algorytmu, program ma do wykonania dosyć dużo pracy. Ponieważ będzie on wykorzystywał przerwanie zegarowe, całą tę pracę będzie wykonywał około 18 razy na sekundę. Jeżeli ktoś uważa, że to za dużo, aby program został niezauważony, to ma rację: na wolnym komputerze zainstalowanie go spowoduje wyraźne spowolnienie pracy. Pojawia się tutaj zasygnalizowany wcześniej problem szybkości programów rezydentnych działających w tle. W tym przypadku można próbować zmieniać algorytm przeszukiwania tablicy słów kluczowych , np. zamiast przeszukiwania liniowego zastosować przeszukiwanie rozproszone. Moim zdaniem istnieje jednak prostsze rozwiązanie: skoro program wywoływany jest za często żeby wykonać tak dużą pracę, to należy podzielić ją na fragmenty, które będą wykonywane w kolejnych wywołaniach. W naszym przypadku program może jednorazowo przeglądać tylko jedną czwartą ekranu. W ten sposób cały ekran będzie aktualizowany mniej więcej cztery razy na sekundę, co w zupełności wystarczy.

Aby w kolejnych wywołaniach program przeglądał kolejne ćwiartki ekranu, zmienna służąca do adresowania pamięci ekranu nie będzie nigdy inicjowana, a jedynie ciągle zwiększana i dzielona modulo 4000. W ten sposób kolejne wywołanie programu zacznie przeszukiwanie ekranu w tym miejscu, w którym skończyło poprzednie, a zmienna adresująca ekran nie wyjdzie nigdy poza dozwolony zakres.

Kompletny tekst programu przytoczono poniżej.

```
/* plik cwords.c */
#define MK_FP(seg,ofs) ((void far*)\
                       (((unsigned long)(seg) << 16 ) |
(unsigned)(ofs)))

char *autor="Adam Sapek    C Key Words (93)$";

int size=65,                                /* rozmiar TSR-a w
paragrafach */
    x,i,off=0,
```

```
   _acrtused=0;                              /* potrzebne przy
Microsoft C  */

unsigned char far *screen,far *scr;

char *listtab[]={"for","int","if","char","while","case","do",
              "else","return","void","default","struct",
              "switch","far","extern","break","goto",
              "float","const","continue","double","union",
              "unsigned","static","sizeof","long","short",
              "auto","register","typedef",0},**list;

void main()
{
char c;

if(*(char far *)0x449==7)                       /* gdzie jest
pamięć video ? */
  scr=MK_FP(0xb000,0x0000);                     /* Hercules, MDA
-> B000:0  */
 else
  scr=MK_FP(0xb800,0x0000);                     /* VGA, EGA,
itp. -> B800:0  */
for(x=0;x<500;x++,off=(off+2)%4000)
{
 screen=scr+off;
 if((unsigned char)(*screen-'a')<='z'-'a')       /* jeżeli mała
litera       */
  {
   c=screen[-2];                                 /* poprzedni
znak          */
   if((unsigned char)(c-'a')>'z'-'a' &&          /* nie jest
literą ani _    */
      (unsigned char)(c-'A')>'Z'-'A' && c!='_' )
    {
     list=listtab;
     while(*list)
      {
      i=0;
      while(screen[2*i]==(*list)[i])          /* porównaj napis
na ekranie   */
         i++;                                 /* z kolejnym
słowem kluczowym */
      if(!(*list)[i])                         /* jeżeli zgodne
*/
       {
         c=screen[2*i];                          /* następny
znak na ekranie */
         if((unsigned char)(c-'a')>'z'-'a' &&    /* jeżeli nie
jest literą,  */
            (unsigned char)(c-'A')>'Z'-'A' &&    /* cyfrą ani
podkreśleniem  */
            (unsigned char)(c-'0')>'9'-'0' && c!='_' )
              {
               screen++;
               while(i)
                 screen[2*(--i)]|=0xF;      /* to rozjaśnij
znalezione słowo */
               break;
              }
       }
```

```
        list++;                          /* sprawdz następne
słowo z listy */
      }
    }
  }
}
}
```

Program należy skompilować w modelu *Small* a następnie utworzyć program COM poleceniem:

```
tsr 1c cwords
```

Po uruchomieniu uzyskanego w ten sposób programu, na ekranie pojawi się napis:

```
Adam Sapek    C Key Words (93)
```

i od tego momentu każde słowo kluczowe języka C będzie wypisywane się na ekranie rozjaśnionym atrybutem.

Przykład drugi

Poprzedni program należy do klasy "rezydentów" wykonujących w tle jakąś pracę. Inną kategorią są programy "czuwające", które uaktywniają się tylko w konkretnej sytuacji. Bardzo często powodem uaktywnienia programu rezydentnego jest naciśnięcie jakiegoś klawisza. W takim przypadku istnieje dwie możliwości: program może przechwycić przerwanie sprzętowe klawiatury 09h lub przerwanie BIOS-u 16h.

Przerwanie 09h jest przerwaniem sprzętowym, generowanym zawsze przy naciśnięciu (i puszczeniu) klawisza. Standardowa procedura obsługi tego przerwania sprawdza, który klawisz został naciśnięty i wpisuje kod odpowiedniego znaku do bufora klawiatury.

Z kolei przerwanie 16h jest przerwaniem programowym. Procedura obsługi tego przerwania realizuje między innymi funkcje pobierania danych wprowadzanych z klawiatury. W przeciwieństwie do obsługi wyprowadzenia danych na ekran, gdzie BIOS jest często omijany a programy odwołują się bezpośrednio do pamięci video, zdecydowana większość (praktycznie wszystkie) programy realizują odczytywanie klawiatury przy użyciu funkcji przerwania 16h. Funkcje przerwania 16h umożliwiają sprawdzenie, czy w buforze klawiatury znajduje się jakiś znak i pobranie znaku z bufora. Jeżeli funkcja pobierająca znak zostanie wywołana gdy bufor jest pusty, to będzie ona czekała na wprowadzenie znaku z klawiatury. Łatwo sobie wyobrazić, że gdyby programy, chcąc pobrać znak z klawiatury, wywoływały od razu funkcję pobrania znaku, to instalowanie programu rezydentnego wykorzystującego przerwanie 16h byłoby pozbawione sensu. Na szczęście, powszechnie przestrzegana jest zasada, że najpierw w pętli wywołuje się funkcję sprawdzającą, czy w buforze klawiatury jest znak do pobrania, a dopiero po pojawieniu się znaku pobiera się go funkcją 00h. Dzięki te-

mu możemy być prawie pewni, że przerwanie 16h będzie wywoływane dosyć regularnie.

Wróćmy do wyboru przerwania dla programu uaktywnianego naciśnięciem klawisza. Uważam, że zawsze, gdy to jest możliwe, należy unikać przejmowania przerwań sprzętowych. Możliwość uruchomienia w każdej chwili programu "podwieszonego" pod takie przerwanie w większości przypadków jest wadą, a nie zaletą. W praktyce, program wykorzystujący przerwanie 16h nie zostanie wywołany tylko w czasie trwania operacji dyskowych. Uaktywnienie programu w takiej chwili jest raczej niepożądane, gdyż niepotrzebnie zwiększa ryzyko utraty danych (zwiększa się prawdopodobieństwo wystąpienia awarii, gdy plik nie jest całkowicie zapisany). Istnieją oczywiście sytuacje, w których trzeba wykorzystać przerwanie sprzętowe. Miałem okazję kiedyś pisać program rezydentny, będący nakładką na pewien nieprofesjonalny program. "Rezydent" miał umożliwiać wpisywanie często powtarzających się danych przy pomocy naciśnięcia jednego klawisza. Po napisaniu programu okazało się, że współpracuje on ze wszystkim z wyjątkiem programu, dla którego był przeznaczony. Powodem był fakt, że program ten nie wywoływał funkcji sprawdzającej w buforze obecność znaku, lecz od razu pobierał ten znak. W ten sposób znaki wpisywane z klawiatury były od razu odczytywane z bufora przez funkcję czytającą, która na nie cały czas czekała, zaś program rezydentny nie miał szans na "zauważenie" żadnego znaku. Po zamianie przerwania 16h na przerwanie sprzętowe klawiatury 09h programy współpracowały bez zarzutu.

Instalując program korzystający z przerwania klawiatury trzeba pamiętać, że może on zostać wywołany w dowolnym momencie, a więc także z wnętrza DOS-u. Wywołanie programu wykorzystującego funkcje usługowe DOS-u (przerwanie 21h) w trakcie wykonywania którejś z nich spowoduje najprawdopodobniej załamanie systemu. Jednym ze sposobów zabezpieczenia się przed taką sytuacją, jest sprawdzanie przed uruchomieniem programu rezydującego, czy wykonywana jest jakaś funkcja DOS-u. Wykorzystuje się do tego wewnętrzny znacznik DOS-u InDOS, którego adres zwraca funkcja 34h w rejestrach ES:BX. Wartość tego znacznika równa zero oznacza, że nie jest wykonywana żadna funkcja DOS-u. Znacznik InDOS jest ustawiony także wtedy, gdy DOS, nic nie robiąc, czeka na naciśnięcie klawisza. Ponieważ system znajduje się wówczas w jałowej pętli, uruchomienie programu rezydentnego nie może mu zaszkodzić. Aby umożliwić wywołanie programu w takiej sytuacji, wykorzystuje się przerwanie 28h, generowane przez system w trakcie oczekiwania na wprowadzenie danych z klawiatury.

Zasygnalizowane powyżej problemy związane z uruchamianiem programów rezydentnych, szczególnie wykorzystujących przerwanie 09h, są bardzo rozległym zagadnieniem. Po wyczerpujące informacje radzę sięgnąć do specjalistycznej literatury dotyczącej systemu DOS.

Program, który poniżej przedstawię, nie będzie uaktywniał się po naciśnięciu klawisza, a mimo to będzie stale czytał klawiaturę i będzie wywoływany prze-

rwaniem 16h. Jego zadaniem jest wygaszenie ekranu jeżeli przez trzy minuty nie zostanie naciśnięty żaden klawisz. Program ten będzie korzystał z kilku funkcji napisanych w asemblerze (w tym ze zdefiniowanych wcześniej w tym rozdziale funkcji **_getch** i **checkch**), a także ze standardowych funkcji **C**.

Wygaszenie ekranu może polegać na wpisaniu odpowiednich wartości do obszaru pamięci video, natomiast do wygaszenia kursora konieczne jest skorzystanie z funkcji BIOS-u.

Poniżej przedstawiam definicje dwóch funkcji służących do wygaszenia i wyświetlania kursora.

```
; plik kursor.asm
   .MODEL SMALL
   .CODE

   proc _hide_cursor     ; funkcja chowa kursor
        mov  ah,03h
        mov  bh,0h
        int  10h         ; pobierz opis kursora
        or   ch,20h      ; ustaw atrybuty na
"niewidoczny"
        mov  ah,1h
        int  10h         ; zapisz opis kursora
        ret
   endp

   proc _show_cursor     ; funkcja pokazuje kursor
        mov  ah,03h
        mov  bh,0h
        int  10h         ; pobierz opis kursora
        and  ch,0dfh     ; kasuj atrybut "niewidoczny"
        mov  ah,1h
        int  10h         ; zapisz opis kursora
        ret
   endp
   public _show_cursor, _hide_cursor
   end
```

Funkcja **hide_cursor** pobiera słowo opisujące kształt i atrybuty kursora i ustawia atrybuty tak by kursor był niewidoczny, natomiast funkcja **show_kursor** przywraca widoczność kursora. W ten sposób para wywołań funkcji:

```
hide_cursor();
/* ... */
show_cursor();
```

nie powoduje zmiany kształtu kursora.

Jak już wspomniałem, w programie zostaną użyte także funkcje z bibliotek standardowych języka C. W tym przypadku będzie to generator liczb pseudolosowych.

Wzmiankowałem wcześniej, że wiele funkcji standardowych daje zbyt długi kod jak na programy rezydentne. Nie dotyczy to jednak wszystkich funkcji. Jeżeli istnieje potrzeba zastosowania funkcji bibliotecznych należy wybierać funkcję jak najniższego poziomu. Na przykład do obsługi plików lepiej użyć "bliskich systemowi" funkcji zdefiniowanych w pliku **io.h** niż funkcji zdefiniowanych w **stdio.h**.

Przy próbie dołączania niektórych funkcji z bibliotek standardowych konsolidator może zgłosić komunikat o braku definicji pewnych symboli. Prawdopodobnie będą to zmienne definiowane w oryginalnym Startup-ie. Można w takim przypadku spróbować zdefiniowania zmiennych o takich nazwach w programie głównym i zobaczyć, czy program będzie działał poprawnie (zwykle będzie).

Wspominałem już wcześniej, że wyjście na ekran najlepiej realizować w programach rezydentnych przez bezpośrednie odwołania do pamięci video. Pamięć ekranu można reprezentować na wiele sposobów.

W poniższym programie zmienna screen jest zadeklarowana następująco:

```
char far *screen;
```

a więc pamięć ekranu jest reprezentowana przez daleki wskaźnik znakowy, co umożliwia łatwe przepisanie jego zawartości. W razie potrzeby można definiować bardziej wyrafinowane struktury odwzorowujące ekran, np.

```
int (far *screen)[80];
```

W zasięgu tej definicji można odwoływać się do konkretnego znaku na ekranie przy pomocy jego współrzędnych, np.

```
screen[10][40]=0x0F41;
```

Należy przy tym pamiętać, że pierwszy indeks odpowiada współrzędnej Y, a starszy bajt wpisywanego słowa określa atrybuty wyświetlanego znaku.

Żeby jeszcze bardziej zbliżyć się do "normalnego spojrzenia" na ekran, można zdefiniować go jako tablicę struktur. Struktura będzie opisywać każdy znak na ekranie jako kod znaku i atrybut:

```
typedef struct {
                char code,attr;
               }character;

character (far *screen)[80];
```

W zasięgu takiej definicji można odwoływać się zarówno do atrybutów, jak i kodów konkretnych znaków przez ich współrzędne, np.:

```
for(x=0;x<80;x++)
  if(screen[10][x].code=='A')screen[10][x].attr=0x0f;
/* rozjaśnij litery 'A' w jedenastym wierszu od góry */
```

Oto pełny tekst programu *saver* wygaszającego ekran, po około 3 minutach od ostatniego naciśnięcia klawisza.

```
/* plik saver.c */
#include <stdlib.h>

#define MK_FP(seg,ofs) ((void far*)\
                       (((unsigned long)(seg) << 16 ) |
(unsigned)(ofs)))

int size=80,                                /* rozmiar TSR-a w
paragrafach  */
    _acrtused=0;                            /* potrzebne w
Microsoft C      */

char *autor="Adam Sapek   Screen Saver (C) 92$";
unsigned char far *timer_ticks=MK_FP(0x0040,0x006c),
/* zegar */
             far *screen,
             far *screen2;

struct {
        unsigned char x,y;
       }snake[4]={{4,4},{3,3},{2,2},{1,1}};

char dx[8]={0,1,2,1,0,-1,-2,-1},
     dy[8]={-1,-1,0,1,1,1,0,-1};

unsigned off,time=300;

void at(char x,char y)                          /* ustaw
współrzędne (x,y) */
{
 off=160*y+2*x;
}
void print(char *s)                     /* pisz na ekranie ciąg
dwóch znaków */
{
 screen[off]=s[0];
 screen[off+2]=s[1];
}
void move(unsigned char far* e1,unsigned char far *e2)   /*
przepisz e2 do e1 */
{
 int x;
 for(x=0;x<4000;x++)
   e1[x]=e2[x],e2[x]=(x%2)?7:32;
}
void main()
{
 char direct=3,y,x,a;

 if(checkch()!=-1||time==300)             /* początek lub
naciśnięto klawisz */
    { time=timer_ticks[1]; srand(*timer_ticks); } /* zapamiętaj
czas posiej   */
                                                 /* generator
liczb losowych */
 if((unsigned char)(timer_ticks[1]-time)>=13)    /* czy minęły
3 min. ?      */
  {
```

```
   if(*(char far *)0x449==7)                    /* gdzie
pamięć video ?     */
      screen=MK_FP(0xb000,0x0000);              /* Hercules ->
B000:0       */
   else
      screen=MK_FP(0xb800,0x0000);              /* VGA itp. ->
B000:0       */
   screen2=screen+4000;                         /* screen2 -
druga strona   */

   move(screen2,screen);                       /* przepisz
zawartość ekranu */

   hide_cursor();                              /* schowaj
kursor            */

   while(checkch()==-1)                        /* aż nie
zosatnie naciśnięty*/
    {                                          /* jakis
klawisz             */
     do{  if(rand()%11==0)direct=rand()%8;     /* średnio co
11 kroków nowy */
          x=snake[0].x+dx[direct];             /* kierunek
*/
          y=snake[0].y+dy[direct];
       }while(x<0||x>=79||y<0||y>24);          /* czy można w
tym kierunku  */

     at(x,y);                  print("▓▓");      /* rysuj
węża            */
     at(snake[0].x,snake[0].y); print("▓▓");
     at(snake[1].x,snake[1].y); print("▒▒");
     at(snake[2].x,snake[2].y); print("▒▒");
     at(snake[3].x,snake[3].y); print("  ");

     for(a=3;a>=1;a--)                         /* przesuń węża
*/
       {
        snake[a].x=snake[a-1].x;
        snake[a].y=snake[a-1].y;
       }
     snake[0].x=x; snake[0].y=y;

     time=*timer_ticks;
     while(*timer_ticks-time<4);              /* poczekaj 4
takty zagara   */
    }
    _getch();              /* pobierz znak z bufora     */
    move(screen,screen2);  /* odtwórz ekran             */
    time=timer_ticks[1];
    show_cursor();        /* pokaż kursor              */
 }
}
```

W celu uzyskania programu SAVER.COM, instalowanego jako program rezydentny przechwytujący przerwanie 16h, należy skompilować plik SAVER.C w modelu *Small*, po czym połączyć otrzymany plik SAVER.OBJ z odpowiednim kodem startowym i bibliotecznymi:

```
tsr 16 saver getch checkch kursor
```