ROZDZIAŁ OSIEMNASTY: **PROGRAMY REZYDENTNE**

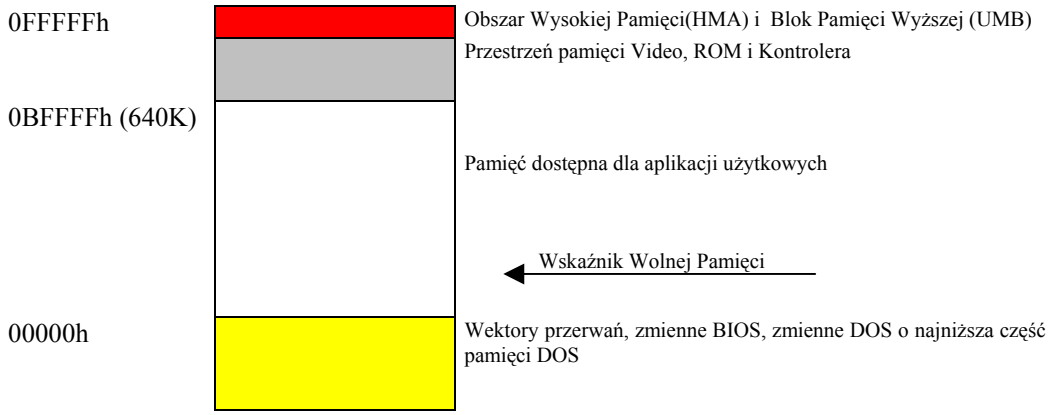
Większość aplikacji MS-DOS jest nierezydentnych. Ładują się one do pamięci, wykonują, kończą a DOS używa zaalokowanej pamięci aplikacji dla kolejnego programu wykonywanego przez użytkownika. Programy rezydentne stosują te same zasady , z wyjątkiem tej ostatniej. Program rezydentny, po zakończeniu, nie zwraca całej pamięci doz powrotem do DOS'a. Zamiast tego część programu pozostaje rezydentna, gotów do reaktywacji przez jakiś inny program w przyszłym czasie.

Programy rezydentne ,również znane jako programy „zakończ i zostań" lub TSR, malutkiej ilości wielozadaniowości w jedno zadaniowym systemie operacyjnym. Dopóki Microsoft Windows będzie popularny, programy rezydentne będą najpopularniejszym sposobem zezwalania pozostawania aplikacjom wielodostępnym na koegzystencję w pamięci w jednym czasie. Chociaż Windows zmniejszył zapotrzebowanie na TSR'y do przetwarzania w tle, TSR'y są jeszcze doceniane przy pisaniu sterowników, narzędzi antywirusowych i łat programów. Rozdział ten omawia kwestie jakie wypada znać kiedy piszemy programy rezydentne

---

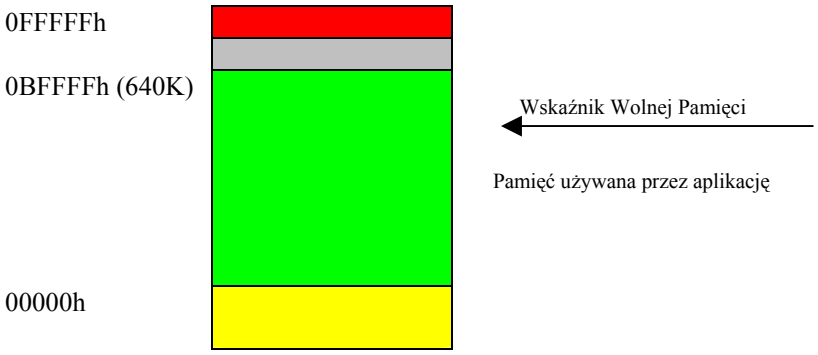
18.1 UŻYWANIE PAMIĘCI PRZEZ DOS I TSR'Y

Kiedy po raz pierwszy inicjujemy DOS'a, rozmieszczenia komórek pamięci wygląda następująco:



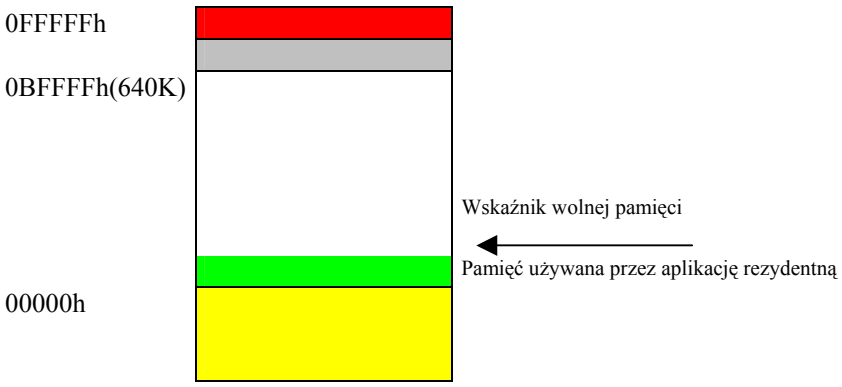
Mapa Pamięci DOS (brak aktywnej aplikacji)

DOS utrzymuje wskaźnik wolnej pamięci, który wskazuje początek bloku wolnej pamięci. Kiedy użytkownik uruchomi jakiś program użytkowy, DOS ładuje tą aplikację poczynając od adresu, który zawiera wskaźnik wolnej pamięci. Ponieważ DOS generalnie uruchamia tylko jedną aplikację w czasie, cała pamięć ze wskaźnika wolnej pamięci , do końca RAM (0BFFFFh) jest dostępna dla aplikacji:



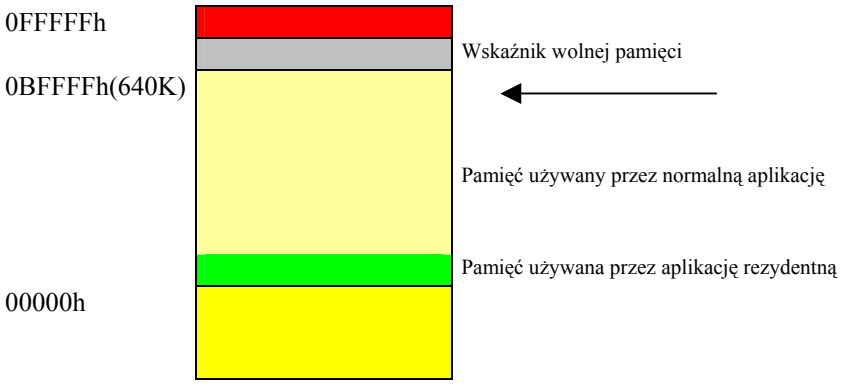
Mapa Pamięci DOS (uruchomiona aplikacja)

Kiedy program kończy się normalnie poprzez funkcję DOS'a 4Ch (makro Biblioteki Standardowej exitpgm), MS-DOS żąda zwrotu pamięci używanej przez aplikację i resetuje wskaźnik wolnej pamięci do poziomu niskiej pamięci DOS.
MS-DOS dostarcza drugiego wywołania zakończenia, który jest identyczny z wywołaniem zakończenia z jednym wyjątkiem, nie resetuje wskaźnika wolnej pamięci odbierając całą pamięć używaną przez aplikację. Zamiast tego ten TSR wywołuje zwolnienie określonego bloku pamięci. Wywołanie TSR'a (ah=31h) wymaga dwóch parametrów, kod procesu zakończenia w rejestrze al. (zazwyczaj zero) a dx musi zawierać rozmiar bloku pamięci do ochrony, w paragrafach. Kiedy DOS wykonuje ten kod, modyfikuje wskaźnik wolnej pamięci żeby wskazywał na lokację dx*16 bajtów powyżej PSP programu. To pozostawi pamięć jak pokazano:



Mapa Pamięci DOS (z aplikacją rezydentną)

Kiedy użytkownik wykonuje nową aplikację, DOS ładuje ją do pamięci pod nowy adres wskaźnika wolnej pamięci

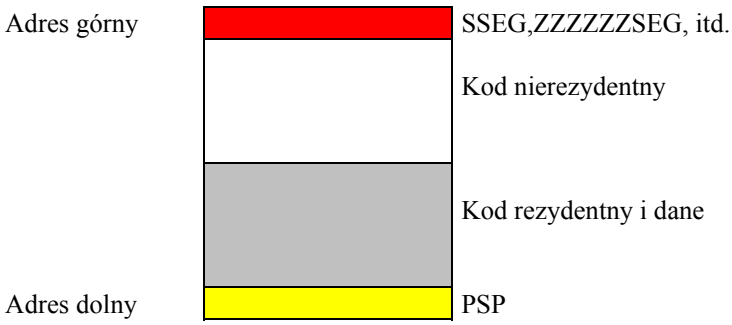


Mapa Pamięci DOS (z aplikacją rezydentną i normalną)

Kiedy ta zwykłą aplikacja się zakończy, DOS odbierze jej pamięć i ustawia wskaźnik wolnej pamięci na jej lokacji przed uruchomieniem aplikacji – powyżej programu rezydentnego. Poprzez zastosowanie schematu wskaźnika wolnej pamięci, DOS może chronić pamięć używaną przez program rezydentny.

Sztuczką z użyciem wywołani TSR'a jest obliczanie jak wiele paragrafów powinno pozostać rezydentnymi. Większość TSR'ów zawiera dwie sekcje kodu: część rezydentną i część nie rezydentną. Część rezydentna jest danym, głównym programem i wspiera podprogramy które wykonują się kiedy uruchamiamy program z linii poleceń. Kod ten prawdopodobnie nigdy nie wykona się ponownie. Dlatego też, nie powinniśmy zostawiać go w pamięci kiedy kończy się nasz program. W końcu każdy bajt zużyty przez program TSR jest jednym bajtem mniej dostępnym dla innych aplikacji.

Część rezydentna programu jest kodem, który pozostaje w pamięci i dostarcza jakichś funkcji koniecznych TSR'owi. Ponieważ PSP jest zazwyczaj przed pierwszym bajtem kodu programu, skuteczne użycie wywołania przez DOS TSR'a, nasz program musi być zorganizowane jak następuje:



Organizacja pamięci dla programu rezydentnego

Aby skutecznie używać TSR'a musimy zorganizować nasz kod i dane tak aby część kodu rezydentnego załadować pod dolny adres pamięci a część nie rezydentną załadować pod górny adres pamięci. MASM i Microsoft Linker dostarczają udogodnień, które pozwalają nam sterować porządkiem ładowania segmentów wewnątrz naszego kodu. Prostym rozwiązaniem, jednak, jest włożenie wszystkich naszych kodów rezydentnych i danych do pojedynczego segmentu i upewnić się, że ten segment pojawi się najpierw w każdym module źródłowym programu. W szczególności, jeśli używamy pliku SHELL.ASM z Biblioteki Standardowej UCR, musimy upewnić się, że zdefiniowaliśmy nasz rezydentny segment przed zawarciem dyrektyw dla plików biblioteki standardowej. W przeciwnym razie MS-DOS załaduje wszystkie podprogramy biblioteki standardowej przed segmentem rezydentnym co spowoduje zmarnowanie znacznej ilości pamięci. Zauważmy, że musimy tylko zdefiniować najpierw nasz segment rezydentny, nie musimy umieszczać wszystkich kodów rezydentnych i danych przed dołączeniami. Poniższe pracuje dobrze:

```
ResidentSeg     segment   para public  'rsiedent'
ResidentSeg     ends

EndResident     segment   para public  'EndRes'
EndResident     ends

                .xlist
                include           stdlib.a
                inlucdelib        stdlib.lib
                .list

ResidentSeg     segment   para  public  'resident'
                assume    cs: ResidentSeg, ds.:ResidentSeg

PSP             word              ?                   ,ta zmienna musi być tutaj

; Wkładamy kod rezydentny i dane tu

ResidentSeg     ends

dseg            segment   para  public  'data'
```

```
cseg            segment para public ‘code’
                asume   cs:cseg, ds:dseg

; Wkładamy tu kod nierezydentny

cseg            ends
                itd.
```

Cel segmentu EndResident stanie się jasny za chwilę. Po więcej informacji na temat porządku pamięci DOS zajrzyj do Rozdziału Szóstego.
Teraz jedynym problemem jest wyliczenie rozmiaru kodu rezydentnego, w paragrafach. Z konstrukcji naszego kodu w sposób pokazany powyżej, określenie rozmiar programu rezydentnego jest całkiem łatwe, używając następujących instrukcji kończących część nierezydentny naszego kodu (w cseg):

```
                mov     ax, ResidentSeg         ;potrzebny dostęp do ResidentSeg
                mov     es, ax
                mov     ah, 62h                 ;wywołanie przez DOS PSP
                int     21h
                mov     es:PSP, bx              ;zachowuje wartosć PSP w zmiennej PSP

; Następujący kod oblicza rozmiar części rezydentnej kodu. Segment EndResident jest pierwszym segmentem
; w pamięci po kodzie rezydentnym. Wartość PSP programu jest adresem segmentu na początku bloku
;rezydentnego, Przez obliczenie EndResident – PSP obliczymy rozmiar części rezydentnej w paragrafach

                mov     dx, Endresident         ;pobranie adresu segmentu EndResident
                sub     dx, bx                  ;odjęcie PSP

;Okay, wykonujemy wywołanie TSR, zabezpieczamy tylko kod rezydentny

                mov     ax, 3100h               ;AH=31h (TSR), AL=0 (kod powrotu)
                int     21h
```

Wykonywanie powyższego kodu zwraca sterowanie do MS-DOS, zachowując nasz kod rezydentny w pamięci.

Jest jeden szczegół końcowego zarządzania pamięcią do rozważenia przed przejściem do innych tematów związanych z programami rezydentnymi – dostęp do danych wewnątrz programu rezydentnego. Procedury wewnątrz programu rezydentnego stają się aktywne w odpowiedzi na bezpośrednie wywołanie z jakiegoś innego programu lub przerwania sprzętowego (zobacz następną sekcję) Na wejściu, podprogram rezydentny może wyszczególnić, że pewne rejestry zawierają różne parametry, ale jednej rzeczy jakiej nie możemy oczekiwać od kodu wywołującego to właściwego ustawienia rejestrów segmentowych dla nas. Faktycznie, jedyny rejestr segmentowy jaki będzie zawierał znaczącą wartość (dla kodu rezydentnego) jest kod rejestru segmentowego. Ponieważ wiele funkcji rezydentnych będzie chciało uzyskać dostęp do danych lokalnych, to znaczy, że te funkcje mogą musieć ustawić ds. lub jakiś inny rejestr(y) na wejściu początkowym. Na przykład przypuśćmy, że mamy funkcję, licznik, który po prostu zlicza liczbę razy jaką jakiś inny kod wywołał ją ponieważ jest rezydentny. Jedyną rzeczą jaką zawierałoby ciało tej funkcji to pojedyncza instrukcja inc counter. Niestety taka instrukcja zwiększałaby zmienną przy offsecie counter w aktualnym segmencie danych(to jest segmencie wskazywanym przez rejestr ds.). Jest mało prawdopodobne, że ds. wskazywałby segment danych związany z procedurą zliczania. Dlatego też będziemy zwiększać jakieś słowo w innym segmencie (prawdopodobnie w segmencie danych kodu wywołującego). Może to dać katastrofalny wynik.

Są dwa rozwiązania tego problemu. Pierwszym jest włożenie wszystkich zmiennych w segmencie kodu (bardzo popularna praktyka w sekcji kodu rezydentnego) i użycie przedrostka przesłonięcia segmentu cs: we wszystkich zmiennych. Na przykład, do zwiększenia zmiennej counter możemy użyć instrukcji inc cs:counter. Ta technika działa dobrze jeśli jest tylko kilka odniesień do zmiennych w naszej procedurze. Jednak cierpi ona na kilka poważnych wad. Po pierwsze przedrostek przesłonięcia segmentu czyni nasze instrukcje większymi i wolniejszymi; jest to poważny problem jeśli uzyskujemy dostęp do wielu różnych zmiennych w całym kodzie rezydentnym. Po drugie, łatwo jest zapomnieć umieścić przedrostek przesłonięcia segmentu przed zmienną, w skutek czego powodujemy, że funkcja TSR zniszczy pamięć w segmencie danych kodu wywołującego. Innym rozwiązaniem problemu segmentu jest zmiana wartości w rejestrze ds. na wejściu do procedury rezydentnej i przywrócić ją na wyjściu. Następujący kod demonstruje jak to zrobić:

```
push    ds.             ;zachowanie oryginalnej wartości DS.
push    cs              ;skopiowanie wartości CS do DS.
pop     ds.
inc     counter         ;zwiększenie wartości zmiennej
pop     ds.             ;przywrócenie oryginalnej wartości DS.
```

Oczywiście, używając przedrostka przesłonięcia segmentu cs: jest tu dużo bardziej sensownym rozwiązaniem . Jednakże kod będzie rozleglejszy a dostęp do wielu zmiennych lokalnych, ładowanie ds. z cs (zakładając, że wkładamy nasze zmienne do segmentu rezydentnego) byłby bardziej wydajne.

18.2 TSR'Y AKTYWNE KONTRA BIERNE

Microsoft identyfikuje dwa typy podprogramów TSR: aktywne i bierne. Bierny TSR jest TSR'em, który aktywuje się w odpowiedzi na wyraźne wywołanie z wykonywanego programu. Aktywny TSR jest TSR'em który odpowiada na przerwanie sprzętowe lub wywołanie przerwania sprzętowego.

TSR'y są prawie zawsze podprogramami obsługi przerwań (zobacz „Struktura przerwań 80x86 i Podprogramy obsługi przerwań (ISR)"). Aktywne TSR'y są typowymi podprogramami obsługi przerwań sprzętowych a TSR'y bierne są generalnie programami obsługującymi przerwania kontrolowane (zobacz „Przerwania kontrolowane"). Chociaż, w teorii, jest możliwe dla TSR, określenie adresu podprogramu w biernym TSR'ze i bezpośrednie wywołanie tego podprogramu, mechanizm przerwań kontrolowanych 80x86 jest doskonałym urządzeniem dla wywoływania takich podprogramów, więc wiele TSR'ów używa go.

TSR'y bierne generalnie dostarczają wywoływalnej biblioteki podprogramu lub rozszerzenia jakiegoś wywołania DOS lub BIOS. Na przykład, możemy chcieć ponownie wyznaczyć trasę wszystkich znaków wysyłanych do drukarki z pliku. Przez aktualizację wektora 17h (zobacz „Porty równoległe PC") możemy przechwycić wszystkie znaki przeznaczone dla drukarki. Lub możemy dodać dodatkowy sposób działania do podprogramu BIOS przez zmianę w jego wektorze przerwań. Na przykład możemy dodać nową funkcję wywołującą do podprogramu obsługi video BIOS 10h (zobacz „MS-DOS, PC_BIOS i I/O plików") poprzez wyszukanie specjalnych wartości w ah i przekazanie wszystkich innych wywołań int 10h do oryginalnego programu obsługi. Innym zastosowaniem biernego TSR'a jest dostarczenie nowego rodzaju zbioru usług w całym nowym wektorze przerwań, których nie może już dostarczyć BIOS. Obsługa myszki dostarczana przez sterownik mouse.com jest dobrym przykładem takiego TSR'a.

Aktywne TSR'y generalnie służą jednej z dwóch funkcji. Albo obsługują bezpośrednio przerwania sprzętowe albo nakładają się na przerwania sprzętowe żeby można było aktywować je w podstawowym okresie bez jawnego wywołania z aplikacji. Programy pop-up są dobrym przykładem aktywnego TSR'a. Program pop-up sam podłącza się pod przerwanie klawiatury PC (int 9). Naciskając klawisz aktywujemy taki program. Pogram może czytać z portu klawiatury PC (zobacz „Klawiatura PC") aby zobaczyć czy użytkownik wcisnął specjalną sekwencję klawiszy. Jeśli pojawiłaby się taka sekwencja klawiszy, aplikacja może zachować część pamięci ekranu i „pop-up" na ekran, wykonując jakieś funkcje żądane przez użytkownika a potem przywrócić ekran kiedy zostanie to zrobione. Program Sidekick™ firmy Borland jest przykładem niezmiernie popularnego programu TSR , chociaż istnieje wiele innych.

Nie wszystkie aktywne TSR'y są jednak pop-up. Dobrym przykładem aktywnego TSR'a są pewne wirusy. Aktualizują one różne wektory przerwań, które aktywują je automatycznie aby mogły wykonać swoje nikczemne dzieło. Na szczęście, niektóre programy anty wirusowe są również dobrymi przykładami aktywnego TSR'a, aktualizują te same wektory przerwań i wykrywają aktywne wirusy próbując ograniczyć szkody jakie może wyrządzić wirus

Zauważmy, że TSR może zawierać oba składniki, aktywny i bierny,. To znaczy, mogą być podprogramy, które wywołują przerwania sprzętowe i inne, które aplikacja wywołuje jawnie. Jednakże jeśli podprogram jest aktywny w programie rezydentnym, będziemy twierdzić , że cały TSR jest aktywny.

Poniższy program jest krótkim przykładem TSR'a, który dostarcza podprogramów aktywnych i biernych. Program ten aktualizuje wektory przerwań int 9 (przerwanie klawiatury) i int 16 (przerwania kontrolowane klawiatury) . Co jakiś czas system generuje przerwanie klawiaturowe , aktywuje podprogram (int 9) zwiększając licznik. Ponieważ klawiatura zazwyczaj generuje dwa przerwania klawiaturowe przy naciśnięciu klawisza, dzieląc tą wartość przez dwa tworzymy przybliżoną liczbę klawiszy wciśniętych podczas startu TSR'a. Podprogram bierny, związany z wektorem int 16h ,zwraca liczbę naciśnięć klawisza do programu wywołującego. Poniższy kod dostarcza dwóch programów, TSR'a i krótkiej aplikacji wyświetlającej liczbę naciśnięć klawisza od startu TSR'a.

```
; To jest przykład aktywnego TSR'a, który zlicza aktywowane przerwania klawiaturowe
;Zdefiniowany segment rezydentny musi przyjść przed wszystkim innym
```

```
ResidentSeg     segment         para  public  'Resident'
ResidentSeg     ends

EndResidentSeg  segment         para  public 'EndRes'
EndResidentSeg  ends

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

; Segment rezydentny, który przechowuje kod TSR'a:

ResidentSeg     segment         para   public 'Resident'
                assume          cs:ResidentSeg, ds:nothing

; następująca zmienna zlicza liczbę przerwań klawiaturowych

KeyIntCnt       word    0

;Te  dwie zmienne zawierają oryginalne wartości wektora przerwań INT 9 i  INT 16

OldInt9         dword   ?
OldInt16        dword   ?

;MyInt9 -       System wywołuje ten podprogram zawsze kiedy wystąpi przerwanie klawiaturowe.
;               Podprogram ten  zwiększa zmienną KeyIntCnt a potem przekazuje sterowanie
;               do oryginalnego programu obsługi Int 9

MyInt9          proc    far
                inc     ResidentSeg:KeyIntCnt
                jmp     ResidentSeg:OldInt9
MyInt9          endp

;MzInt16 +      Jest to bierny składnik tego TSR'a. Aplikacja jawnie wywołuje ten podprogram
;               instrukcją INT 16h. Jeśli AH zawiera 0FFh, podprogram ten zwraca liczbę
;               przerwań klawiaturowych w rejestrze   AX. Jeśli AH zawiera jakąś inną wartość
;               podprogram ten przekazuje sterownie do oryginalnego podprogramu obsługi INT 16
;               (przerwania kontrolowane klawiatury)

MyInt16         proc    far
                cmp     ah, 0FFh
                je      ReturnCnt
                jmp     ResidentSeg : OldInt16                  ;wywołanie  oryginalnego podprogramu

; Jeśli AH =  0FFh, zwraca liczbę przerwań klawiaturowych

ReturnCnt:      mov     ax, ResidentSeg : KeyIntCnt
                iret
MyInt16         endp

ResidentSeg     ends

cseg            segment         para  public  'code'
                assume          cs:cseg, ds:ResidentSeg

Main            proc
                Meminit

                Mov     ax, ResidentSeg
```

```
                mov     ds, ax
                mov     ax, 0
                mov     es, ax

                print
                byte    "Keybord  inetrrupt counter TSR program", cr, lf
                byte    "Installing.....", cr, lf,0
```

;Aktualizacja wektorów przerwań INT 9 i INT 16. Zauważmy ,że powyższe instrukcje uczynią ResidentSeg
; aktualnym segmentem danych, więc możemy przechować stare wartości INT 9 i INT 16 bezpośrednio w
; zmiennych OldInt9 i OldInt16

```
                cli                             ;wyłączenie przerwań!
                mov     ax, es:[9*4]
                mov     word  ptr  OldInt9, ax
                mov     ax, es:[9*4+2]
                mov     word  ptr OldInt9+2, ax
                mov     es:[9*4], offset MyInt9
                mov     es:[9*4+2], seg ResidentSeg

                mov     ax, es:[16h*4]
                mov     word  ptr OldInt16, ax
                mov     ax, es:[16h*4+2]
                mov     word  ptr OldInt16+2, ax
                mov     es:[16h*4], offset MyInt16
                mov     es:[16h*4+2], seg ResidentSeg
                sti                             ;OK, włączamy ponownie przerwania
```

;Połączyliśmy, teraz jedyną rzeczą do zrobienia jest zakończenie i pozostanie rezydentnym

```
                print
                byte    „Installed.", cr, lf,0

                mov     ah, 62h                 ;pobranie wartości PSP tego programu
                int     21h

                mov     dx, EndResident
                ;obliczenie rozmiaru programu
                sub     dx, bx
                mov     ax, 3100h               ;polecenie TSR DOS'a
                int     21h
Main            endp
cseg            ends

sseg            segment         para stack 'stack'
stk             db      1024  dup ("stack")
sseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       db      16  dup  (?)
Zzzzzzseg       ends
                end     Main
```

Tu jest aplikacja, która wywołuje MyInt16 drukującą liczbę naciśnięć klawisza:

;jest to program towarzyszący TSR'owi keycnt. Program wywołuje podprogram „MyInt16" w TSR'ze
; określając liczbę przerwań klawiaturowych. Wyświetla przybliżoną liczbę uderzeń w klawisze
; (przerwania klawiaturowe / 2) i wychodzi)

```
                .xlist
```

```
                include         stdlib.a
                includelib      stdlib.lib
                .list

cseg            segemnt         para  public ‘code’
                assume          cs:cseg, ds:nothing

Main            proc
                meminit
                print
                byte            “Approximate number of keys pressed: “,0
                mov             ah, 0FFh
                int     16h
                shr     ax, 1                           ;musi być dzielone przez dwa
                putu
                putcr
                ExitPgm
Main            endp
cseg            ends

sseg            segment         para  stack  ‘stack’
stk             db      1024  dup (“stack”)
sseg            ends

zzzzzzseg       segment         para  public  ‘zzzzzz’
LastBaytes      db      16   dup  (?)
Zzzzzzseg       ends
                end     Main
```

18.3 WSPÓŁBIEŻNOŚĆ

Jednym dużym problemem z aktywnymi TSR’ami jest to, że ich wywołanie jest asynchroniczne. Mogą się aktywować przy dotknięciu klawiatury, przerwaniu zegarowym lub przez przybyły znak do portu szeregowego. Ponieważ aktywują się one przy przerwaniach sprzętowych, PC może akurat wykonywać jakiś kod kiedy nadciągnie przerwanie. O nie jest problem, chyba ,że TSR sam zdecyduje wywołać jakiś obcy kod, na przykład podprogram DOS’a lub BIOS’a lub jakiś inny TSR. Na przykład głównym programem może być wywołanie DOS’a kiedy przerwanie zegarowe Aktywuje TSR, przerwanie wywołujące DOS , podczas gdy CPU jeszcze wykonuje kod wewnątrz DOS’a. Jeśli TSR próbuje wywołać DOS w tym miejscu, nastąpi powrót do DOS’a. Oczywiście, DOS nie jest współbieżny, wić stworzy to wiele różnych problemów (zazwyczaj zawieszenie systemu). Kiedy piszemy aktywnego TSR’a , który wywołuje inne podprogramy poza tymi dostarczonymi bezpośrednio w TSR’ze, musimy zwrócić uwagę na możliwy problem współbieżności .

Zauważmy, że bierne TSR’y nigdy nie cierpiały z tego powodu. Faktycznie, każdy podprogram TSR wywoływany biernie, będzie wykonywane w środowisku kodu wywołującego chyba, ze jakiś inny sprzętowy ISR lub aktywny TSR wykona wywołanie naszego podprogramu, wtedy nie musimy martwić się o współbieżność biernego podprogramu. Jednakże współbieżność jest kwestią aktywnego TSR’a i podprogramu biernego, który wywołuje aktywny TSR.

18.3.1 PROBLM WSPÓŁBIEŻNOŚCI Z DOS

DOS jest prawdopodobnie najdotkliwszym punktem projektantów TSR.DOS nie jest współbieżny, zawiera wiele usług jakich może użyć TSR. Uświadomiwszy to sobie , Microsoft dodał pewne wsparcie dla DOS pozwalające TSR’om sprawdzać czy DOS jest aktualnie aktywny. W końcu współbieżność jest problemem tylko jeśli wywołujemy DOS podczas gdy jest już aktywny. Jeśli nie jest aktywny, możemy wywołać z TSR’a bez żadnych ubocznych skutków.

MS-DOS dostarcza specjalnej jednobajtowej flagi (InDOS), która zawiera zero jeśli DOS jest aktualnie aktywny i wartość niezerową jeśli DOS przetwarza żądanie aplikacji. Poprzez testowanie flagi InDOS nasz TSR może określić czy może bezpiecznie dokonać wywołania DOS’a. Jeśli ta flaga to zero, możemy zawsze wywołać DOS. Jeśli flaga ta zawiera jeden nie będziemy zdolni do wywołania DOS’a .MS-DOS dostarcza funkcji wywołującej Get InDOS Flag Addess, która zwraca adres flagi InDOS. Używając tej funkcji ładujemy ah

wartością 34h i wywołujemy DOS. DOS zwróci adres flagi InDOS w es:bx. Jeśli zachowamy ten adres, nasz program rezydentny będzie mógł przetestować lagę InDOS aby sprawdzić czy DOS jest aktywny.

W rzeczywistości są dwie flagi, które powinniśmy przetestować, flaga InDOS i flaga błędu krytycznego (critter). Obie te flagi powinny zawierać zero przed wywołaniem DOS'a z TSR'a. W DOS'ie w wersji 3.1 i późniejszych, flaga błędu krytycznego pojawia się w bajcie tuż przed flagą InDOS.

Więc co powinniśmy zrobić jeśli te dwie flagi nie są zerami? Dosyć łatwo jest powiedzieć „hej, wrócimy do tego później , kiedy MS-DOS wróci do programu użytkownika" Ale jak to zrobić? Na przykład ‚jeśli przerwanie klawiatury aktywuje nasz TSR i przekazujemy sterowanie do rzeczywistego podprogramu obsługi klawiatury ponieważ DOS jest zajęty, nie możemy oczekiwać, że nasz TSR magicznie się zrestartuje później, kiedy DOS nie będzie dłużej aktywny.

Sztuczką jest aktualizacja naszego TSR w przerwaniu zegarowym tak jak i przerwaniu klawiaturowym. Kiedy przerwanie naciśnięcia klawisza wzbudzi nasz TSR i odkryjemy ‚że DOS jest zajęty, ISR klawiatury może po prostu ustawić flagę, która powie mu ‚żeby spróbował później; wtedy przekazuje sterowanie do oryginalnego programu obsługi klawiatury. Tymczasem ISR zegara, jaki napisaliśmy ‚stale sprawdza tą flagę jaką stworzyliśmy. Jeśli flaga jest wyzerowana, po prostu przekazujemy sterowanie do oryginalnego podprogramu obsługi przerwania zegarowego, jeśli flaga jest ustawiona, wtedy kod sprawdza flagi InDOS i CritErr. Jeśli mówią, że DOS jest zajęty, ISR zegarowy przekazuje sterowanie do oryginalnego programu obsługi . Wkrótce po zakończeniu DOS'a kiedykolwiek się to zdarzy, przerwanie zegarowe nadejdzie i wykryje, że DOS nie jest dłużej aktywny. Teraz nasz ISR może przejąć i zrobić konieczne wywołania DOS'a. Oczywiście, ponieważ nasz kod zegarowy określa, że DOS nie jest zajęty, powinien wyzerować flagę „ Ja chcę obsłużyć" aby przyszłe przerwanie zegarowe nie nieumyślnie zrestartowały TSR .

Jest tylko jeden problem z tym podejściem. Są pewne wywołania DOS , które mogą pobierać nieskończoną ilość czasu do wykonania. Na przykład, jeśli wywołujemy DOS do odczytu klawisza z klawiatury (lub wywołujemy podprogram getc z Biblioteki Standardowej, który wywołuje DOS do odczytu klawisza), mogą być godziny, dni lub nawet dłużej zanim ktoś naciśnie klawisz. Wewnątrz DOS'a jest pętla, która czeka dopóki użytkownik rzeczywiście ni naciśnie klawisza. I dopóki użytkownik naciska jakiś klawisz, flaga InDOS będzie pozostawała nie zerowa. Jeśli napisaliśmy TSR oparty o zegar, buforujący dane co kilka sekund i potrzebujemy zapisać wynik na dysku, przepełnimy nasz bufor nowymi danymi jeśli czekamy na użytkownika, który właśnie poszedł na obiad, naciskając klawisz w DOS'owym programie command.com.

Szczęśliwie, MS-DOS dostarcza rozwiązanie tego problemu – przerwania jałowe. Kiedy MS-DOS jest w nieskończonej pętli oczekującej na urządzenie I/O, nieustannie wykonuje instrukcję int 28h. Przez aktualizację wektora int 28h, nasz TSR może określić kiedy DOS znajduje się w takiej pętli. Kiedy DOS wykonuje instrukcję int 28h, bezpiecznie jest wywołać DOS, którego numer funkcji (wartość w ah) jest większa niż 0Ch.

Więc jeśli DOS jest zajęty kiedy nasz TSR chce wykonać wywołanie DOS'a, musi użyć albo przerwania zegarowego albo przerwania jałowego (int 28h) do aktywacji części naszego TSR'a, który musi wykonać wywołanie DOS. Jedna rzecz jaką musimy zapamiętać na końcu to to, że kiedykolwiek testujemy lub modyfikujemy każdą z powyższych flag, jesteśmy w sekcji krytycznej. Upewnijmy się, że przerwania są wyłączone. Jeśli nie , nasz TSR wykona aktywację dwóch kopii samego siebie lub może skończyć wprowadzania DOS'a w tym samym czasie kiedy inny TSR wprowadza DOS.

Przykład TSR'a używającego tej techniki pojawi się trochę później, ale jest kilka dodatkowych problemów współbieżności jakie musimy omówić najpierw.

18.3.2 PROBLEM WSPÓŁBIEŻNOŚCI Z BIOS

DOS nie jest jedynym niewspółbieżnym kodem jaki może chcieć wywołać TSR. Podprogramy BIOS PC również podlegają pod tą kategorię. Niestety , BIOS nie dostarcza flagi „InBIOS" lub zwielokrotnionego przerwania. Sami musimy dostarczyć takiej funkcjonalności.

Kluczem do zapobieżenia współbieżności podprogramu BIOS'a jaki chcemy wywołać jest zastosowanie „otoczki". Otoczka jest to krótkim ISR który aktualizuje istniejące przerwanie BIOS specjalnie do manipulowania flagą InUSe. Na przykład przypuśćmy, że musimy wywołać int 10h (usługa video) z wewnątrz naszego TSR. Możemy użyć następującego kodu do dostarczenia flagi „Int10InUse" którą nasz TSR może przetestować:

```
MyInt10     proc    far
            inc     cs:Int10InUse
            pushf
            call    cs: OldInt10
            dec     cs: Int10InUse
            iret
MyInt10     endp
```

Zakładając, że zainicjalizowaliśmy zmienną Int10InUse zerem, flaga w użyciu będzie zawierał zero, kiedy bezpiecznie jest wykonana instrukcja int 10h w naszym TSR'ze, będzie zawierała wartość niezerową kiedy program obsługi przerwania int 10h jest zajęty. Możemy użyć tej flagi jak flagi InDOS do wstrzymania wykonywania naszego kodu TSR.
Podobnie Jak w DOS'ie jest kilka podprogramów BIOS, które mogą pobierać nieokreśloną ilość czasu do zakończenia. Odczytując klawisz z bufora klawiatury, odczytujemy lub zapisujemy znaki do portu szeregowego, lub drukujemy znaki na drukarce. W pewnych przypadkach możliwe jest stworzenie otoczki, która pozwoli naszemu TSR'owi aktywować się, podczas gdy podprogram BIOS wykonuje jedną z tych pętli odpytujących, co nie jest prawdopodobnie żadną korzyścią. Na przykład, jeśli program czeka aż drukarka odbierze znak zanim wyśle inny do drukowania, nasz TSR musi zapobiec temu i próbować wysłać znak do drukarki, która tego nie osiąga (innymi słowy, składa dane wysyłane do drukarki). Dlatego też otoczki BIOS generalnie nie martwią się o nieokreślone odroczenie w podprogramie BIOS

5,8,9,D,E,10, 13, 16, 17,21,28

Jeśli ten problem wpadnie naszemu TSR'owi i pewnej aplikacji ,możemy chcieć umieścić otoczki wokół następujących przerwań aby zobaczyć czy to rozwiąże nasz problem: int 5, int 8, int 9, int B, int C, int D, int E, int 10, int 13, int 14, int 16 lub int 17. To są popularni sprawcy problemów, kiedy konstruujemy TSR.

18.3.3 PROBLEM WSPÓŁBIEŻNOŚCI Z INNYM KODEM

Problem współbieżności w innym kodzie może być również wywołany. Na przykład rozważmy Bibliotekę Standardową UCR. Biblioteka Standardowa UCR nie jest współbieżna. Nie jest to zazwyczaj duży problem z dwóch powodów. Po pierwsze TSR'y nie wywołują podprogramów Biblioteki Standardowej. Zamiast tego dostarczają wyników, których może użyć normalna aplikacja; te aplikacje używające Biblioteki Standardowej manipulują takimi wynikami. Drugim powodem jest to, że kiedy zawrzemy jakiś podprogram Biblioteki Standardowej w TSR'ze, aplikacja miałaby oddzielną kopie tego podprogramu bibliotecznego. TSR może wykonać instrukcję strcmp kiedy aplikacja jest w środku podprogramu strcmp, ale to nie jest ten sam podprogram! TSR nie jest współbieżny z kodem aplikacji, wykonuje oddzielny podprogram.

Jednakże wiele z funkcji Biblioteki Standardowej wywołują DOS lub BIOS. Wywołania takie nie sprawdzają czy DOS lub BIOS są już aktywne. Dlatego też, wywołanie wielu podprogramów Biblioteki Standardowej z wnętrza TSR może spowodować współbieżność DOS'a lub BIOS'a .

Istnieje jedna sytuacja, kiedy TSR może powrócić do programu Biblioteki Standardowej. Przypuśćmy, że nasz TSR ma oba składniki, bierny i aktywny. Jeśli aplikacja główna dokonuje wywołania podprogramu pasywnego w TSR'ze a ten program wywołuje program Biblioteki Standardowej, istnieje możliwość, że system przerwań może przerwać podprogram Biblioteki Standardowej a aktywna część TSR'a może ponownie wrócić do tego samego kodu. Chociaż taka sytuacja są raczej rzadkie, powinniśmy mieć na uwadze taką możliwość.

Oczywiście najlepszym rozwiązaniem jest unikanie Biblioteki Standardowej wewnątrz TSR'ów . Z innych powodów, podprogramy Biblioteki Standardowej są trochę duże, a TSR'y powinny być tak małe jak to możliwe.

18.4 PRZWRANIE RÓNOCZESNYCH PROCESÓW (INT 2FH)

Kiedy instalujemy bierny TSR lub aktywny TSR z biernym składnikiem, będziemy musieli wybrać jakiś wektor przerwań do aktualizacji, aby inne programy mogły komunikować się z naszym biernym podprogramem. Możemy wybrać wektor przerwań prawie losowo, powiedzmy int 84, ale spowoduje to problem kompatybilności. Co się zdarzy jeśli ktoś inny używa tego wektora przerwań? Czasami wybór wektora przerwań jest jasny. Na przykład, jeśli nasz bierny TSR rozszerza usługę klawiaturową int 16h, ma sens aktualizacja wektora int 16h i dodanie dodatkowo funkcji przed i po tych dostarczonych już przez BIOS. Z drugiej strony, jeśli tworzymy sterownik dla nowego urządzenia dla PC, prawdopodobnie nie będziemy chcieli nakładać funkcji wspierającej dla tego urządzenia na jakieś inne przerwania. Mimo to przypadkowe wykorzystanie nieużywanego wektora przerwań jest ryzykowne; jak wiele innych programów zdecydowało się na zrobienie tego samego? NA szczęście MS-DOS dostarcza rozwiązania: przerwanie równoczesnych procesów. Int 2Fh dostarcza ogólnego mechanizmu dla instalacji, testowania obecności i komunikowania z TSR'em.

Używając przerwania równoczesnych procesów, aplikacja umieszcza wartość identyfikacyjną w ah i numer funkcji w al. a potem wykonuje instrukcję int 2Fh. Każdy TSR w łańcuchu int 2Fh porównuje wartość w ah ze swoją własną unikalną wartością identyfikatora. Jeśli wartości pasują, TSR przetwarza polecenia określone przez wartość w rejestrze al. Jeśli identyfikowane wartości nie pasują, TSR przekazuje sterowanie do następnego w łańcuchu programu obsługi int 2Fh.

Oczywiście, to zredukuje problem tylko w pewnym stopniu, ale nie wyeliminuje go. Pewnie, nie musimy odgadywać losowego numeru wektora przerwań, ale musimy jeszcze wybrać losowy numer identyfikacyjny. Przecież wydaje się rozsądne, że musimy wybrać ten numer przed zaprojektowaniem TSR'a i jakieś aplikacji, która go wywołuje, w końcu jak aplikacja wiedziałaby jaką wartość załadować do ah jeśli dynamicznie przydzielalibyśmy tą wartość kiedy TSR byłby rezydentny?

Cóż, jest parę sztuczek jakie możemy wykorzystać do dynamicznego przydzielania identyfikatora TSR i pozwalają zainteresowanej aplikacji określić ID TSR'a. Poprzez konwencję, funkcja zero jest wywołaniem „Czy tu jesteś?" Aplikacja powinna zawsze wykonywać tą funkcję do określenia czy TSR jest obecny w pamięci przed wykonaniem jakiejś żądanej usługi. Zazwyczaj , funkcja zero zwraca zero w al. jeśli TSR nie jest obecny, zwraca 0FFh jeśli jest obecny. Jednakże , kiedy ta funkcja zwraca 0FFh, jedynie mówi nam, że jakiś TSR odpowiedział na zapytanie; nie gwarantuje, że to TSR, który nas interesuje jest obecny w pamięci. Jednak przez rozszerzenie nieco konwencji, jest bardzo łatwo zweryfikować obecność żądanego TSR. Przypuśćmy, że funkcja zero wywołuje również zwrot wskaźnika do unikalnej identyfikacji ciągu w rejestrze es:di. Wtedy kod testujący obecność określonego TSR'a, może testować ten ciąg kiedy int 2Fh wykryje obecność TSR'a. Poniższy fragment kodu demonstruje jak TSR może określić czy TSR identyfikowany jako „Randy's INT 10h Extension" jest obecny w pamięci; kod ten może również określać unikalną identyfikację kodu dal tego TSR'a dla późniejszych odniesień:

```
; Skanowanie wszystkich możliwych TSR'ów. Jeśli jest zainstalowany jeden, zobaczmy czy  jest to ten
; którym jesteśmy zainteresowani

                mov     cx, 0FFh                ;to będzie numer ID
IDLoop:         mov     ah, cl                  ; ID -> AH
                push    cx                      ;zachowanie cx przed wywołaniem
                mov     al., 0                  ;test obecności kodu funkcji
                int     2Fh                     ;wywołanie przerwania równoczesnych procesów
                pop     cx                      ;przywrócenie cx
                cmp     al., 0                  ;zainstalowano TSR?
                je      TryNext                 ;zwraca zero jeśli nie  ma żadnego
                strcmpl                         ;zobacz czy to ten jaki chcemy
                byte    „Randy's INT „
                byte    „10h Extension", 0
                je      Success                 ;skocz jeśli to nasz
TryNext:        loop    IDLoop                  ;w przeciwnym razie spróbuj ponownie
                jmp     NotInstalled            ;niepowodzenie jeśli doszliśmy do tego miejsca

Success:        mov     FuncID,  cl             ;zachowanie wyniku funkcji
                -
                -
                -
```

Jeśli ten kod zakończy się powodzeniem, zmienna FuncID zawiera wartość identyfikującą dal rezydentnego TSR'a. Jeśli niepowodzeniem, program prawdopodobnie będzie przerwany lub w przeciwnym razie zapewnić, że nigdy nie wywoła zaginionego TSR'a.

Powyższy kod pozwala aplikacji łatwe wykrycie obecności i określenia numeru ID dal określonego TSR'a. Kolejne pytanie: „Jak zdobędziemy numer ID dal TSR po raz pierwszy?" . Następna sekcja zajmie się tą kwestią, również tym jak TSR musi odpowiedzieć na przerwanie równoczesnych procesów.

18.5 INSTALOWANIE TSR'A

Chociaż omawialiśmy już jak uczynić program rezydentnym, jest kilka aspektów instalacji TSR, które musimy poznać. Po pierwsze co się stanie jeśli użytkownik zainstaluje TSR a potem spróbuje zainstalować go po raz drugi bez usunięcia tego , który już jest rezydentny? Po drugie jak możemy przydzielić numer identyfikacyjny TSR'owi, który nie wchodzi w konflikt z TSR'em który już jest zainstalowany? Ta sekcja zwróci się ku tym kwestiom.

Pierwszym problemem jest próba reinstalacji TSR'a. Chociaż można wyobrazić sobie typ TSR'a, który pozwala na wiele kopii samego siebie w pamięci w tym samym czasie, takich TSR'ów jest kilka i przejściowych. W większości przypadków , mając wiele kopii TSR'a w pamięci, w najlepszym razie marnujemy pamięć, a w najgorszym razie mamy krach systemu. Dlatego też, chyba, że napiszemy specyficzny TSR, który pozwala na wiele kopii samego siebie w pamięci w tym samym czasie, powinniśmy sprawdzać czy TSR jest już

zainstalowany przed ponowną jego instalacją. Kod ten jest identyczny z kodem aplikacji używany aby zobaczyć czy TSR jest zainstalowany, jedyną różnicą jest to, że TSR powinien wydrukować nieprzyjemną wiadomość i odmówić przejścia do TSR'a jeśli znajdzie jego kopię już zainstalowaną w pamięci. Robi to poniższy kod:

```
                mov     cx, 0FFh
SearchLoop:     mov     ah, cl
                push    cx
                mov     al, 0
                int     2Fh
                pop     cx
                cmp     al, 0
                je      TryNext
                strcmpl
                byte    "Randy's INT "
                byte    "10h Extension", 0
                je      AlreadyThere
TryNext:        loop    SearchLoop
                jmp     NotInstaled

AlreadyThere:   print
                byte    "A copy of this TSR already exist i n memory", cr,lf
                byte    "Aburting installation process.", cr, lf, 0
                ExitPgm
                -
                -
                -
```

W poprzedniej sekcji, pokazaliśmy jak napisać kod, który pozwoliłby aplikacji określić ID TSR'a określonego programu rezydentnego. Teraz musimy popatrzeć jak dynamicznie wybrać numer identyfikacyjny TSR'a, który nie wchodzi w konflikt z innym TSR'em. Jest to jeszcze inna modyfikacja pętli przeszukującej. Faktycznie, możemy zmodyfikować powyższy kod, tak aby wykonywał to dla nas. Wszystko co musimy zrobić to zachować jakąś wartość ID którą ma zainstalowany TSR. Musimy tylko dodać kilka linijek do powyższego kodu dla wykonania tego:

```
                mov     FuncID, 0               ;inicjalizacja FuncID zerem
                mov     cx, 0FFh
SearchLoop:     mov     ah, cl
                push    cx
                mov     al, 0
                int     2Fh
                pop     cx
                cmp     al, 0
                je      TryNext
                strcmpl
                byte    "Randy's INT "
                byte    "10h Extension", 0
                je      AlreadyThere

; Notka:  przypuszczalnie DS wskazuje na rezydentny segment danych, który zawiera zmienną FuncID.
          W przeciwnym razie musimy zmodyfikować następujący punkt jakiegoś rejestru segmentowego
           W segmencie zawierającym FuncID i użyć właściwego segmentu przesłaniającego FuncID

TryNext:        mov     FuncID, cl              ;zachowanie możliwego ID funkcji jeśli ten
                loop    SearchLoop              ; identyfikator nie jest używany
                jmp     NotInsatlled

AlreadyThere:   print
                byte    "A copy of this TSR already exist i n memory", cr,lf
                byte    "Aburting installation process.", cr, lf, 0
                ExitPgm
```

```
NotInstalled:   cmp     FuncID, 0                       ;jeśli ID ne są dostępne będzie zawierała zero
                jne     GoddID
                print
                byte    „There are too many TSRs alraedy installed.”, cr, lf
                byte    “Sorry, aborting instalation process.”, cr. Lf, 0
                ExitPgm
GoodID:
```

Jeśli ten kod dotrze do etykiety “GoodID” wtedy poprzednia kopia TSR nie jest obecna w pamięci a zmienna FuncID zawiera niewykorzystywany identyfikator funkcji.

Oczywiście kiedy instalujemy nasz TSR w ten sposób, musimy nie zapomnieć zaktualizować przerwanie 2Fh w łańcuchu int 2Fh. Również, musimy napisać program obsługi przerwania 2Fh przetwarzający wywołani int 2Fh. Poniżej mamy bardzo prosty programik obsługi przerwania równoczesnych procesów dla kodu jaki skonstruowaliśmy:

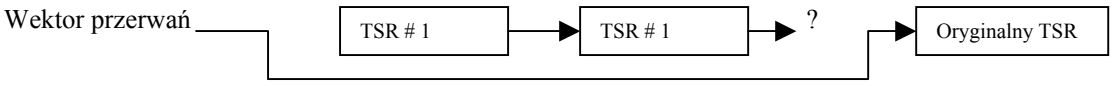
```
FuncID      byte    0                   ;powinien być w segmencie rezydentnym
OldInt2F    dword   ?

MyInt2F     proc    far
            cmp     ah, cs: FuncID      ;wywołanie dla nas?
            je      ItsUs
            jmp     cs: OldInt2F

;Teraz dekodujemy wartość funkcji w AL:

ItsUs:      cmp     al., 0              ;weryfikujemy obecność wywołania?
            jne     TryOtherFunc
            mov     al., 0FFh           ;zwracamy wartość “obecności” w AL
            lesi    IDString            ;zwracamy wskaźnik do ciągu w es:di
            iret                        ;powrót do kodu wywołującego
IDString    byte    „”Randy’s INT „
            byte    “10h Extension”, 0

; w dole, obsługa innych żądań zwielokrotnionych. Ten kod nie oferuje niczego, ale jest gdzie mam być
;testuje wartość w AL. określając która funkcja będzie się wykonywać

TryOtherFunc:
            -
            -
            -
            iret
MyInt2F     endp
```

18.6 USUWANIE TSR’A

Usunięcia TSR jest trochę trudniejsze niż jego instalacja. Są trzy rzeczy, które musi zrobić kod usuwający aby właściwie usunąć TSR’a z pamięci: po pierwsze musi zatrzymać jakieś oczekujące działanie (np. TSR może mieć ustawione jakieś flagi do rozpoczęcia działania w przyszłości) ; po drugie musi odtworzyć wszystkie wektory przerwań do ich poprzedniej postaci; po trzecie musi zwrócić całą zarezerwowaną pamięć z powrotem do DOS’a aby inne aplikacje mogły z niej korzystać. Podstawową trudnością z tymi trzema działaniami jest to, że nie zawsze jest możliwe poprawne zwrócenie wektorów przerwań.

Jeśli usuwany kod naszego TSR po prostu przywraca starą wartość wektora przerwań, możemy stworzyć rzeczywiście duży problem. Co się stanie jeśli użytkownik uruchomi jakieś inne TSR’y po uruchomieniu naszego i aktualizują one ten sam wektor przerwań jak nasz? To stworzyłoby łańcuch przerwań, który może wyglądać jak poniższy:

Jeśli odtwarzamy wektor przerwań oryginalną wartością, stworzymy co następuje:



To skutecznie blokuje TSR'y w łańcuchu w naszym kodzie. Gorzej jeszcze, to blokuje tylko te przerwania, które te TSR'y mają w użyciu wraz z naszym TSR'em, inne przerwania które te TSR'y aktualizują są jeszcze aktywne. Kto wie jak te przerwania będą się zachowywały w takich okolicznościach?

Jednym rozwiązaniem jest wydrukowanie komunikatu o błędzie informujący użytkownika, że nie mogą usunąć tego TSR'a dopóki nie usuną TSR'ów zainstalowanych uprzednio. Jest to powszechny problem z TSR'ami i większości użytkowników DOS'a, którzy instalują i usuwają TSR'y powinno być wygodniej ze świadomością ,że muszą usuwać TSR'y w odwrotnym porządku niż je instalowali.

Byłoby kuszącą sugestią nowa konwencja, że TSR'y powinny być posłuszne; być może jeśli numer funkcji to 0FFh, TSR powinien przechować wartość w es:bx w wektorze przerwań określonym w cl. To pozwoliłoby TSR'owi, który chciałby usunąć się przekazać adres swojego oryginalnego programu obsługi do poprzedniego TSR'a w łańcuchu. Są w związku z tym podejściem trzy problemy: po pierwsze prawie żaden TSR istniejący obecnie nie wspiera tej cechy; po drugie jakieś TSR'y mogą używać funkcji 0FFh do czegoś jeszcze, wywołując je z tą wartością, nawet jeśli znamy ich numer ID, możemy stworzyć problem; w końcu, ponieważ usunęliśmy TSR z łańcucha przerwań nie znaczy to ,że możemy (naprawdę) zwolnić pamięć używaną przez TSR. Schemat zarządzania pamięcią DOS (sprawa wolnego wskaźnika) działa podobnie jak stos. Jeśli są inne TSR'y zainstalowane powyżej naszego w pamięci, większość aplikacji nie będzie zdolna do użycia zwolnionej pamięci przez usunięcie naszego TSR'a.

Dlatego też, również zaadoptujemy strategię prostego informowania użytkownika, że nie może usunąć TSR jeśli są zainstalowane inne w dzielonym łańcuchu przerwań. Oczywiście to przywołuje dobre pytanie, jak możemy określić czy są inne TSR'y włączone w nasze przerwania? Cóż, nie jest to tak trudne. Wiemy, że wektory przerwań 80x86 powinny wskazywać jeszcze na nasze podprogramy, jeśli uruchomiliśmy ostatni TSR. Więc wszystko co musimy zrobić to porównać zaktualizowane wektory przerwań z adresami naszych podprogramów obsługi. Jeśli WSZYSTKIE pasują, wtedy możemy bezpiecznie usunąć TSR z pamięci. .tylko jeden z nich nie pasuje, wtedy nie możemy usunąć TSR'a z pamięci . Poniższa sekwencja kodu testuje aby zobaczyć czy jest OK. usunięcie TSR'a zawierającego ISR'y dla int 2Fh i int 9:

```
;OkayToRmv -   Podprogram ten zwraca ustawioną flagę przeniesienia jeśli usunięcie bieżącego TSR'a
;              z pamięci jest OK. Sprawdza wektory przerwań dla int 2fh i int 9 upewniając się ,że
;              wskazują one jeszcze nasze lokalne podprogramy. Kod ten zakłada, że DS. wskazuje segment
;              danych kodu rezydentnego

OkayToRmv     proc    near
              push    es
              mov     ax, 0                   ;ES wskazuje tablice wektora przerwań
              mov     es, ax
              mov     ax, word  ptr  OldInt2F
              cmp     ax, es: [2fh*4]
              jne     CantRemove
              mov     ax, word  ptr oldInt2F+2
              cmp     ax, es; [2Fh*4+2]
              jne     CantRemove

              mov     ax, word ptr  Oldint9
              cmp     ax, es:[9*4]
              jne     CantRemove
              mov     ax, word  ptr  OldInt+2
              cmp     ax, es: [9*4+2]
              jne     CantRemove

;Możemy bezpiecznie usuwać ten TSR z pamięci

              stc
              pop     es
```

```
                ret

; jeśli coś jest nie tak, nie możemy usunąć  tego TSR'a

CantRemove:     clc
                pop     es
                ret
OkayToRmv
```

Zanim TSR spróbuje usunąć sam siebie, powinien wywołać podprogram taki jak ten aby zobaczyć czy usuwanie jest możliwe.

Oczywiście, fakt ,że żaden inny TSR nie jest połączony do tych samych przerwań, nie gwarantuje ,że nie ma TSR'ów powyżej naszego w pamięci. Jednakże , usuwając TSR w tym przypadku nie spowoduje krachu systemu. Prawda, możemy nie być zdolni do odebrania pamięci jeśli TSR jest używany (przynajmniej dopóki suwamy inne TSR'y), ale przynamniej usuwanie nie tworzy komplikacji.

Usuwanie TSR'a z pamięci wymaga dwóch wywołań DOS, jednego dla zwolnienia pamięci używanej przez TSR i jednego do zwolnienia pamięci używanej przez obszar środowiska powiązanego z TSR'em. Robiąc to musimy zrobić dealokację wywołania DOS'a. To wywołanie wymaga przekazania adresu segmentu bloku udostępnionego w rejestrze es. Dla samego programu TSR , musimy przekazać adres PSP TSR'a . Jest to jeden z powodów dla którego musimy zachować jego PSP kiedy instalujemy go po raz pierwszy. Innym wywołaniem zwalniającym jaki musimy zrobić jest zwolnienie przestrzeni związanej z blokiem środowiska. Adres tego bloku jest pod offsetem 2Ch w PSP. Więc prawdopodobnie powinniśmy zwolnić go najpierw. Poniższy kod wykonuje zwalnianie pamięci powiązanej z TSR'em:

```
;Prawdopodobnie, zmienna PSP została zainicjalizowana adresem PSP tego programu przed wywołaniem
; TSR'a

                mov     es, PSP
                mov     es, es:[2Ch]            ;pobranie adresu bloku środowiska
                mov     ah, 49h                 ;wywołanie dealokacji bloku DOS
                int     21h

                mov     es, PSP                 ;teraz zwalniamy pamięć przestrzeni programu
                mov     ah, 49h
                int 21h
```

Niektóre słabo napisane TSR'y nie dostarczają żadnych udogodnień pozwalających nam usuwać je z pamięci. Jeśli ktoś chce usunąć taki TSR będzie musiał ponownie uruchomić PC. Oczywiście, jest to kiepskie projektowanie. Każdy TSR jaki zaprojektujemy do czegoś innego niż szybki test, powinien posiadać zdolność do usunięcia się z pamięci. Przerwanie równoczesnych procesów z numerem funkcji jeden jest często używane do tego celu. Usuwając TSR z pamięci, jakiś program przekazuje ID TSR'a i numer funkcji jeden do TSR'a. Jeśli TSR może usunąć się z pamięci, robi to i zwraca wartość oznaczającą powodzenie. Jeśli TSR nie może usunąć się z pamięci, zwraca jakąś część warunku błędu.

Generalnie, program usuwający jest samym TSR'em ze specjalnym parametrem, który mówi mu o usunięciu TSR'a bieżąco załadowanego do pamięci. Trochę później w tym rozdziale przedstawimy przykład TSR'a, który działa dokładnie w ten sposób .

18.7 INNE ZAGADNIENIA ZWIĄZANE Z DOS

Oprócz problemów współbieżności z DOS, jest kilka innych kwestii z jakim nasz TSR musi działać jeśli zamierza czynić wywołania DOS. Chociaż nasze wywołania nie muszą powodować współbieżności DOS'a, jest całkiem możliwe ,że wywołania DOS'a przez nasze TSR'y przeszkadzają strukturom danych używanym przez wykonywane aplikacje. Te struktury danych zawierają stos aplikacji, PSP, DTA i rozszerzony rekord informacji o błędzie DOS.

Kiedy aktywny lub bierny TSR przejmuje sterowanie z CPU, działa w środowisku głównej (pierwszoplanowej) aplikacji. Na przykład, TSR'y zwracają adres i jakąś wartość zachowaną na stosie, odłożoną na stos aplikacji. Jeśli TSR nie używa dużo przestrzeni stosu, jest świetnie , nie musimy przełączać stosów. Jednakże, jeśli TSR zżera znaczne ilości przestrzeni stosu z powodu wywołania rekurencyjnego lub alokacji

zmiennych lokalnych, TSR powinien zachować wartość ss i sp aplikacji i przełączyć na stos lokalny. Przed powrotem TSR powinien przełączyć się z powrotem na stos aplikacji pierwszoplanowej.

Podobnie, jeśli TSR wykonuje DOS'owe pobranie adresu psp, DOS zwraca adres PSP pierwszoplanowej aplikacji, a nie PSP TSR'a. PSP zawiera kilka ważnych adresów, których używa DOS przy zajściu błędu. Na przykład PSP zawiera adres programu zakończenia, program obsługi ctrl-braek, i obsługi błędu krytycznego. Jeśli nie przełączymy PSP z aplikacji pierwszoplanowej na TSR'a i wystąpi jeden z wyjątków (np. wystąpi ctrk-braek, lub błąd dyskowy) program obsługi związany z tą aplikacją może go przejąć. Dlatego też, kiedy robimy wywołania DOS, które w wyniku mogą dać jeden z tych warunków, musimy przełączać PSP. Podobnie, kiedy nasz TSR zwraca sterowanie do aplikacji pierwszoplanowej , musi przywrócić wartość PSP. MS-DOS dostarcza dwóch funkcji, które pobierają i ustawiają bieżący adres PSP. Funkcja DOS GetPSP (ah=51h) ustawia aktualny adres PSP programu do wartości z rejestru bx. Funkcja DOS GetPSP (ah = 50h) zwraca adres bieżącego PSP programu w rejestrze bx. Zakładając, że nierezydentna część naszego TSR'a zapisała jego PSP w zmiennej PSP, przełączamy pomiędzy PSP TSR'a a PSP aplikacji pierwszoplanowej jak pokazano:

```
;Zakładamy, że wprowadziliśmy kod TSR'a, określając, że jest OK. wywołując DOS, i przełączamy DS.
; żeby wskazywał nasze lokalne zmienne

        mov     ah, 51h                 ;pobranie adresu PSP aplikacji
        int     21h
        mov     AppPSP, bx              ;zachowanie lokalnie PSP aplikacji
        mov     bx, PSP                 ;zmiana systemowego PSP na PSP TSR'a
        mov     ah, 50h                 ;ustawienie funkcji Set PSP
        int     21h
        -
        -                               ;kod TSR'a
        -
        mov     bx, AppPSP              ;przywrócenie adresu systemowego PSP aby
        mov     ah, 50h                 ;wskazywał PSP aplikacji
        int     21h

    < porządkujemy i wracamy z TSR'a>
```

Inną globalną strukturą, której używa DOS jest adres transferu dysku .Ten bufor adresowy był używany dla I/O dysków w DOS wersji 1.0. Od tego czasu głównym zastosowaniem dla DTA były funkcje znajdowania pierwszego pliku i znajdowanie pliku kolejnego (zobacz „MS-DOS, PC-BIOS i I/O plików"). Oczywiście, jeśli aplikacja jest w środku stosowania danych w DTA a nasz TSR robi wywołanie DOS'a, które zmienia dane w DTA, będziemy wpływać na proces pierwszoplanowy. MS-DOS dostarcza dwóch funkcji, które pozwalają nam pobrać i ustawić adres DTA. Funkcja Get DTA Address, z ah -= 2Fh, zwraca adres DTA w rejestrach es:bx. Funkcja Set DTA Address (ah = 1Ah) ustawia DTA na wartość z pary rejestrów ds.:dx. Tymi dwoma funkcjami możemy zachowywać i przywracać DTA, co robiliśmy dla powyższego adresu PSP. DTA jest zazwyczaj pod offsetem 80h w PSP, następujący kod zachowuje DTA aplikacji pierwszoplanowej i ustawia aktualny DTA TSR'ów pod offsetem PSP:80

```
;Kod ten czyni takie same założenia jak przykład poprzedni

        mov     ah, 2Fh                 ;ustawienie DTA aplikacji
        int     21h
        mov     word  ptr  AppDTA, bx
        mov     word  ptr AppDTA+2, es

        push    ds
        mov     ds, PSP                 ;DTA jest w PSP
        mov     dx, 80h                 ;pod offsetem 80h
        mov     ah, 1ah                 ;ustawienie funkcji Set DTA
        int     21h
        pop     ds.
        -
        -                               ;kod TSR'a
```

```
        push    ds.
        mov     dx, word ptr AppDTA
        mov     ds, word ptr  AppDTA+2
        mov     ax, 1ah                 ;ustawienie funkcji Set DTA
        int     21h
```

Ostatnią kwestią związaną z TSR'em to to jak współpracuje z informacją o rozszerzonym błędzie w DOS. Jeśli TSR przerywa program bezpośrednio po tym jak DOS wraca do tego programu, może być jakaś informacja o błędzie, którą aplikacja pierwszoplanowa musi sprawdzić w rozszerzonej informacji o błędzie DOS'a. Jeśli TSR robi wywołanie DOS'a DOS może umieścić tą informację w statusie wywołania DOS'a przez TSR. Kiedy sterowanie jest zwracane do aplikacji pierwszoplanowej, może ona odczytać rozszerzony status błędu i pobrać tą informację generowaną przez wywołanie DOS'a przez TSR, nie wywołującą aplikację DOS. DOS dostarcza dwóch funkcji asymetrycznych, Get Exteneded Error i Set Extended Error, które ,odpowiednio, odczytują i zapisują te wartości. Funkcja Get Extended Error zwraca status błędu w rejestrach ax, bx, cx, dx, si ,di i ds. Musimy zachować rejestry w strukturze danych, która przybiera następującą postać:

```
ExtError        struct
eeAX            word    ?
eeBX            word    ?
eeCX            word    ?
eeDX            word    ?
eeSI            word    ?
eeDI            word    ?
eeDS            word    ?
eeES            word    ?
                word    3  dup  (0)             ;zarezerwowane
ExtError        ends
```

Funkcja Set Extended Error wymaga przekazania adresu do tej struktury w rejestrach ds:si (to dlatego te dwie funkcje są asymetryczne). Dla zachowania rozszerzonej informacji o błędzie możemy użyć kodu podobnego do tego:

```
; zachowujemy założenia takie jak  dla powyższych podprogramów. Również zakładamy, że struktura
;danych błędu nazywa się ERR i jest w tym samym segmencie co kod

        push    ds.                     ;zachowujemy wskaźnik do naszego DS.
        mov     ah, 59h                 ;ustawimy funkcję Get extended error
        mov     bx, 0                   ;wymagane przez tą funkcję
        int     21h

        mov     cs: ERR.eeDS, ds.
        pop     ds                      ;odzyskanie wskaźnika do naszego DS
        mov     ERR.eeAX, ax
        mov     ERR.eeBX, bx
        mov     ERR.eeCX, cx
        mov     ERR.eeDX, dx
        mov     ERR.eeSI, si
        mov     ERR.eeDI, di
        mov     ERR.eeES, es
        -
        -                               ;tu przychodzi kod TSR'a
        -
        mov     si, offset  ERR         ;ds już wskazuje poprawny segment
        mov     ax, 5D0Ah               ;5D0Ah jest kodem Set Extended Error
        int     21h
    <sprzątamy i wychodzimy>
```

Poniższy program rozszerza program licznika naciśnięć klawiszy przedstawiany trochę wcześniej w tym rozdziale. Ten program monitoruje naciśnięcia klawiszy i co minutę zapisuje dane do pliku listującego dane, czas i przybliżoną liczbę naciśnięć w ostatniej minucie. Ten pogram może nam pomóc odkryć ile czasu spędzamy pisząc w przeciwieństwie do rozmyślań przy monitorze

```
;Jest to przykład aktywnego TSR'a który zlicza przerwania klawiaturowe podczas aktywacji. Co minutę
; zapisuje liczbę przerwań klawiaturowych, które wystąpiły w poprzedniej minucie do pliku wyjściowego.
; Kontynuuje dopóki użytkownik nie usunie programu z pamięci.
;
; Użycie:
;       nazwa pliku KEYEVAL -       zaczyna zapisywać dane z naciśnięć klawiszy do tego pliku
;       KEYEVAL  REMOVE -           usuwa program rezydentny z pamięci
;
; Ten TSR sprawdza aby upewnić się ,że nie ma już aktywnej kopii w pamięci. Kiedy mamy przerwanie
; z dysku I/O, sprawdza aby upewnić się ,że DOS nie jest zajęty i zachowuje aplikacje globalne (PSP, DTA
; i rozszerzone info o błędzie). Kiedy usuwa się z pamięci , upewnia się, że nie ma innych łańcuchów
; przerwań w jakimś z jego przerwań zanim zacznie się usuwać.
;
; Definicja segmentu rezydentnego musi nadejść przed wszystkim innym

ResidentSeg     segment         para public  'Resident'
ResidentSeg     ends

EndResident     segment         para  public  'EndRes'
EndResident     ends

                .xlist
                .286
                .include        stdlib.a
                includelib      stdlib.lib
                .list

; Segment rezydentny przechowujący kod TSR'a:

ResidentSeg     segment         para  public 'Resident'
                assume          cs:ResidentSeg, ds:nothing

;Int 2Fh numer ID dla tego TSR'a:

MyTSRID         byte    0

;Następująca  zmienna zlicza liczbę przerwań klawiaturowych

KeyIntCnt       word    0

; Counter licznik zliczający liczbę milisekund jakie minęły, SecCounter  zlicza liczbę sekund (do 60)

Counter         word    0
SecCounter      word    0

;FileHandle jest uchwytem dla pliku logującego:

FileHandle      word    0

;NeedIO określa czy mamy w toku operacje I/O

NeedIO          word    0

;PSP  jest adresem psp dla tego programu
```

```
PSP             word    0

;Zmienne, które mówią nam czy DOS, INT 13h lub INT  16h są zajęte:

InInt13         byte    0
InInt16         byte    0
InDOSFlag       dword   ?

;Zmienne te zawierają oryginalne wartości wektorów przerwań jakie zaktualizowaliśmy

OldInt9         dword   ?
OldInt13        dword   ?
OldInt16        dword   ?
OldInt1C        dword   ?
OldInt28        dword   ?
OldInt2F        dword   ?

;struktura danych DOS:

ExtErr          struct
eeAX            word    ?
eeBX            word    ?
eeCX            word    ?
eeDX            word    ?
eeSI            word    ?
eeDI            word    ?
eeDS            word    ?
eeES            word    ?
                word    3  dup  (0)
ExtErr          ends

XErr            ExtErr  {}                      ;status rozszerzonego błędu
AppPSP          word    ?                       ;wartość PSP aplikacji
AppDTA          dword   ?                       ;adres DTA aplikacji

;Następujacec dane są w rekordzie wyjściowym. Po posortowaniu tych danych do tych zmiennych
; TSR zapisuje te dane na dysk

month           byte    0
day             byte    0
year            word    0
hour            byte    0
minute          byte    0
second          byte    0
Keystrokes      word    0
RecSize         =       $ - month


;MyInt9 -       System wywołuje ten podprogram za każdym razem kiedy wystąpi przerwanie klawiaturowe.
;               Zwiększa zmienną KeyIntCnt a potem przekazuje sterowanie do oryginalnego programu
;               obsługi przerwania Int 9
;
MyInt9          proc    far
                inc     ResidentSeg : KeyIntCnt
                jmp     ResidentSeg : OldInt9
MyInt9          endp


;Myint1C-       Przerwanie zegarowe. Zlicza 60 sekund a potem próbuje zapisać rekord pliku wyjściowego.
;               Oczywiście ta funkcja musi skakać po różnych problematycznych częściach kodu.
```

```
MyInt1C         proc    far
                assume  ds.ResidentSeg

                push    ds
                push    es
                pusha                           ;zachowujemy wszystkie rejestry
                mov     ax, ResidentSeg
                mov     ds., ax

                pushf
                call    OldInt1C

; Najważniejsze najpierw to ustawić nasz licznik przerwań żeby mógł liczyć co minutę. Ponieważ, mamy
; przerwania co 54.92549 milisekundy, musimy wykazać się większą precyzją niż 18 razy na sekundę,
; żeby synchronizacja nie odbiegła zbyt daleko

                add     Counter , 549           ;54,9 ms na int 1C
                cmp     Counter, 10000          ;1 sekunda
                jb      NotSecYet
                sub     Counter, 10000
                inc     SecCounter
NotSecYet:

;If NeedIO nie jest zerem, wtedy w toku jest operacja I/O. Nie narusza to wartości wyjściowej jeśli  jest
; taki przypadek

                cli                                     ;to jest rejon  krytyczny
                cmp     NeedIO, 0
                jne     SkipSetNIO

;Okay, żadnego I/O w toku, zobaczmy czy minuta minęła od ostatniego razu kiedy zapisywaliśmy naciśnięcie
; klawisza do pliku. Jeśli tak, czas zacząć inną operację I/O

                cmp     SecCounter, 60          ;przeszła już minuta?
                jb      Int1Cdone
                mov     NeedIO, 1               ;flaga potrzebna dla I/O
                mov     ax, KeyIntCnt           ;kopiuje to do  bufora wyjściowego
                shr     ax, 1                   ;po obliczeniu # naciśnięcia klawisza
                mov     KeyStrokes, ax
                mov     KeyIntcnt,0             ;reset do kolejnej minuty
                mov     SecCounter, 0

SkipSetNIO:     cmp     NeedIO, 1               ;czy I/O jest już w toku? Lub zrobiony?
                jne     Int1CDone

                call    ChkDOSStatus            ;zobacz czy DOS / BIOS są wolne
                jnc     Int1CDone               ;skocz jeśli zajęte

                call    DoIO                    ;zrób I/O jeśli DOS jest wolny

Int1CDone:      popa                            ;przywrócenie rejestrów i wyjście
                pop     es
                po      ds.
                iret
MyInt1C         endp
                assume  ds.:nothing

;MyInt28 -      przerwanie jałowe. Jeśli DOS jest w pętli oczekiwania na zakończenie I/O, wykonuje
;               instrukcję int 28h za każdym razem, kiedy przechodzi przez pętlę Możemy zignorować
;               flagi InDOS i CritErr w tym czasie i zrobić I/O jeśli inne przerwania są wolne
```

```
MyInt28         proc    far
                assume  ds.:ResidentSeg

                push    ds
                push    es
                pusha                           ;zachowanie wszystkich rejestrów

                mov     ax, ResidentSeg
                mov     ds., ax

                pushf                           ;wywołanie kolejnego INT 28h
                call    OldInt28                ; ISR w łańcuchu

                cmp     NeedIO, 1               ;czy mamy tymczasem I/O?
                jne     Int28Done

                mov     al, InInt13             ;zobacz czy BIOS jest zajęty
                or      al.,InInt16
                jne     Int28Done

                call    DoIO                    ;zróbmy I/O jeśli  BIOS jest wolny

Int28Done:      popa
                pop     es
                po      ds.
                iret
MyInt28         endp
                assume  ds.:nothing

;MyInt16-       jest to otoczka dla  programu obsługi INT 16h (przerwane kontrolowane klawiatury)

MyInt16         proc    far
                Inc     ResidentSeg : InInt16

; Wywołanie oryginalnego programu obsługi :

                pushf
                call    ResidentSeg: OldInt16

;Dla INT 16h musimy zwrócić flagi, które  pochodzą z poprzedniego wywołania

                pushf
                dec     ResidentSeg :InInt16
                popf
                retf    2                       ;lewy IRET do zachowania flag
MyInt16         endp

;MyInt13-       To jest tylko otoczka dla programu obsługi INT 13h (przerwanie kontrolowane I/O dysku)

MyInt13         proc    far
                inc     ResidentSeg: InInt13
                pushf
                call    ResidentSeg: OldInt13
                pushf
                dec     ResidentSeg: InInt13
                popf
                retf    2
MyInt13         endp

; ChkDOSStatus-         Zwraca wyzerowaną flagę przeniesienia jeśli  podprogramy DOS lub BIOS są zajęte
```

```
;                       i nie możemy ich przerwać

ChkDOSStatus  proc    near
              assume  ds.: ResidentSeg
              les     bx, InDOSFlag
              mov     al, es:[bx]                 ;pobranie flagi InDOS
              or      al, es:[bx-1]               ;OR z flagą CritErr
              or      al, InInt16                 ;OR z naszą  wartością otoczki
              or      al., InInt13
              je      Okay2Call
              clc
              ret

Okay2Call:    clc
              Ret
ChkDOSStatus  endp
              assume  ds:nothing

; PreserveDOS-  pobranie kopii  bieżącego PSP, DTA i rozszerzonej informacji DOS'a i zachowanie tego
;             stanu rzeczy. Potem ustawia PSP  do naszego lokalnego PSP i DTA do PS;80h

PreserveDOS   proc    near
              assume  ds :ResidentSeg

              mov     ah, 51h                     ;pobranie PSP aplikacji
              int     21h
              mov     AppPSP, bx                  ;zachowanie na później

              mov     ah, 2Fh                     ;pobranie DTA aplikacji
              int     21h
              mov     word  ptr  AppDTA, bx
              mov     word  ptr  AppDTA+2, es

              push    ds
              mov     ah, 59h                     ;pobranie rozszerzonej informacji o błędzie
              xor     bx, bx
              int     21h

              mov     cs: Xerr.eeDS, ds.
              pop     ds
              mov     XErr.eeAX, ax
              mov     XErr.eeBX, bx
              mov     XErr.eeCX, cx
              mov     XErr.eeDX, dx
              mov     XErr.eeSI, si
              mov     XErr.eeDI, di
              mov     XErr.eeES, es

; Okay, wskazują wskaźniki DOS'a na nas:

              mov     bx, PSP
              mov     ah, 50h                     ;ustawienie PSP
              int     21h

              push    ds.                         ;ustawienie DTA pod adresem PSP:80h
              mov     ds., PSP
              mov     dx, 80h
              mov     ah, 1Ah                     ;ustawienie funkcji DTA
              int     21h
              pop     ds.
```

```
                ret
PreserveDOS     endp
                assume  ds.:nothing

;RestoreDOS-    Przywraca ważne globalne dane  DOS'a  z powrotem  do wartości aplikacji

RestoreDOS      proc    near
                assume  ds.: ResidentSeg

                mov     bx, AppPSP
                mov     ah, 50h                 ;ustawienie PSP
                int     21h

                push    ds.
                lds     dx, AppDTA
                mov     ah, 1Ah                 ;ustawienie DTA
                int     21h
                pop     ds.
                push    ds.

                mov     si, offset XErr         ; zachowanie rozszerzonego  błędu
                mov     ax, 5D0Ah               ;przywrócenie funkcji XErr
                int     21h
                pop     ds.
                ret
RestoreDOS      endp
                assume  ds.:nothing

;DoIO-          Ten podprogram przetwarza każdą z tych operacji I/O wymagane do zapisania danych do pliku

DoIO            proc    near
                assume  ds.:ResidentSeg

                mov     NeedIO, 0FFh            ;dla nas flaga zajęta

; włączamy z powrotem przerwania (wyzerowaliśmy  sekcję krytyczną ponieważ zapisaliśmy 0FFh do NeedIO)

                sti
                call    PreserveDOS             ;zachowujemy dane DOS

                mov     ah, 2Ah                 ;funkcja Get Date DOS
                int     21h
                mov     month, dh
                mov     day, dl
                mov     year, cx

                mov     ah, 2Ch                 ;pobranie funkcji Get Time DOS
                int     21h
                mov     hour, ch
                mov     minute, cl
                mov     second, dh

                mov     ah, 40h                 ;funkcja zapisu DOS
                mov     bx, FileHandle          ;zapis danych do tego pliku
                mov     cx, RecSize             ;tyle bajtów
                mov     dx, offset month        ;zaczynamy od tego adresu
                int     21h                     ;ignorujemy zwracane błędy (!)
                mov     ah, 68h                 ;funkcja potwierdzająca DOS
                mov     bx, FileHandle          ;zapis danych do tego pliku
                int     21h                     ;ignorujemy zwracany błąd (!)
```

```
                mov     NeedIO, 0                       ;gotowy do ponownego startu
                call    RestoreDOS

PhasesDone:     ret
DoIO            endp
                assume  ds: nothing

;MyInt2F-       Dostarcza wsparcia int 2Fh (przerwanie równoczesnych procesów) dla tego TSR'a. Przerwanie
;               równoczesnych procesów rozpoznaje następujące pod funkcje (przekazane w AL.):
;               00 – Weryfikacja obecności.     Zwraca 0FFh w AL. i wskaźnik do ID ciągu w es:di jeśli
;                                               ID TSR'a (w AH) pasuje do tego szczególnego TSR'a
;
;               01 – Usunięcie                  Usuwa TSR z pamięci. Zwraca 0 w AL. jeśli pomyślnie
;                                               1 w AL. jeśli niepowodzenie
;

MyInt2F         proc    far
                assume  ds:nothing

                cmp     ah, MyTSRID             ;Pasuje do naszego identyfikatora TSR'a?
                je      YepItsOurs
                jmp     OldInt2F

;Okay, wiemy ,że to jest nasze ID, teraz sprawdzamy na możliwość funkcji usuwającej

YepItsOurs:     cmp     al., 0                  ;weryfikacja funkcji
                jne     TryRmv
                mov     al., 0ffh               ;zwraca z powodzeniem
                lesi    IDString
                iret                            ;powrót do kodu wywołującego

IDString        byte    :Keypress Logger  TSR", 0
TryRmv:         cmp     al, 1                   ;wywołanie usuwania
                jne     IllegalOp

                call    TstRmvable              ;zobacz czy można usunąć
                je      CanRemove               ;skocz jeśli nie można
                mov     ax, 1                   ;teraz zwraca niepowodzenie
                iret

;Okay,  chcemy usunąć go i możemy usunąć go z pamięci. Zajmiemy się wszystkim tutaj

                assume  ds:nothing

CanRemove       push    ds.
                push    es
                pusha
                cli                             ;wyłączając przerwania nabroimy w wektorach przerwań
                mov     ax, 0
                mov     es, ax
                mov     ax, cs
                mov     ds., ax

                mov     ax, word  ptr  OldInt9
                mov     es:[9*4], ax
                mov     ax, word  ptr OldInt9+2
                mov     es:[9*4+2], ax

                mov     ax, word  ptr  OldInt13
                mov     es:[13h*4], ax
```

```
                mov     ax, word  ptr OldInt13+2
                mov     es:[13h*4+2], ax


                mov     ax, word  ptr  OldInt16
                mov     es:[16h*4], ax
                mov     ax, word  ptr OldInt16+2
                mov     es:[16h*4+2], ax

                mov     ax, word  ptr  OldInt1C
                mov     es:[1Ch*4], ax
                mov     ax, word  ptr OldInt1C+2
                mov     es:[1Ch*4+2], ax

                mov     ax, word  ptr  OldInt28
                mov     es:[28*4], ax
                mov     ax, word  ptr OldInt28+2
                mov     es:[28*4+2], ax

                mov     ax, word  ptr  OldInt2F
                mov     es:[2Fh*4], ax
                mov     ax, word  ptr OldInt2F+2
                mov     es:[2Fh*4+2], ax

;Okay,  w ten sposób zamknęliśmy plik. Notka: INT 2F nie powinno musieć działać z DOS ponieważ jest to
; funkcja biernego TSR'a

                mov     ah, 3Eh                 ;polecenia zamknięcia pliku
                mov     bx, FileHandle
                int     21h

;Okay, ostatnia rzecz przed wyjściem – Oddajmy zaalokowaną pamięć dla tego TSR'a  z powrotem do DOS

                mov     ds., PSP
                mov     es, ds.:[2Ch]           ;wskaźnik do bloku środowiska
                mov     ah, 49h                 ;DOS zwalnia funkcję pamięci
                int     21h
                mov     ax, ds                  ;przestrzeń zwalnianego kodu programu
                mov     es, ax
                mov     ah, 49h
                int     21h

                popa
                pop     es
                pop     ds.
                mov     ax, 0                   ;zwrócone z powodzeniem

,Wywołanie z niepoprawnymi wartościami funkcji. Próbujemy zrobić to z jak najmniejszą szkodą

IllegalOp:      mov     ax, 0                   ;Kto wie co  myślano?
                iIret
MyInt2F         endp
                assume  ds. :nothing

; TstRmvable -  Sprawdzamy aby zobaczyć czy możemy usunąć ten TSR z pamięci. Zwraca ustawioną flagę
;               jeśli możemy usunąć go, wyzerowaną w przeciwnym razie

TstRmvable      proc    near
                cli
                push    ds
```

```
                mov     ax, 0
                mov     ds, ax

                cmp     word  ptr  ds:[9*4], offset  MyInt9
                jne     TRDone
                cmp     word ptr  ds:[9*4 +2], seg MyInt9
                jne     TRDone

                cmp     word  ptr  ds:[13h*4], offset  MyInt13
                jne     TRDone
                cmp     word ptr  ds:[13h*4 +2], seg MyInt13
                jne     TRDone

                cmp     word  ptr  ds:[16h*4], offset  MyInt16
                jne     TRDone
                cmp     word ptr  ds:[16h*4 +2], seg MyInt16
                jne     TRDone

                cmp     word  ptr  ds:[1Ch*4], offset  MyInt1Ch
                jne     TRDone
                cmp     word ptr  ds:[1Ch*4 +2], seg MyInt1Ch
                jne     TRDone

                cmp     word  ptr  ds:[28h*4], offset  MyInt28
                jne     TRDone
                cmp     word ptr  ds:[28h*4 +2], seg MyInt28
                jne     TRDone

                cmp     word  ptr  ds:[2Fh*4], offset  MyInt2F
                jne     TRDone
                cmp     word ptr  ds:[2Fh*4 +2], seg MyInt2F
TRDone:         pop     ds
                sti
                ret
TstRmvable      endp
ResidentSeg     ends

cseg            segment         para public  'code'
                assume          cs:cseg,  ds: ResidentSeg

;SeeIfPresent-          Sprawdzamy aby zobaczyć czy nas TSR jest już obecny w pamięci. Ustawia flagę
;                       zera jeśli jest, wyzerowaną jeśli nie jest

SeeIfPresent    proc    near
                push    es
                push    ds.
                push    di
                mov     cx, 0ffh                ;zaczynamy z ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                   ;funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                  ;obecny w pamięci?
                je      TryNext
                strcmpl
                byte    „Keypress Logger TSR”, 0
                je      Success

TryNext:        dec     cl                      ;testuje ID użytkownika od 80h..FFh
```

```
                js      IDLoop
                cmp     cx, 0                   ;zeruje flagę zera
Success:        pop     di
                pop     ds
                pop     es
                ret
SeeIfPresent    endp

;FindID-        Określamy pierwszy (cóż, ostatni właściwie) ID TSR'a dostępnego w łańcuchu przerwań
;               równoczesnych procesów. Zawraca tą wartość w rejestrze CL
;
;               Zwraca ustawioną flagę zera jeśli zlokalizuje puste gniazdo
;               Zwraca wyzerowaną flagę zera jeśli niepowodzenie

FindID          proc    near
                push    es
                push    ds
                push    di

                mov     cx, 0ffh                ;zaczynamy z ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                   ;funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                  ;obecny w pamięci?
                je      Success
                dec     cl                      ;testowanie ID użytkownika od 80h..FFh
                js      IDLoop
                xor     cx, cx
                cmp     cx, 1                   ;zerowanie flagi zera
Success:        pop     di
                pop     ds.
                pop     es
                ret
FindID          endp

Main            proc
                meminit

                mov     ax, residentSeg
                mov     ds, ax

                mov     ah, 62h                 ;pobranie wartości PSP tego programu
                int     21h
                mov     PSP, bx

;Zanim zrobimy cokolwiek. Musimy sprawdzić parametry linii poleceń. Musimy mieć albo poprawną
; nazwę pliku albo polecenie „usuń". Jeśli  usunięcie pojawi się w linii poleceń, wtedy usuwając  rezydenta
;  kopiujemy  z pamięci używając przerwania równoczesnych procesów (2Fh) . Jeśli usunięcia nie ma w linii
;poleceń, będziemy mieli nazwę pliku i lepiej będzie nie kopiować już załadowanego do pamięci

                argc
                cmp     cx,1                    ;musi mieć dokładnie jeden parametr
                je      GoodParamCnt
                print
                byte    „Usage: „ , cr, lf
                byte    " KeyEval  filename", cr. Lf
                byte    " or KeyEval REMOVE", cr, lf, 0
                ExitPgm
```

```
; Sprowadzenie dla polecenia REMOVE

GoodParamCnt:  mov      ax,1
               argv
               atricmpl
               byte     “REMOVE”. 0
               jne      TstPresent

               call     SeeIfPresent
               je       RemoveIt
               print
               byte     “TSR is not present in memory, cannot rmove”
               byte     cr, lf, 0
               ExitPgm

RemoveIt:      mov      MyTSRID, cl
               printf
               byte     “Removing TSR (ID #%d) from memory...”,0
               dword    MyTSRID

               mov      ah, cl
               mov      al, 1                    ;usunięcie cmd, ah zawiera ID
               int      2Fh
               cmp      al., 1                   ;powodzenie?
               je       RmvFailure
               print
               byte     „removed>”, cr, lf, 0
               ExitPgm

RmvFailure:    print
               byte     cr, lf
               byte     “Could not remove TSR from memory.”, cr, lf
               byte     “Try  removing other TSRs in the reverse order “
               byte     “you instaled them.”, cr, lf, 0
               ExitPgm

;Okay, zobaczmy czy TSR jest już w pamięci. Jeśli tak przerywamy  proces instalacji

TstPresent:    call     SeeIfPresent
               jne      GetTSRID
               print
               byte     “TSR is already present in memory”, cr, lf
               byte     “Aborting instalation process”, cr, lf, 0
               ExtPgm

; Pobieramy ID dla naszego TSR’a i zapisujemy go

GetTSRID:      call     FindID
               je       GetFileName
               print
               byte     „Too many resident TSRs, cannot instal”, cr, lf,0
               ExitPgm

;sprawdzamy nazwę pliku i otwieramy  ten plik

GetFileName:   mov      MyTSRID, cl
               printf
               byte     “Keypress logger TSR program”,cr, lf
               byte     “TSR ID = %d”,cr, lf
```

```
                byte    "Processing file: ", 0
                dword   MyTSRID

                puts
                putcr

                mov     ah, 3Ch                         ;tworzymy plik
                mov     cx, 0                           ;normalny plik
                push    ds.
                push    es                              ;nazwę wskazuje ds.:dx
                pop     ds.
                mov     dx, di
                int     21h                             ;otwarcie pliku
                jnc     GoodOpen
                print
                byte    „DOS error #", 0
                puti
                print
                byte    „opening file.", cr,lf,0
                ExitPgm

GoodOpen:       pop     ds
                mov     FileHandle, ax                          ;zachowujemy uchwyt pliku

InstallInts:    print
                Byte    „Installing  interrrupts...", 0

;Aktualizujemy wektory przerwań int 9, 13h, 16h, 1Ch, 28h i 2Fh. Zauważmy ,ze powyższe instrukcje czynią
; ResidentSeg bieżącym segmentem danych, więc możemy przechować starą  wartość bezpośrednio w zmiennej
; OldIntxx

                cli                                     ;wyłączamy przerwania!
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[9*4]
                mov     word  ptr OldInt9, ax
                mov     ax, es:[9*4+2]
                mov     word  ptr OldInt9+2, ax
                mov     es:[9*4], offset MyInt9
                mov     es:[9*4+2], seg ResidentSeg


                mov     ax, es:[13h*4]
                mov     word  ptr OldInt13, ax
                mov     ax, es:[13h*4+2]
                mov     word  ptr OldInt13+2, ax
                mov     es:[13h*4], offset MyInt13
                mov     es:[13h*4+2], seg ResidentSeg

                mov     ax, es:[16h*4]
                mov     word  ptr OldInt16, ax
                mov     ax, es:[16h*4+2]
                mov     word  ptr OldInt16+2, ax
                mov     es:[16h*4], offset MyInt16
                mov     es:[16h*4+2], seg ResidentSeg

                mov     ax, es:[1Ch*4]
                mov     word  ptr OldInt1C, ax
                mov     ax, es:[1Ch*4+2]
                mov     word  ptr OldInt1C+2, ax
```

```
                mov     es:[1Ch*4], offset MyInt1C
                mov     es:[1Ch*4+2], seg ResidentSeg


                mov     ax, es:[28h*4]
                mov     word  ptr OldInt28, ax
                mov     ax, es:[28h*4+2]
                mov     word  ptr OldInt28+2, ax
                mov     es:[28h*4], offset MyInt28
                mov     es:[28h*4+2], seg ResidentSeg


                mov     ax, es:[2Fh*4]
                mov     word  ptr OldInt2F, ax
                mov     ax, es:[2Fh*4+2]
                mov     word  ptr OldInt2F+2, ax
                mov     es:[2Fh*4], offset MyInt2F
                mov     es:[2Fh*4+2], seg ResidentSeg
                sti                                     ;włączamy ponownie przerwania

; Jedyna rzecz jaka pozostaje to TSR

                print
                byte    „Installed.”, cr, lf, 0

                mov     dx, EndResident         ;Obliczamy rozmiar programu
                sub     dx, PSP
                mov     ax, 3100h               :polecenie TSR DOS
                int     21h
Main            endp
cseg            ends

sseg            segment         para  stack  ‘stack’
stk             db      1024  dup (“stack”)
sseg            ends

zzzzzzseg       segment         para  public ‘zzzzzz’
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

Poniżej mamy krótki przykład programu, który czyta dane z pliku tworzony z powyższego programu i tworzy prosty raport o dacie, czasie i naciskaniu klawiszy

```
;Program ten odczytuje plik stworzony przez program TSR  KEYEVAL.EXE . Wyświetla log zawierający dane,
; czas i liczbę naciśnięć klawiszy

                .xlist
                .286
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment         para  public  ‘data’

FileHandle      word    ?

month           byte    0
day             byte    0
year            byte    0
```

```
hour            byte    0
minute          byte    0
second          byte    0
KeyStrokes      word    0
RecSize         =       $ - month

dseg            ends

cseg            segemnt         para  public  'code'
                assume  cs: cseg, ds: dseg

,SeeIfPresent -         Sprawdzamy czy nasz TSR jest obecny w pamięci. Ustawiamy flagę zera jeśli jest
;                       zeruje jeśli nie jest
;
SeeIfPresent    proc    near
                push    es
                push    ds.
                pusha
                mov     cx, 0ffh                        ;Zaczynamy z ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                           ;weryfikujemy funkcję obecności
                int     2Fh
                pop     cx
                cmp     al., 0                          ;Obecny w pamięci?
                je      TryNext
                strcmpl
                byte    „Keypress Logger  TSR", 0
                je      Success

TryNext:        dec     cl                              ;Testujemy ID użytkownika od 80h..FFh
                js      IDLoop
                cmp     cx, 0                           ;Zerowanie flagi zera
Success:        popa
                pop     ds.
                pop     es
                ret
SeeIfPresent    endp

Main            proc
                meminit

                mov     ax, dseg
                mov     ds., ax

                argc
                cmp     cx, 1                   ;musi mieć przynajmniej jeden parametr
                je      GoodParmCnt
                print
                byte    „Usage:",cr,lf
                byte    " KEYRPT filename", cr, lf,0
                ExitPgm

GoodParmCnt     mov     ax, 1
                argv

                print
                byte    "Keypress logger report program", cr, lf
                byte    "Porcessing file:",0
                puts
```

```
                putcr

                mov     ah, 3Dh                 ;polecenie otwarcia pliku
                mov     al., 0                  ;otwarcie do odczytu
                push    ds.
                push    es                      ;ds.:dx wskazuje nazwę
                pop     ds.
                mov     dx, di
                int     21h                     ;otwarcie pliku
                jnc     GoodOpen
                print
                byte    „DOS error #”, 0
                puti
                print
                byte    , opening file.“, cr, lf,0
                ExitPgm

GoodOpen:       pop     ds
                mov     FileHandle, ax                  ;zachowanie uchwytu pliku

;Okay, czytamy daną i wyświetlamy ją:

ReadLoop:       mov     ah, 3Fh                 ;polecenie odczytu pliku
                mov     bx, FileHandle
                mov     cx, RecSize             ;liczba bajtów
                mov     dx, offset month        ;miejsce na umieszczenie danych
                int     21h
                jc      ReadError
                test    ax, ax                  ;EOF?
                je      Quit

                mov     cx, year
                mov     dl, day
                mov     dh, month
                dtoam
                puts
                free
                print
                byte    “, “, 0

                mov     ch, hour
                mov     cl, minute
                mov     dh, second
                mov     dl, 0
                ttoam
                puts
                free
                printf
                byte    “, keystroke = %d\n”, 0
                dword   KeyStorkes
                jmp     ReadLoop
ReadError:      print
                Byte    “Eror reading file”, cr, lf, 0

Quit:           mov     bx, FileHandle
                mov     ah, 3Eh                 ;zamknięcie pliku
                int     21h
                ExitPgm

Main            endp
```

```
cseg            ends

sseg            segemnt         para stack ‘stack’
stk             db      1024  dup  (“stack”)
sseg            ends

zzzzzzseg       segment         para  public ‘zzzzzz’
LastBytes       db      16  dup  (?)
Zzzzzzseg       ends
                end     Main
```

---

18. 9 PROGRAMY PÓŁREZYDENTNE

Program półrezydentny, jest to program, który czasowo ładuje się do pamięci, wykonuje inny program (proces potomny) a potem usuwa się z pamięci po zakończeniu procesu potomnego. Programy półrezydentne zachowują się jak programy rezydentne podczas wykonywania potomka, ale nie pozostają w pamięci kiedy potomek się kończy.

Głównym zastosowaniem programów półrezydentnych jest rozszerzenie istniejących aplikacji lub aktualizację aplikacji (proces potomny). Fajną rzeczą programu półrezydentnego jest to, ze nie musimy modyfikować pliku „.EXE” aplikacji bezpośrednio na dysku. Jeśli z jakiegoś powodu aktualizacja się nie powiodła, nie musimy niszczyć pliku „.EXE”, musimy tylko zlikwidować kod obiektu w pamięci

Aplikacja półrezydentna, podobnie jak TSR ma część nierezydentną i rezydentną . Część rezydentna pozostaje w pamięci dopóki wykonuje się proces potomny. Część nierezydentna inicjalizuje program a potem przekazuje sterowanie do części rezydentnej , która ładuje aplikację potomną nad częścią rezydentną. Kod nierezydentny aktualizuje wektory przerwań i robi wszystkie rzeczy jakie robi TSR z wyjątkiem, że nie wykonuje poleceń TSR’a. Zamiast tego, program rezydentny ładuje aplikację do pamięci i przekazuje sterowanie do tego programu. Kiedy aplikacja zwraca sterowanie do programu rezydentnego, wychodzi do DOS używając standardowej funkcji ExitPgm (ah = 4Ch)

Kiedy aplikacja jest uruchomiona, kod rezydentny zachowuje się jak inny TSR. Chyba ,że proces potomny jest świadomy programu półrezydentnego, lub program półrezydentny aktualizuje wektor przerwań normalnie używanej aplikacji, program półrezydentny prawdopodobnie będzie aktywnym programem rezydentnym aktualizującym jedno lub więcej przerwań sprzętowych. Oczywiście, wszystkie zasady które obowiązywały przy aktywnych TSR’ach również obowiązują przy aktywnych programach półrezydentnych

Poniżej jest bardzo ogólny przykład programu półrezydentnego. Program ten „RUN.ASM”, uruchamia aplikację, której nazwa i parametry linii poleceń pojawiają się jako parametry lini poleceń do uruchomienia. Innym słowy:

```
c> run  pgm. Exe parm1  parm2  itd.
```

jest odpowiednikiem

```
pgm  parm1  parm1  itd
```

Zauważmy, że musimy dostarczyć rozszerzenie „.EXE” lub „.COM” do nazwy programu. Kod ten zaczyn się od wyodrębnienia nazwy programu i parametrów lini poleceń z uruchomionej lini poleceń. Uruchamia wbudowaną strukturę exec a potem wywołuje DOS do wykonania programu. Przy zwrocie, uruchamia stały stos i wraca do DOS

```
;RUN.ASM -       szkielet programu półrezydentnego
;
;       Usage:
;               RUN <program.exe> <program’s command  line>
;       lub     RUN <program.com> <program’s command line>
;
; RUN wykonuje określony program  z dostarczonymi parametrami linii poleceń. Początkowo może to wyglądać
; na głupi program. W końcu dlaczego nie uruchomić programu bezpośrednio z DOS i zupełnie przeskoczyć
; RUN? W rzeczywistości jest dobry powód dla RUN – Pozwala nam (przez zmodyfikowanie pliku źródłowego
; RUN) ustawić  środowisko wcześniej uruchomionego programu i zeruje to środowisko po zakończeniu
; programu („środowisko” w tym sensie nie koniecznie odnosi się do obszaru środowiska MS-DOS).
```

```
;
; Na przykład, mamy użyć tego programu do przełączenia do trybu TSR wcześniej wykonywanego pliku EXE
; a potem przywrócić tryb działania tego TSR'a po zakończeniu programu.
;
; Ogólnie, możemy stworzyć nową wersję RUN.EXE (i, prawdopodobnie,  daje unikalny numer) dla każdej
aplikacji dla jakiej chcemy użyć tego programu
;
;
;-----------------------------------------------------------------------------------------------------------------
;
;
; wkładamy ten segment jako pierwszy ponieważ  chcemy załadować podprogramy Biblioteki Standardowej jako
; ostatnie w pamięci, więc  podciągają pod część nierezydentną

CSEG            segment  para  public  'CODE'
CSEG            ends
SSEG            segemnt para  stack  'stack'
SSEG            ends
ZZZZZZZSEG      segment para public ,zzzzzzseg'
ZZZZZZZSEG      ends

; zawieramy makra Biblioteki Standardowej UCR

                include  consts.a
                include  stdin.a
                include  stdout.a
                include  misc.a
                include  memory.a
                include  strings.a

                includelib          stdlib.lib

CSEG            segment   para  public  'CODE'
                assume    cs:cseg, ds:cseg

; Zmienne używane przez ten program

; struktura EXEC MS-DOS

ExecStruct      dw      0
                dd      CmdLine
                dd      DfltFCB
                dd      DfltFCB

DfltFCB         db      3, ,, ,,, 0, 0 ,0 ,0 ,0
CmdLine         db      0, 0dh, 126  dup (,, ,,);          ;linia poleceń dla programu
PgmName         dd      ?                                  ;wskazuje nazwę programu

Main            proc
                mov     ax, cseg                           ;pobranie wskaźnika do segmentu zmiennych
                mov     ds., ax

                Meminit                                    ;start menadżera pamięci

;jeśli chcemy zrobić coś zanim wykonamy linię poleceń określonego programu , tu jest dobre miejsce na to



; ----------------------------------------------------------------------------------------------------------
```

```
; Teraz pobieramy nazwę programu, itp., z linii poleceń  i wykonujemy go

                argc                            ;zobaczmy ile parametrów lini poleceń mamy
                or      cx, cx
                jz      Quit                    ;wychodzimy kiedy brak

                mov     ax, 1                   ;obranie pierwszego parametru (nazwa programu)
                argv
                mov     word  ptr  PgmName, di  ;zachowujemy wskaźnik do  nazwy
                mov     word  ptr PgmNanme+2, es

;Okay, dla każdego słowa w lini poleceń po nazwie pliku, kopiujemy to słowo do bufora CmdLine i
; oddzielamy każde słowo spacją, podobnie jak COMMAND.COM  robi a parametrami lini poleceń w
; procesie

                lea     si, CmdLine+1           ;indeks do cmdline
ParmLoop:       dec     cx
                jz      ExecutePgm

                inc     ax                      ;wskazuje następny parametr
                argv

                push    ax
                mov     byte  ptr [si], ' '     ; pierwsza pozycja i separator w linii
                inc     CmdLine
                inc     si
Cpylp:          mov     al., es:[di]
                cmp     al, 0
                je      StrDone
                inc     CmdLine                 ;zwiększenie bajtu
                mov     ds.:[si], al.
                inc     si
                inc     di
                jmp     CpyLp

StrDone:        mov     byte  ptr ds:[si], cr   ;w tym przypadku jest koniec
                pop     ax                      ;pobranie parametru #
                jmp     ParmLoop

;Okay, zbudujemy strukturę wykonującą MS-DOS i konieczną linię poleceń, teraz zobaczymy uruchamianie
; programu. Pierwszy krok to zwolnienie całej pamięci, której ten program nie uzywa.

ExecutePgm:     mov     ah, 62h                 ;pobranie wartości naszego PSP
                int     21h
                mov     es, bx
                mov     ax, zzzzzzseg           ;obliczamy rozmiar kodu rezydentnego
                sub     ax, bx
                mov     bx, ax
                mov     ah, 4ah                 ;udostępnienie nieużywanej pamięci
                int     21h

;Ostrzeżenie! Żadnej funkcja Biblioteki Standardowej po tym punkcie. Udostępniamy pamięć, którą
;one tu sytuują.

                mov     bx, seg  Execstruct
                mov     es, bx
                mov     bx, offset ExecStruct   ;wskaźnik  do rekordu programu
                lds     dx, PgmName
                mov     ax, 4b00h               ;exec pgm
                int     21h
```

```
; Kiedy wrócimy, nie możemy  liczyć ,ż będzie poprawnie. Najpierw, stały wskaźnik stosu a potem możemy
; zakończyć wszystko co mus być zrobione

                mov     ax, sseg
                mov     ss, ax
                mov     sp, offset  EndStk
                mov     ax, seg  cseg
                mov     ds, ax

;Okay, jeśli mamy wykonać jakąś wielką rzecz po programie, jest to dobre miejsce do wstawienia takiego czegoś
;
;  ---------------------------------------------------------------------------------------------
;
; Zwrócenie sterowania do MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment  para  stack  'stack'
                dw      128  dup (0)
endstk          dw      ?
sseg            ends

; zarezerwowanie jakieś miejsca dla sterty

zzzzzzseg       segment para public  'zzzzzzseg'
Heap            db      200h  dup ( ?)
zzzzzseg        ends
                end     Main
```

Ponieważ RUN.ASM jest raczej prosty, być może przyda się przykład bardziej złożony. Poniżej znajduje się w pełni funkcjonalna aktualizacja dla gry XWING™ firmy Lucasart. Motywacją tej aktualizacji była nieustanna irytacja koniecznością podawania hasła za każdym razem kiedy chce się zagrać w grę. Ta małą aktualizacja przeszukuje kod, który wywołuje podprogram hasła i przechowuje NOP'y ponad kodem w pamięci.

Działanie tego kodu jest trochę trudniejsze niż RUN.ASM. Program RUN wysyłał polecenie wykonania do DOS, który uruchamiał żądany program Wszystkie zmiany systemowe jakich RUN potrzebuje wykonać musi być zrobione przed lub po wykonaniu aplikacji. XWPATCH działa trochę inaczej. Ładuje program XWING.EXE do pamięci i przeszukuje jakiś określony kod (wywołujący podprogram hasła). Ponieważ znajduje ten kod, przechowuje instrukcje NOP na szczycie tego kodu.

Niestety, życie nie jest tak całkiem proste. Kiedy XWING.EXE się ładuje, kod hasła nie jest jeszcze obecny w pamięci. XWING ładuje ten kod później jako nakładkę. Więc program XWPATCH znajduje coś co XWING.EXE ładuje natychmiast do pamięci – kod joysticka. Z tego punktu widzenia, XWPATCH jest po prostu wchłania przestrzeń pamięci ; XWING nigdy nie wywoła go ponownie dopóki XWING się nie zakończy

```
;XWPATCH.ASM
;
;       Użycie:
;               XWPATCH -       musi być w tym samym katalogu co XWING.EXE
;
; Program ten wykonuje program XWING.EXE i  aktualizuje go tak, aby unikać wprowadzania
; hasła za każdym razem gdy go uruchamiamy.
;
; Program ten jest przedstawiony tylko do celów edukacyjnych. Jest on demonstracją tego jak napisać
; półrezydentny program. Nie jest intencją zachęcenie do piracenia programów komercyjnych
; takie zastosowani jest nielegalne i ścigane przez prawo.
; Program jest oferowany bez gwarancji na poprawne działanie. W związku z dynamiczną naturą
; projektowania, program który aktualizuje inny program może nie pracować z drobną zmianą w
; aktualizowanym programie (XWING.EXE). UŻYWASZ TEGO KODU NA WŁASNE RYZYKO.
;
```

```
;-------------------------------------------------------------------------------------------------------------------------

byp     textequ  <byte  ptr>
wp      texteequ <byte  ptr>

; Wkładamy tu definicję segmentu ponieważ Biblioteka Standardowa UCR będzie ładowana po zzzzzzseg
; (w sekcji rezydentnej).

cseg            segment  para  public  'CODE'
cseg            ends

sseg            segment  para  stack  'STACK'
sseg            ends

zzzzzzseg       segment   para  public  'zzzzzzseg'
zzzzzzseg       ends

                .286
                include         stdlib.a
                includelib      stdlib.lib

CSEG            segment  para public  'CODE'
                assume  cs:cseg, ds :nothing

;CountJSCalls – Liczba razy wywołań xwing przez kod Joystick zanim zaktualizujemy wywołanie hasła.

CountJSCalls    dw      250

;PSP -  Przedrostek  Segmentu Programu . Musimy zwolnić pamięć zanim uruchomimy program
;       rzeczywisty

PSP     dw      0

;Program ładuje struktury danych (dla DOS)

ExecStruct      dw      0
                dd      CmdLine
                dd      DfltFCB
                dd      DfltFCB
LoadSSSP        dd      ?
LoadCSIP        dd      ?
PgmName         dd      Pgm
DfltFCB         db      3, " ," , 0 ,0 ,0 ,0 ,0
CmdLine         db      2, "  ", 0dh, 16  dup  ("  ")              ;Linia poleceń dla programu
Pgm             db      'XWING.EXE", 0

;*****************************************************************************************
; XWPATCH zaczyn się tutaj. Jest to część rezydentna pamięci. Wkładamy tu kod, który musi być obecny w
; czasie wykonania lub musi być rezydentny po zwolnieniu pamięci.
;*****************************************************************************************

Main    proc
        mov     cs:PSP, ds.
        mov     ax, cseg                        ;pobranie wskaźnika do segmentu zmiennych
        mov     ds, ax

        mov     ax, zzzzzzseg
        mov     es, ax
        mov     cx, 1024/16
        meminit2
```

; Teraz, zwalniamy pamięć ZZZZZZSEG czyniąc miejsce dla XWING
;Notka: Absolutnie nie wywołujemy podprogramów Biblioteki Standardowej z tego punktu! (ExitPgm jest
; OK., jest to makro które wywołuje DOS) Zauważmy, że po wykonaniu tego kodu, żaden kod ani dane z ;zzzzzzseg nie jest poprawny

```
        mov     bx, zzzzzzseg
        sub     bx, PSP
        inc     bx
        mov     es, PSP
        mov     ah, 4ah
        int     21h
        jnc     GoodRealloc
```

;Okay, skłamałem. Tu jest wywołanie StdLib, ale jest OK. ponieważ nie zdołamy załadować aplikacji na szczyt ;kodu biblioteki standardowej. Ale od tego punktu absolutni żadnych więcej wywołań!

```
        print
        byte    „Memory allocation error”
        byte    cr, lf, 0
        jmp     Quit
```

GoodRealloc:

;Teraz ładujemy program XWING do pamięci:

```
        mov     bx, seg  ExecStruct
        mov     es, bx
        mov     bx, offset ExecStruct       ;wskaźnik do rekordu programu
        lds     dx, PgmName
        mov     ax, 4b01h                   ;ładownie , nie exec, pgm
        inc     21h
        jc      Quit                        jeśli błąd ładujemy plik
```

;Niestety, kod hasła jest ładowny dynamicznie później. Więc nie ma go nigdzie w pamięci, gdzie moglibyśmy
; go przeszukać. Ale wiemy, że kod joysticka jest w pamięci, więc przeszukujemy ten kod. Kiedy go znajdziemy
; zaktualizujemy go tak, że wywoła nasz program SearchPW Zauważmy, że musimy użyć joysticka (i mieć ;zainstalowany) aby ta łatka działała poprawnie

```
        mov     si, zzzzzzseg
        mov     ds., si
        xor     si, si

        mov     di, cs
        mov     es, di
        mov     di, offset  JoystickCode
        mov     cx, JoyLength
        call    FindCode
        jc      Quit                        ;jeśli nie znaleziono kodu joysticka
```

;Aktualizujemy tu kod joysticka XWING

```
        mov     byp  ds.:[si], 09ah         ;dalekie wywołani
        mov     wp   ds.:[si+1]             ;offset SearchPW
        mov     wp  ds:[si+3], cs
```

;Okay ,zaczynamy uruchamianie programu XWING.EXE

```
        mov     ah, 62                      ;pobranie PSP
        int     21h
        mov     ds., bx
```

```
                mov     es, bx
                mov     wp  ds:[10]  , offset Quit
                mov     wp  ds:[12], cs
                mov     ss  wp  cseg:LoadSSSP+2
                mov     sp  wp  cseg:LoadSSSP
                jmp     dword  ptr  cseg:LoadCSIP

Quit:           ExitPgm
Main            endp

; SearchPW  pobieranie wywołanie z XWING, kiedy próbuje skalibrować joystick. Wywołujemy z XWING
; joystick kilkaset razy zanim w rzeczywistości  wyszukamy kod hasła. Powód zrobienia jest taki, że XWING
; wywołuje kod joysticka wcześniej przy teście na obecność joysticka. Kiedy wchodzimy do kodu kalibracji
; wywołujemy odpowiednio  kod joysticka, więc kilkaset wywołań nie będzie bardzo długo tracić ważności
; Kiedy jesteśmy w kodzie kalibracji, kod hasła będzie załadowany do pamięci, więc możemy go tam
; przeszukać

SearchPW        proc    far
                cmp     cs: CountJSCalla, 0
                je      DoSearch
                dec     cs: CountJSCalls
                sti                                     ; kod jaki wykradliśmy z xwing do aktualizacji
                neg     bx
                neg     di
                ret

;Okay, przeszukujemy kod hasła

DoSearch:       push    bp
                mov     bp, sp
                push    ds
                push    es
                pusha

;Przeszukujemy kod hasła w pamięci:

                mov     si, zzzzzzseg
                mov     ds., si
                xor     si, si

                mov     di, cs
                mov     es, di
                mov     di, offset PasswordCode
                mov     cx, PWLength
                call    FindCode
                jc      NotThere                        ;nieśli nie znaleziono kodu hasła

; Aktualizujemy kod hasła XWING tutaj. Najpierw przechowujemy NOP'y ponad pięcioma bajtami
; dalekiego wywołania podprogramu hasła

                mov     byp  ds:[si+11], 090h           ;wyNOPowianie dalekiego wywołania
                mov     byp  ds:[si+12], 090h
                mov     byp  ds:[si+13], 090h
                mov     byp  ds:[si+14], 090h
                mov     byp  ds:[si+15], 090h

;Modyfikujemy adres powrotny i przywracamy zaktualizowany kod joysticka aby nie przeszkadzał skokom

NotThere:       sub     word  ptr  [bp+2], 5            ;zrobienie kopi adresu powrotnego
                les     bx, [bp+2]                      ;pobranie adresu powrotnego
```

```
;Zachowanie oryginalnego kodu joysticka po wywołaniu aktualizacji tego podprogramu

                mov     ax, word  ptr  JoyStickCode
                mov     es:[bx], ax
                mov     ax, word  ptr JoyStickCode+2
                mov     es:[bx+2], ax
                mov     al, byte  ptr  JoyStickCode +4
                mov     es:[bx+4], al

                popa
                pop     es
                pop     ds
                pop     bp
                ret
SearchPW        endp

;*****************************************************************************************
;
;  FindCode:    na wejściu, ES:DI wskazują na jakiś kod w *tym* programie, który pojawia się w grze
;               XWING. DS.:SI wskazują n blok pamięci w grze XWING. FindCode przeszukuje całą
;               pamięć aby znaleźć  podejrzany kawałek kodu i zwraca w DS.:SI wskazują początek tego
;               kodu. Ten kod zakłada, że znajdzie kod!
;               Zwraca wyzerowaną flagę przeniesienia jeśli go znajdzie, ustawioną jeśli nie
;

FindCode        proc    near
                push    ax
                push    bx
                push    dx

DoCmp:          mov     dx, 1000h               ;Szukanie w 4kb blokach
CmpLoop:        push    di                      ;zachowanie wskaźnika do kodu porównującego
                push    si                      ;zachowanie wskaźnika do początku ciągu
                push    cx                      ;zachowanie licznika
        repe    cmpsb
                pop     cx
                pop     si
                pop     di
                je      FoundCode
                inc     si
                dec     dx
                jne     CmpLoop
                sub     si, 1000h
                mov     ax, ds
                inc     ah
                mov     ds, ax
                cmp     ax, 9000h               ;zatrzymanie pod adresem 9000:0
                jb      DoCmp                   ;i  niepowodzenie jeśli nie znaleziono

                pop     dx
                pop     bx
                pop     ax
                stc
                ret

FoundCode:      pop     dx
                pop     bx
                pop     ax
                clc
                ret
```

```
FindCode        endp

;*****************************************************************************************
;
; Wywołanie kodu hasła, który pojawia się w grze XWING. Jest to zazwyczaj dana, której mamy zamiar
; poszukać w kodzie obiektu XWING

PasswordCode    proc    near
                call    $-47h
                mov     [bp-4], ax
                mov     bp-2], dx
                push    dx
                push    ax
                byte    9ah, 04h, 00
PasswordCode    endp
EndPW:

PWLength        =       EndPW – PasswordCode

; Poniżej mamy kod joysticka, który mamy zamiar przeszukać

JoyStickCode    proc    near
                sti
                neg     bx
                neg     di
                pop     bp
                pop     dx
                pop     cx
                ret
                mov     bp, bx
                in      al., dx
                mov     bl, al.
                not     al
                and     al, ah
                jnz     $+11h
                in      al, dx

JoystickCode    endp
EndJSC:

JoyLength       =       EndJSC – JoystickCode
cseg            ends

sseg            segment  para  stack ‘STACK’
                dw      256  dup (0)
endstk          dw      ?
sseg            ends

zzzzzzseg       segment  para  public  ‘zzzzzzseg’
Heap            db      1024   dup (0)
Zzzzzzseg       ends
                end     Main
```

18.10 PODSUMOWANIE

Programy rezydentne dostarczają małej ilości wielozadaniowości pojedynczych zadań w świecie DOS’a. DOS dostarcza wsparcia dla programów rezydentnych w całym elementarnym systemie zarządzania pamięcią. Kiedy aplikacja korzysta z funkcji TSR, DOS modyfikuje wskaźnik pamięci aby zarezerwowana przestrzeń pamięci przez kod TSR’a był chroniony przed działaniami załadowanych w przyszłości programów. Po więcej szczegółów dotyczących tego procesu zobacz:

*”Używanie pamięci przez DOS i TSR’y”

TSR posiada dwie podstawowe formy: aktywną i bierną. Bierne TSR’y nie są samo aktywujące. Pierwszoplanowa aplikacja musi wywołać podprogram pasywnego TSR’a a by go aktywować. Generalnie, interfejs aplikacji pasywnego TSR’a używa mechanizmu przerwań kontrolowanych 80x86 (przerwania programowe). Uruchomienie aktywnego TSR, z drugiej strony, nie zależy od pierwszoplanowej aplikacji. Zamiast tego, łączą się one z przerwaniami sprzętowymi, które aktywują je niezależnie od procesu pierwszoplanowego.

*”Aktywne i pasywne TSR’y”

Natura aktywnych TSR’ów wprowadza wiele problemów zgodności. Podstawowym problemem jest to, że aktywny TSR może chcieć wywołać podprogram DOS’a lub BIOS’a, mając właśnie przerwany jeden z tych systemów. Stwarza to problem ponieważ DOS i BIOS nie są współbieżne. Na szczęście, MS-DOS dostarcza kilka punktów zaczepienia, które dają aktywnym TSR’om zdolność do planowania wywołań DOS kiedy DOS jest nieaktywny. Chociaż podprogramy BIOS nie dostarczają takiej samej zdolności, łatwo jest dodać otoczkę wokół wywołania BIOS pozwalająca nam planować poprawnie wywołania. Dodatkowym problemem z DOS jest to, że aktywny TSR może przeszkadzać jakimś globalnym zmiennym używanym przez proces pierwszoplanowy. Na szczęście DOS pozwala TSR’om zachować i przywracać te wartości, zapobiegając złośliwym problemom zgodności.

*„Współbieżność”
*”Problemy współbieżności z DOS”
*”Problemy współbieżności z BIOS”
*„Problem współbieżności z innym kodem”
*”Inne zagadnienia związane z DOS”

MS-DOS dostarcza specjalnego przerwania do koordynowania komunikacji pomiędzy TSR’ami a innymi aplikacjami. Przerwanie równoczesnych procesów pozwala nam łatwo sprawdzić obecność TSR’a w pamięci, usuwać TSR z pamięci , lub przekazywać różne informacje pomiędzy TSR’em a aktywną aplikacją.

*”Przerwanie równoczesnych procesów (INT 2Fh)

Cóż, pisanie TSR’ów trzyma się surowych zasad. W szczególności, dobry TSR zachowuje pewne konwencje podczas instalacji i zawsze dostarcza użytkownikowi bezpiecznego mechanizmu usunięcia całej wolnej pamięci będącej w użyciu przez TSR. W tych rzadkich przypadkach gdzie TSR nie może się sam usunąć, zawsze pokazuje właściwy błąd i instrukcję jak rozwiązać problem. Po więcej informacji o ładowaniu i usuwaniu TSR’ów zajrzyj

*”Instalowanie TSR’ów”
*”Usuwanie TSR’ów”
*”TSR monitora klawiatury”

Program półrezydentny jest programem, który jest rezydentny podczas wykonywania jakiegoś określonego programu. Sam automatycznie wyładowuje się kiedy kończy się aplikacja. Aplikacja półrezydentna znajduje aplikację jako program zaktualizowany i „TSR’em czasowo udostępnionym”

*”Programy półrezydentne”