Wprowadzenie

Biblia Turbo Assemblera składa się z czterech głównych części. Pierwsza z nich opisuje system operacyjny MS-DOS oraz zawiera wprowadzenie do Turbo Assemblera i Turbo Debuggera. W drugiej i trzeciej części znajduje się opis instrukcji i dyrektyw, używanych przez Turbo Assembler oraz dokładne objaśnienie sposobu ich działania. Ostatnia część zawiera przykłady, które ilustrują jak za pomocą asemblera można sterować różnymi podzespołami komputera.

CZĘŚĆ I: PROGRAMOWANIE W SYSTEMIE MS-DOS

Część ta zawiera podstawy programowania w systemie MS-DOS, opis możliwości Turbo Assemblera oraz sposób zastosowania Turbo Debuggera do śledzenia i testowania pisanych programów.

Rozdział 1: System operacyjny MS-DOS

W rozdziale tym został przestawiony sposób organizacji systemu MS-DOS. Zawiera on również informację o procesorach rodziny 80x86, funkcjach MS-DOS-a, ROM BIOS-a oraz informacje o wykorzystaniu pamięci.

Rozdział 2: Turbo Assembler

Rozdział ten to krótki przegląd instrukcji i dyrektyw, z których składa się program asemblerowy. Zostanie tu pokazane, w jaki sposób napisać, skompilować, skonsolidować i uruchomić prosty program. Rozdział ten zawiera podstawy niezbędne do zrozumienia i opanowania materiału dalszej części książki.

Rozdział 3: Turbo Debugger

Rozdział ten stanowi skrócony przegląd możliwości Turbo Debuggera. Zostaną one zaprezentowane na przykładzie, który dotyczy uruchamiania pewnego, zawierającego kilka błędów, programu.

CZĘŚĆ II: INSTRUKCJE PROCESORA

W części drugiej znajduje się opis wszystkich instrukcji procesorów od 8088 do 80386. Są one pogrupowane w czterech kategoriach.

Rozdział 4: Rozkazy transmisji danych

Instrukcje, znajdujące się w tym rozdziale, używane są do przesyłania danych pomiędzy pamięcią a rejestrami procesorów 80x86. Znajduje się tutaj również opis trybów adresowania, stosu czy rozkazów łańcuchowych.

Rozdział 5: Instrukcje arytmetyczne i logiczne

Rozdział ten zawiera opis instrukcji, używanych do manipulowania na danych. Przedstawione są rozkazy, które wykonują operacje arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie), logiczne (and, or, not, xor) czy przesuwania bitów.

Rozdział 6: Procedury, pętle i skoki

Instrukcje zawarte w tym rozdziale służą do sterowania wykonywaniem się programu. Rozkazy te mogą być, na przykład, zastosowane do utworzenia konstrukcji stosowanych w językach wysokiego poziomu, takich jak pętle *while* i *for*, instrukcja decyzyjna *if-then-else* czy funkcje.

Rozdział 7: Sterowanie procesorem i operacje w trybie chronionym

W rozdziale tym zostały opisane znaczniki i instrukcje, które sterują pracą procesora. Wiele z nich może być zastosowanych tylko w trybie chronionym układów 80286 i 80386. Rozdział ten jest uzupełniony przykładem, który przełącza procesor 80386 w tryb chroniony lub rzeczywisty.

CZĘŚĆ III: OPERATORY i DYREKTYWY TURBO ASSEMBLERA

W części tej znajduje się opis dyrektyw i operatorów udostępnianych przez Turbo Assembler. Zostaną przedstawione operacje na segmentach, sposoby przechowywania danych, tworzenia programów modułowych i używania zaawansowanych instrukcji kompilacji warunkowej.

Rozdział 8: Deklarowanie segmentów

W rozdziale tym znajduje się opis dyrektyw używanych do definiowania i wykorzystywania segmentów. Zostaną przedstawione uniwersalne, ale za to złożone, tradycyjne sposoby deklarowania segmentów, jak również nowe dyrektywy uproszczone.

Rozdział 9: Definicje zmiennych i ich rozmieszczenie w pamięci

W rozdziale tym można zobaczyć, jak za pomocą dyrektyw takich jak DB czy DW można tworzyć złożone struktury danych. Dzięki temu również w asemblerze można używać abstrakcyjnych typów danych, które istnieją w językach wysokiego poziomu takich jak C czy Pascal.

Rozdział 10: Makrodefinicje, procedury i programowanie modułowe

Programowanie modułowe bardzo ułatwia pisanie i zarządzanie dużymi programami. Rozdział ten przedstawia przeznaczone do tego celu dyrektywy Turbo Assemblera. Wyjaśniona jest różnica pomiędzy makroinstrukcjami i funkcjami oraz pokazany sposób tworzenia bibliotek procedur.

Rozdział 11: Obsługa błędów i tworzenie raportów

W rozdziale tym zostanie pokazane zastosowanie kompilacji warunkowej do tworzenia wielu wersji programu z jednego pliku źródłowego. Zostaną również przedstawione dyrektywy, które wpływają na sposób tłumaczenia pewnych instrukcji (np. instrukcji zmien-

noprzecinkowych). Ostatnia część rozdziału zajmuje się poleceniami, które sterują postacią raportu kompilacji.

CZĘŚĆ IV: TECHNIKI PROGRAMOWANIA

W rozdziale tym zostaną wykorzystane umiejętności nabyte w trakcie czytania poprzednich części książki. Przedstawiony zostanie sposób łączenia procedur, napisanych w asemblerze, z modułami powstałymi w językach wysokiego poziomu. Opisane zostaną zasoby komputera obsługiwane przez BIOS i MS-DOS oraz zasoby sprzętowe wykorzystywane bezpośrednio.

Rozdział 12: Moduły asemblerowe w językach wysokiego poziomu

Większość programistów pisze programy w językach wysokiego poziomu, a asembler używany jest przez nich tylko w szczególnych sytuacjach, gdy wymagana jest duża efektywność kodu lub specjalny rodzaj komunikacji ze sprzętem. W rozdziale tym przedstawiony jest sposób wywoływania i przekazywania argumentów do funkcji oraz współdzielenia danych pomiędzy modułami języków wysokiego poziomu i asemblerowymi. Zamieszczony przykład pokazuje, jak zamienić program powstały w języku C na procedurę asemblerową w celu zwiększenia jego efektywności.

Rozdział 13: Wykorzystanie zasobów systemu

Rozdział ten przedstawia sposób wykorzystania funkcji BIOS-a i MS-DOS-a do identyfikacji i zarządzania zasobami komputera. Dzięki temu można w pełni wykorzystać możliwości oferowane przez pamięć, drukarki, klawiaturę czy porty szeregowe.

Rozdział 14: Dostęp i sterowanie sprzętem

Rozdział ten stanowi kontynuację poprzedniego, lecz tym razem połączenie ze sprzętem jest jeszcze większe. Przedstawione są bowiem sposoby bezpośredniego działania na sprzęcie, w który wyposażone są komputery IBM PC. Zamieszczone przykłady ilustrują bezpośrednią współpracę z klawiaturą i portem szeregowym.

Rozdział 15: Karta graficzna: tryb tekstowy i graficzny

W rozdziale tym opisane są powszechnie stosowane w komputerach IBM PC karty graficzne CGA, EGA i VGA. Zamieszczone zostały również odpowiednie przykłady, które ilustrują sposoby wykorzystania ich możliwości.

ORGANIZACJA ROZDZIAŁÓW

Rozdziały od 4 do 11 zaczynają się pełnym opisem komend i instrukcji w nich zawartych. Wyjaśnione są tam wszystkie terminy i podstawowe zasady niezbędne do ich zrozumienia. Przy każdej instrukcji przedstawione są związane z nią polecenia. Dzięki temu można wybrać polecenie najbardziej w danej sytuacji odpowiednie. Zamieszczone jest również wiele przykładów, które ilustrują ich użycie.

Po każdej części opisowej znajdują się strony, które zawierają krótką informację o każdej komendzie przedstawionej w danym rozdziale. Składa się ona z opisu zastosowania danej

instrukcji, jej składni i przykładu ilustrującego jej użycie. Na rysunkach I-1 oraz I-2 pokazany jest układ tych stron.
W celu ułatwienia znalezienia informacji o potrzebnej instrukcji na końcu książki został zamieszczony skorowidz. Zawiera on wszystkie występujące komendy wraz z numerami stron, na których znajduje się ich opis.

Pełna nazwa instrukcji

Kropka pojawia się gdy dana instrukcja jest osiągalna odpowiednio w procesorach 8088/8086, 80286, 80386

Mnemonik instrukcji

Krótki opis użycia funkcji

Składnia pokazuje mnemoniki i operandy

Opisuje oddziaływanie instrukcji na znaczniki

Dodatkowe uwagi o instrukcji, dozwolonych kombinacjach operandów i typowe wykorzystanie w programach

Odwołania do innych instrukcji wraz z opisem

Pokazuje kod instrukcji i taktowanie dla każdego trybu adresowania. Taktowanie podaje liczbę cykli zegarowych potrzebnych do wykonania instrukcji. Oto opis używanych symboli:
88 = 8088, 86 = 8086, 286 = 80286 w trybie rzeczywistym, 386 = 80386 w trybie rzeczywistym
(w) = Liczba cykli dla operacji na słowie w 8088
znak - w informacji o taktowaniu oznacza zakres a nie odejmowanie
n = liczba repetycji lub przesunieć bitów, (noj) = brak skoku (no jump taken)
EA = Czas potrzebny do obliczenia adresu efektywnego. Zależy od trybu adresowania:

Tryb	Przykład Operandu	Liczba cykli
bazowy lub indeksowy	[bx], [bp], [si], [di]	5
przemieszczenie	etykieta	6
bazowy + indeksowy	[bx+si], [bp+di], [bx+di], [bp+si]	7 8
bazowy lub indeksowy + przem.	[bp+6], tablica [si]	9
bazowy + indeksowy + przem.	[bx+si+2], [bp+di+4] [bx+di+6], [bp+si+2]	11 12
zmiana segmentu	ds:[bp+10]	Dodać 2 cykle zegarowe

Jeden lub więcej przykładów ilustrujących wykorzystanie instrukcji

Pop Data Off of the Stack — 88/86 • 286 • 386 •

POP

ZASTOSOWANIE

Pobranie słowa lub podwójnego słowa (tylko 80386) ze szczytu stosu, a następnie modyfikacja wskaźnika stosu.

SKŁADNIA

`pop reg/mem` — *Pobranie danej ze szczytu stosu do rejestru albo miejsca w pamięci*

ZNACZNIKI

Bez zmian

KOMENTARZ

Stos jest używany do przechowania zmiennych tymczasowych oraz adresów powrotu z funkcji. Jedynie w trybie chronionym sprawdzane jest jego przepełnienie. Programista powinien więc upewnić się, czy na stosie jest wystarczająco dużo miejsca dla potrzeb programu.

PATRZ

POPA, POPF — *Inne instrukcje do pobierania elementów ze stosu*
PUSH, PUSHA, PUSHD — *Umieszczanie danych na stosie*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg	0101 lreg	pop ax	12/8	5	4
mem	10001111 mod,000,r/m przesunięcie	pop data	25/17+ EA	5	5
segreg	000sreg111	pop es	12/8	5	7
80386 segreg	00001111 10,sreg,001	pop fs	-	-	7

PRZYKŁAD

Stos jest często wykorzystywany do zapamiętywania wartości rejestrów. Mogą one być następnie użyte do innych celów. Jest to szczególnie ważne w wypadku rejestrów, które mają specjalne zastosowanie.

```
;Poniższa procedura przepisuje łańcuch Source do Dest.
;Rozmiar łańcucha jest podany w jego pierwszym bajcie.
;Rejestry CX, SI, DI i ES są odkładane na stos
;i zdejmowane z niego po dokonaniu przesłania.
SaveCpy  PROC
         push  cx                  ; Zachowaj rejestr CX
         push  si                  ; Zachowaj rejestr SI
         push  di                  ; Zachowaj rejestr DI
         push  es                  ; Zachowaj rejestr ES
         mov   si, offset Source
         mov   di, offset Dest
         push  ds                  ; Skopiuj rejestr DS
         pop   es                  ; do ES
         mov   cl, [si]            ; Pobierz długość
                                   ; łańcucha
```

Rysunek I-1. ***Format strony zawierającej opis instrukcji procesora 80x86.***

Pełna nazwa dyrektywy → **ASSUME**

Krótki opis sposobu wykorzystania dyrektywy → ***ZASTOSOWANIE***

Przekazanie Turbo Assemblerowi informacji, które segmenty są wskazywane aktualnie przez rejestry segmentowe.

Składnia pokazująca operandy (jeśli występują) → ***SKŁADNIA***

```
ASSUME  segreg:segment [, segreg:segment]...
ASSUME  segreg:NOTHING
ASSUME  NOTHING
```

Szczegółowe informacje o dyrektywie wyjaśniające gdzie jest ona szczególnie przydatna i dlaczego. → ***KOMENTARZ***

Ponieważ segmenty mogą się nakładać wiele par segment:przemieszczenie może wskazywać to samo miejsce pamięci. By móc obliczyć przemieszczenie, Turbo Assembler musi wiedzieć, jakie wartości zawierają poszczególne rejestry segmentowe. Dyrektywa ASSUME pozwala na przekazanie tego typu informacji.
Dyrektywa ta nie powoduje jednak załadowania rejestrów segmentowych. Należy się więc upewnić, czy w rejestrach znajduje się ta sama wartość, co użyta w dyrektywie.

Odwołania do innych dyrektyw wraz z opisem ich działania → ***PATRZ***

SEGMENT

Przykład ilustrujący użycie dyrektywy → ***PRZYKŁAD***

Poniższy przykład zawiera typową dyrektywę ASSUME stosowaną w programach z oddzielnymi segmentami kodu, danych i stosu. Rejestry CS i SS zostaną wypełnione podczas ładowania programu. Natomiast rejestr DS musi zostać załadowany przez program. Ponieważ zawartość rejestru ES jest przypadkowa, to przyjmuje się, że nie wskazuje on na nic.

```
ASSUME  CS:CODE, DS:DATA, SS:STACK, ES:NOTHING
mov     ax, SEG DATA  ; Umieść adres segmentu DATA w AX
mov     ds, ax        ; i prześlij go do rejestru DS
.
.
.
```

Rysunek I-2. ***Format strony zawierającej opis dyrektywy Turbo Assemblera.***

I Programowanie w Systemie MS-DOS

» System operacyjny MS-DOS

» Turbo Assembler

» Turbo Debugger

Rozdział 1 *System operacyjny MS-DOS*

Pierwszym krokiem przy pisaniu programów komputerowych jest zapoznanie się ze środowiskiem, w którym mają one pracować. Procesor, pamięć, urządzenia zewnętrzne lub system operacyjny mają duży wpływ na kształt pisanych programów. Nie projektuje się przecież wielkich graficznych interfejsów dla komputerów wyposażonych w procesor o małej mocy obliczeniowej i dysponujących niewielką ilością pamięci. O wiele lepszym rozwiązaniem dla tej klasy sprzętu jest prosty system tekstowy.
Większość programów napisanych w Turbo Assemblerze jest uruchamiana na komputerach klasy IBM PC i pracuje pod kontrolą systemu operacyjnego MS-DOS. Dlatego też znajomość tego systemu jest niezbędna przy projektowaniu oprogramowania. Rozdział ten wyjaśni, co to jest MS-DOS oraz to jak współpracują z nim programy użytkownika.
Jeśli programowałeś już w językach wysokiego poziomu, to część informacji, zawartych w tym rozdziale, może być ci znana. Lecz teraz musisz spojrzeć na te zagadnienia z innej strony, z punktu widzenia języka asemblera. Często jest to równoznaczne sięgnięciu do detali, których nie musiałeś brać pod uwagę, programując w językach takich jak C czy Pascal.
Każdy program musi w jakimś stopniu współpracować z systemem operacyjnym. W najprostszym przypadku MS-DOS ładuje go do pamięci i uruchamia. Ale może on też zarządzać pamięcią operacyjną czy urządzeniami zewnętrznymi. Część pierwsza tego rozdziału opisuje sposób działania systemu operacyjnego oraz interakcję z nim programów użytkownika.
System operacyjny MS-DOS przeznaczony jest dla procesorów rodziny 80x86. Każda instrukcja języka asemblera odpowiada jednej instrukcji procesora 80x86. Dlatego też konieczne jest dokładne zrozumienie sposobu działania tego procesora. Opis jego architektury zawarty jest w pierwszej części tego rozdziału.

Rodzina procesorów 80x86

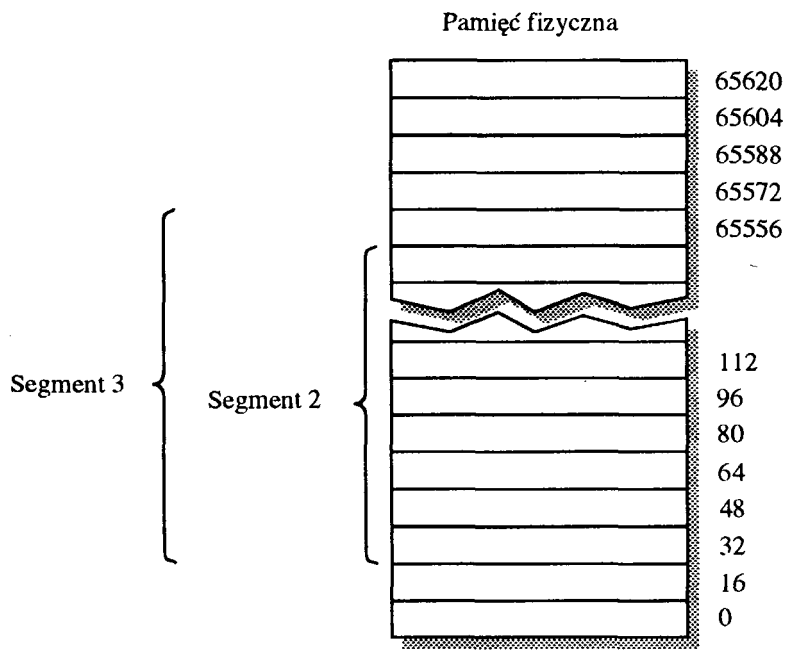
Wszystkie komputery, pracujące pod kontrolą systemu MS-DOS, są zbudowane na bazie mikroprocesorów z rodziny 80x86 firmy INTEL. Nowe generacje procesorów są kompatybilne z już istniejącymi. Oznacza to, że programy napisane przy użyciu instrukcji najstarszego z nich (8088) będą działały na każdym komputerze, który jest wyposażony w mikroprocesor tej rodziny. Na razie instrukcje dodane w nowych układach nie są zbyt często wykorzystywane w programach. Tak więc główną zaletą nowych generacji mikroprocesorów pozostaje większa szybkość.
Inną ważną ich cechą jest możliwość zaadresowania większej ilości pamięci. Procesory 8088 i 8086 mogą zaadresować do 1 MB (1,048,576 bajtów) pamięci, 80286 posiada już przestrzeń adresową równą 16 MB, a 80386 ma możliwość zaadresowania 4 GB (4,294,967,296 bajtów). Niestety, MS-DOS nie potrafi wykorzystać całej tej pamięci. Można samemu napisać program, który pozwoli dostać się do niej bez wiedzy systemu operacyjnego. Lecz lepszym sposobem obsługi pamięci rozszerzonej jest zainstalowanie programu, który wykonuje to zadanie. Jego przewagą jest to, że ma on wbudowany algorytm

przydziału pamięci, taki, jaki ma MS-DOS dla pamięci głównej. Zapobiega to oddziaływaniu na siebie wielu programów, które używają pamięci rozszerzonej.

Segmentacja pamięci

Pamięć w programach dla procesorów 80x86 podzielona jest na segmenty o maksymalnym rozmiarze 64K, które mogą na siebie zachodzić. Każde odwołanie do pamięci musi więc zawierać adres segmentu i przemieszczenie danej w segmencie. Procesor ma cztery rejestry segmentowe, które są wykorzystywane do adresowania pamięci operacyjnej. Rejestr segmentowy kodu programu (CS) wskazuje segment, z którego pobierane są kolejne rozkazy do wykonania. Rejestr segmentowy danych (DS) wskazuje segment, w którym zapamiętywane są dane używane w programie. Rejestr segmentowy stosu (SS) wskazuje segment pamięci, w którym zdefiniowany jest stos. Rejestr segmentowy dodatkowy (ES) wskazuje dodatkowy segment danych. Powszechną notacją, stosowaną dla oznaczenia adresu, jest zapisanie najpierw adresu segmentu, a za nim, po dwukropku, przesunięcia. Na przykład, dana w segmencie 40h i przesunięciu 12h jest zapisywana jako 40:12. Można również używać nazw segmentów. Dla danej o przesunięciu 24h, znajdującej się w segmencie danych, piszemy DS:24.

Rysunek 1-1 ilustruje zasadę segmentacji pamięci. Każdy segment zaczyna się na granicy 16 bajtów. Pozwala to na umieszczenie 20-bitowego adresu w 16-bitowym rejestrze. Procesor przyjmuje, że najmłodsze 4 bity są równe 0. Natomiast, odwołując się do danej, procesor dodaje do 20-bitowego adresu segmentu 16-bitowe przesunięcie tej danej w segmencie. Jest to jej adres fizyczny w pamięci operacyjnej.



Rysunek 1-1. *Segmentacja pamięci.*

MS-DOS automatycznie wypełnia rejestry CS, ES i SS. Natomiast program użytkownika musi sam załadować rejestr DS adresem segmentu danych programu. Oczywiście, jeśli program posiada więcej niż 64K pamięci przeznaczonej na dane, to musi odpowiednio modyfikować zawartości rejestrów DS i ES.

Rejestry

Dostępnych jest dziewięć 16-bitowych rejestrów. Osiem z nich może być wykorzystanych w instrukcjach matematycznych lub instrukcjach przesłania. Dziewiąty to wskaźnik rozkazów (IP). Zawiera on adres następnej do wykonania instrukcji. Istnieją specjalne rozkazy, które mogą zmienić jego zawartość.

Wyżej wymienione rejestry nie są dokładnie rejestrami ogólnego przeznaczenia. Mimo że mogą pełnić rolę rejestrów danych, każdy z nich, w niektórych instrukcjach, posiada specjalną funkcję. Ich zastosowanie przedstawia tabela 1-1.

Istnieje również osiem 8-bitowych rejestrów. Są one starszymi i młodszymi bajtami rejestrów AX, BX, CX i DX. Pierwsza litera nazwy każdego rejestru 8-bitowego jest taka sama jak pierwsza litera rejestru 16-bitowego w skład którego on wchodzi. Drugą literą jest H (oznaczenie starszego bajtu) lub L (oznaczenie młodszego bajtu). Na przykład, rejestr AX jest podzielony na rejestry AH i AL.

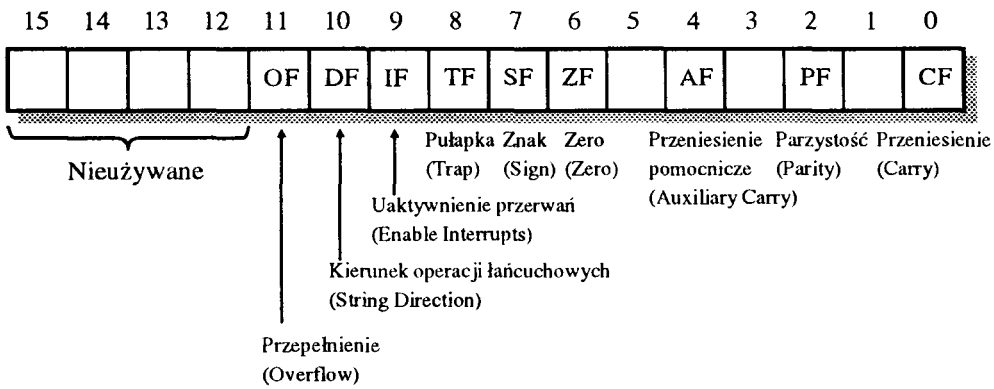
Ostatni rejestr to rejestr znaczników. Informują one o pewnych cechach wyniku operacji arytmetycznej bądź logicznej, stanie przerwań, kierunku operacji łańcuchowych. Rysunek 1-2 objaśnia znaczenie poszczególnych bitów tego rejestru. Istnieją instrukcje, które pozwalają ustawiać je, zerować i testować. Są one przedstawione w tabeli 1-2.

Tabela 1-1. ***Zastosowanie rejestrów***

Rejestr	Zastosowanie
AX	Akumulator. Niektóre instrukcje (mnożenie, dzielenie) mogą używać tylko akumulatora.
BX	Rejestr bazowy. Zawiera przemieszczenie względem początku segmentu danych.
CX	Rejestr licznikowy. Określa liczbę wykonań grupy instrukcji.
DX	Rozszerzenie akumulatora w 32-bitowych operacjach.
SI	Rejestr indeksowy. Zawiera adres źródła w operacjach łańcuchowych.
DI	Rejestr indeksowy. Zawiera adres przeznaczenia w operacjach łańcuchowych.
SP	Wskaźnik stosu.
BP	Rejestr bazowy. Zawiera przemieszczenie względem początku segmentu stosu.
CS	Rejestr segmentowy, numer segmentu kodu programu.
DS	Rejestr segmentowy, numer segmentu danych programu.
ES	Rejestr segmentowy, numer dodatkowego segmentu danych programu.
SS	Rejestr segmentowy, numer segmentu stosu programu.

Tabela 1-2. ***Sterowanie znaczników***

Nazwa	Opis
CLC	Wyzerowanie znacznika przeniesienia
CLD	Wyzerowanie znacznika kierunku
CLI	Wyzerowanie znacznika przerwania
CMC	Zanegowanie znacznika przeniesienia
LAHF	Załadowanie młodszego bajtu rejestru znaczników do AH
POPF	Pobranie znaczników ze stosu
PUSHF	Odłożenie na stos rejestru znaczników
SAHF	Przesłanie rejestru AH do młodszego bajtu rejestru znaczników
STC	Ustawienie znacznika przeniesienia
STD	Ustawienie znacznika kierunku
STI	Ustawienie znacznika przerwań



Rysunek 1-2. ***Rejestr znaczników.***

Stos

Aby prawidłowo wywołać podprogram konieczne jest istnienie stosu. Program, wywołując podprogram, odkłada na stos adres następnej instrukcji za rozkazem wywołania. Po jego wykonaniu adres ten jest pobierany do wskaźnika rozkazów i dzięki temu program może kontynuować swoje działanie od tego właśnie miejsca. Stos jest również używany do tymczasowego przechowywania zawartości rejestrów, podczas gdy one same są wykorzystywane do innych celów. Na przykład, by w programie wykonać pewną sekwencję instrukcji kilka razy, należy użyć rejestru CX. Jeśli zawiera on ważną informację, to program powinien odłożyć jego zawartość na stos, a po wykonaniu pętli pobrać ją z niego.
Wszystkie operacje stosowe dotyczą segmentu pamięci, wskazywanego przez rejestr SS. Rejestr SP zawiera przesunięcie danej ostatnio odłożonej na stos. Gdy program zapisuje na

stos daną, to procesor zmniejsza SP o dwa i umieszcza ją pod adresem SS:SP. Pobierając daną ze stosu procesor, odczytuje ją spod adresu SS:SP, a następnie zwiększa rejestr SP o dwa. Operacje na stosie zawsze dotyczą danej 16-bitowej, toteż wartość SP zmienia się o dwa.

Niestety używanie stosu wiąże się często z wieloma błędami. Jeśli na stos zapisywane jest kilka rejestrów, to muszą one potem zostać odczytane w odwrotnej kolejności. Inaczej w rejestrach znajdą się niewłaściwe wartości. Innym częstym błędem jest pobranie zbyt dużej lub zbyt małej ilości danych ze stosu, a następnie wykonanie rozkazu powrotu z podprogramu. To powoduje, że program zaczyna wykonywać się od nieokreślonego adresu. W najlepszym wypadku powoduje to zawieszenie się komputera, w najgorszym może spowodować skok do procedury formatującej twardy dysk.

Stos może być używany do przesyłania danych pomiędzy rejestrami. Na przykład, nie istnieje instrukcja przesłania danej między rejestrami segmentowymi. Problem ten można pokonać wykorzystując stos:

```
.
.
.
push    ds     ; wyślij rejestr DS na stos
pop     es     ; pobierz tą wartość do rejestru ES
.
.
.
```

Program może zmieniać położenie stosu, modyfikując zawartość rejestrów SS i SP. Należy pamiętać, że nie wolno dokonywać żadnych operacji na stosie, dopóki oba rejestry nie przyjmą nowych wartości. Lecz mimo to nadal istnieje problem, gdy między instrukcjami, zmieniającymi ich zawartość, pojawi się przerwanie. Aby temu zapobiec procesor ignoruje przerwania pojawiające się za instrukcją, która zmienia zawartość rejestru SS. Następny rozkaz powinien więc uaktualnić rejestr SP:

```
mov  ax, SEG NewStackSeg ; Pobierz do AX nowy adres
                         ; segmentu stosu
mov  ss, ax              ; Załaduj rejestr SS
mov  sp, 1024            ; Ustaw wskaźnik stosu. Podczas
                         ; wykonywania tej instrukcji
                         ; przerwania są ignorowane, gdyż
                         ; poprzedni rozkaz zmieniał
                         ; zawartość SS
```

Przerwania

O wielu zdarzeniach, które zachodzą w systemie, procesor dowiaduje się za pośrednictwem przerwań. Na przykład, gdy jest wciskany klawisz, to układ sterujący klawiaturą generuje przerwanie i dzięki temu procesor wie, że musi z niej odczytać daną. Po pojawieniu się przerwania procesor kończy wykonywany rozkaz. Następnie odkłada na stos rejestry znaczników, CS oraz IP i przekazuje sterowanie do procedury obsługi danego przerwania.

Podprogramy te muszą być zakończone rozkazem IRET, który umożliwi powrót do przerwanego programu, przez odtworzenie rejestrów IP, CS i znaczników.
Adresy procedur obsługi przerwań umieszczone są w tablicy wektorów przerwań. Jest to specjalnie wydzielony fragment pamięci operacyjnej od adresu 0:0 do maksimum 40:0. Pojedynczy element tej tablicy składa się z przemieszczenia i numeru segmentu, które są umieszczone w pamięci w tej właśnie kolejności i zajmują 4 bajty. Każde wciśnięcie i zwolnienie klawisza generuje przerwanie numer 9. Adres obsługi tego przerwania znajduje się więc w miejscu 0:24h (9*4).
Programy mogą definiować własne procedury obsługi przerwań. Jednak przy wymianie adresów w wektorze przerwań trzeba uważać, by w tym czasie nie pojawiło się przerwanie. Z tego powodu lepiej jest, by operacją tą zajął się system operacyjny. Funkcja systemowa 25h ustawia, a 35h pobiera adres procedury obsługi danego przerwania. (Więcej informacji o przerwaniach można znaleźć w rozdziale 14.)

MS-DOS

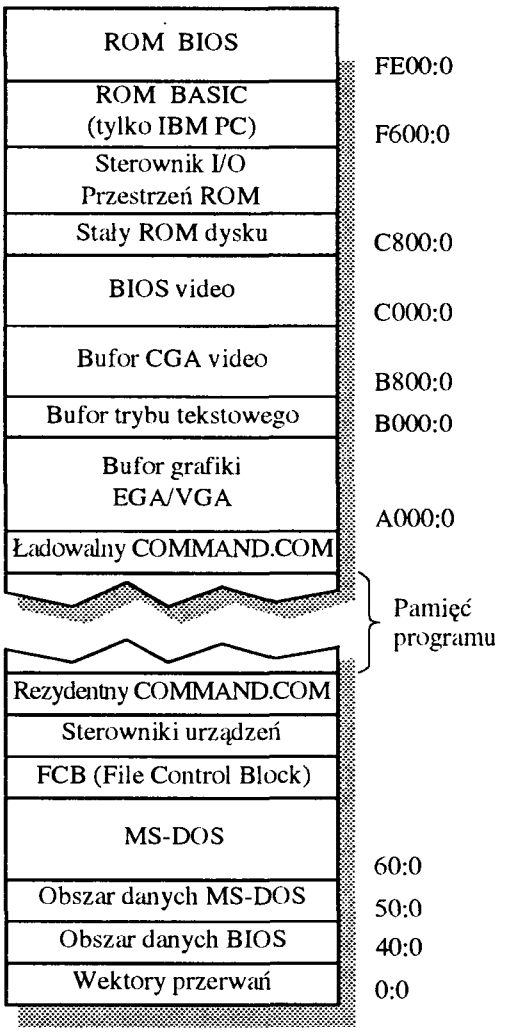
MS-DOS jest jednozadaniowym systemem operacyjnym przeznaczonym dla mikrokomputerów opartych na procesorach 80x86. Większość maszyn pracujących pod tym systemem jest kompatybilna z IBM PC. Ale istnieją i takie, których jedyną częścią wspólną z IBM PC jest typ użytego procesora.
MS-DOS wykonuje wiele użytecznych funkcji, które w znacznym stopniu są zależnych od sprzętu. Ładuje i uruchamia programy, zarządza pamięcią operacyjną i systemem plików, komunikuje się z urządzeniami zewnętrznymi itp. Bez tego nawet prosty program stawałby się wielkim projektem. Wyobraź sobie, jak trudne byłoby pisanie baz danych, gdyby MS-DOS nie obsługiwał plików.

Z czego składa się MS-DOS

MS-DOS podzielony jest na kilka podsystemów. Rysunek 1-3 przedstawia mapę pamięci systemu. Składa się on z czterech głównych części. Jedną z nich jest procesor komend, który jest umieszczony w pliku COMMAND.COM. Jest on odpowiedzialny jest za komunikację z użytkownikiem. Pobiera on od niego komendy, interpretuje je i wykonuje.

SYSTEM PLIKÓW

Jednym z zadań podprogramów, zawartych w tej części, jest kontrola rozmieszczenia plików na dysku. Rysunek 1-4 pokazuje format dysku w systemie MS-DOS. Cały dysk jest podzielony na wiele grup (ang. clusters). Każda grupa składa się z sektorów, które dzielą się już tylko na poszczególne bajty danych. W grupie można umieścić pięć rodzajów danych. Jednym z nich są dane, które inicjują system. Znajdują się one zawsze w pierwszej grupie na dysku. Zawierają konfigurację systemu i program ładujący, który ładuje cały system do pamięci. Procedura, zawarta w ROM BIOS, czyta je podczas restartu systemu (np. po włączeniu komputera).
W grupie może być też zawarta tablica rozmieszczenia plików (FAT). Zawiera ona informację o każdej grupie na dysku. W połączeniu z danymi zawartymi w katalogu pozwala to na stwierdzenie, do jakiego pliku należy dana grupa.



Rysunek 1-3. ***Mapa pamięci DOS.***

Trzeci typ grupy zawiera katalog. Jest to po prostu baza danych o plikach i podkatalogach w nim się znajdujących. Rysunek 1-5 przedstawia format katalogu i objaśnia, jak na podstawie tablicy FAT można ustalić położenie pliku.

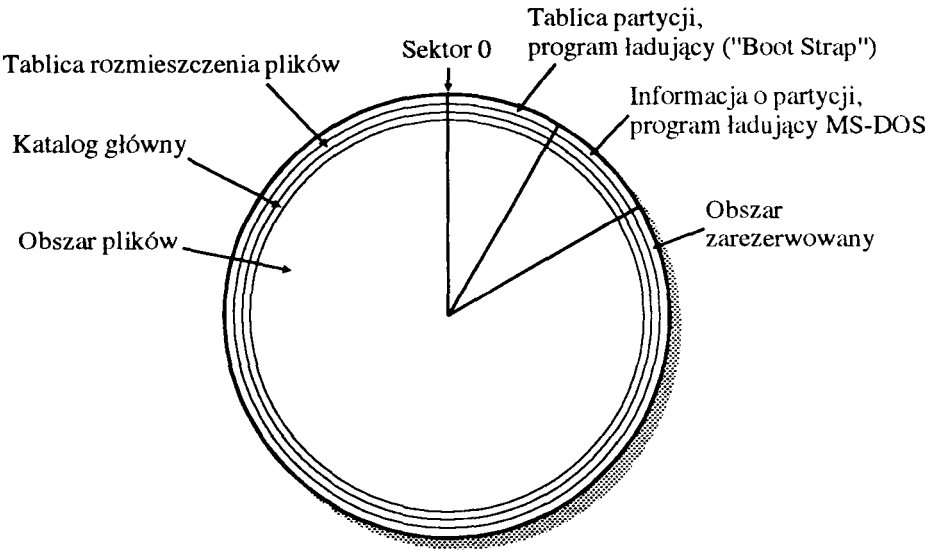
Właśnie pliki są czwartym rodzajem informacji przechowywanej w grupie. Grupa może należeć tylko do jednego pliku. Jeśli jest on mniejszy niż jej rozmiar, reszta miejsca pozostaje nie wykorzystana. Tak więc, gdy na dysku jest dużo małych plików, to może już zabraknąć wolnych grup, mimo że istnieje na nim wiele niewykorzystanej przestrzeni. W tej sytuacji najlepiej jest przeformatować dysk, dzieląc go na mniejsze grupy.

Ostatnim typem grupy jest wolna przestrzeń dyskowa. Grupy te są dostępne przy tworzeniu czy rozszerzaniu plików i katalogów. Grupę nieprzydzieloną system oznacza wpisując 0

w odpowiednie miejsce tablicy FAT. Gdy MS-DOS usuwa lub skraca plik, modyfikuje tylko tablicę FAT, tak by zwolnione grupy były zaznaczone jako dostępne. Natomiast dane pliku pozostają na dysku. O ile więc grupa nie zostanie przypisana innemu plikowi, możliwe jest odzyskanie utraconego zbioru. Wystarczy tylko odpowiednio uaktualnić zapis w katalogu i tablicy FAT.

PROGRAMY OBSŁUGI URZĄDZEŃ

Gdy system plików zapisuje lub czyta dane z dysku, wywołuje program obsługi urządzenia (ang. *device driver*). Programy te pozwalają używać w systemie bardzo różnych rodzajów



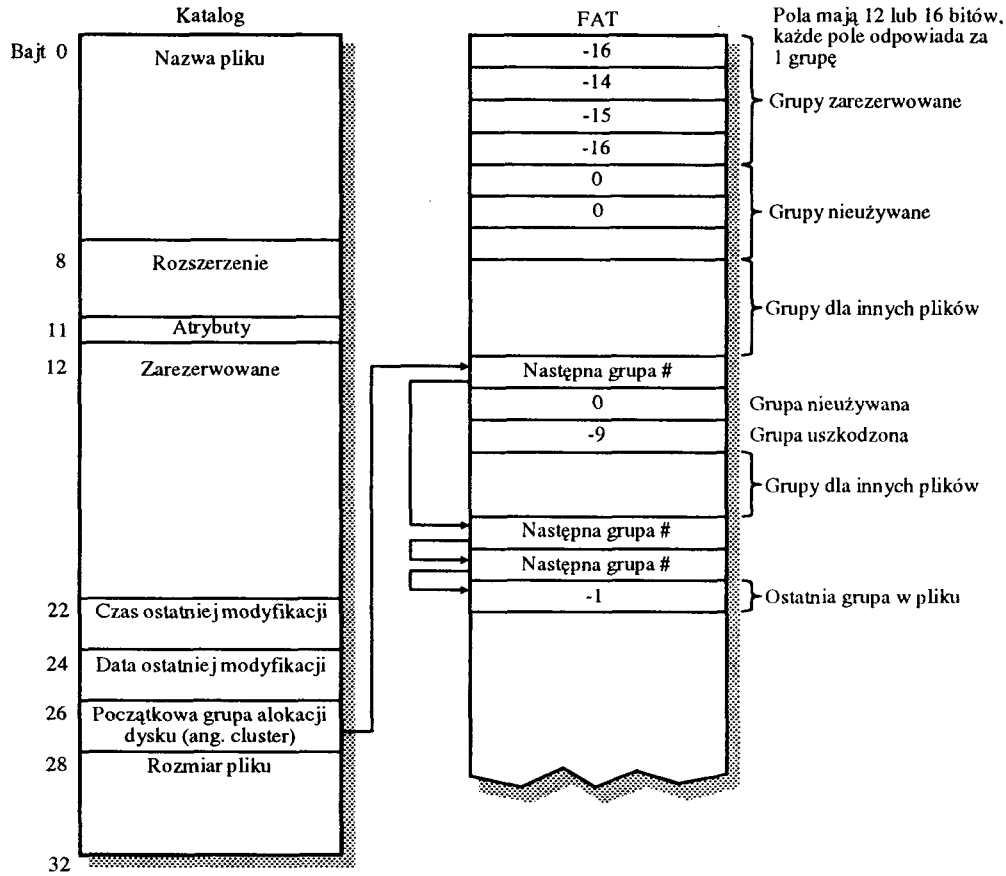
Rysunek 1-4. ***Format dysku w systemie DOS.***

dysków czy innych urządzeń zewnętrznych. Natomiast z punktu widzenia samego systemu wszystkie one są takie same. MS-DOS wysyła komendę np. "odczytaj sektor 12", a dopiero program obsługi podejmuje czynności konieczne by sektor o takim numerze rzeczywiście odczytać.

Niektóre programy obsługi np. dla ekranu czy klawiatury są już wbudowane w MS-DOS. Można je jednak zamienić lub dodać nowe, edytując plik CONFIG.SYS. Plik ten musi znajdować się w głównym katalogu dysku, z którego ładowany jest system. Typowy plik CONFIG.SYS może zawierać takie oto komendy:

```
FILES = 20
BUFFERS = 20
DEVICE = ANSI.SYS
```

MS-DOS analizuje ten plik w czasie startu systemu. W przedstawionym przykładzie pierwsza linia mówi systemowi, że maksymalna liczba jednocześnie otwartych plików nie może



Rysunek 1-5. ***Format katalogu i tablicy FAT***

być większa niż 20. Druga informuje, że MS-DOS dla operacji odczytu i zapisu do plików powinien stworzyć 20 buforów. Ostatnia linia jest komendą, która nakazuje systemowi załadować program obsługi urządzenia o nazwie ANSI.SYS. Zastępuje on wbudowany program do obsługi ekranu i klawiatury.

Należy się upewnić, czy plik CONFIG.SYS pozwala na istnienie w systemie co najmniej 20 otwartych plików, 20 buforów i czy instaluje plik ANSI.SYS. Dwa pierwsze wymogi konieczne są do uruchomienia Turbo Assemblera, zaś programy obsługi, zawarte w pliku ANSI.SYS, wykorzystywane są przez przykładowe programy zamieszczone w tej książce.

ŁĄCZNOŚĆ SYSTEMU Z INNYMI PROGRAMAMI

Ostatni podsystem MS-DOS-u odpowiedzialny jest za współpracę całego systemu z programami użytkownika. MS-DOS oferuje szereg użytecznych funkcji. By móc je wykonać należy przekazać sterowanie do systemu. Możliwe jest to dzięki przerwaniom programowym. Przerwanie programowe jest bardzo podobne do przerwania sprzętowego. Inna jest

tylko jego przyczyna. Nie jest ono bowiem generowane przez zdarzenie zewnętrzne, lecz przez specjalną instrukcję procesora. Procesor 80x86 pozwala, aby w systemie było do 256 różnych przerwań. Pierwsze 16 z nich związane jest z przerwaniami sprzętowymi. MS-DOS używa pięciu z pozostałych 240. Opisuje je tabela 1-3.

Tabela 1-3. ***Przerwania MS-DOS***

Przerwanie	Opis
20h	Zakończenie programu użytkownika
21h	Uniwersalne funkcje systemowe
25h	Fizyczny odczyt kolejnych sektorów
26h	Fizyczny zapis kolejnych sektorów
27h	Zakończenie programu, program zostaje w pamięci jako rezydentny

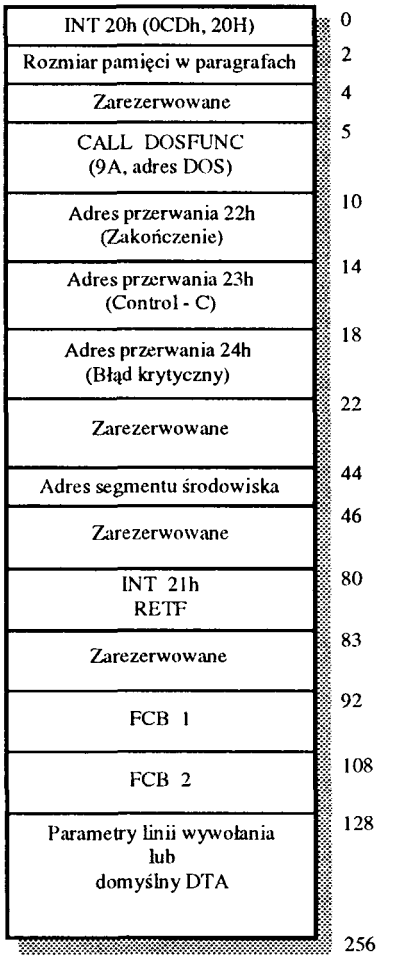
MS-DOS oferuje znacznie więcej niż tylko te pięć czynności. Przerwanie 21h pozwala na dostęp do ponad 75 różnych funkcji. Funkcję wybiera się, podając w rejestrze AH jej numer, a dane wejściowe przekazywane są w pozostałych rejestrach. (Opis funkcji systemowych zamieszczony jest w Dodatku C.)

MS-DOS a programy użytkownika

Jeżeli każemy systemowi wykonać pewien program, to wówczas uruchamiamy cały łańcuch zdarzeń. Najpierw program musi zostać odnaleziony na dysku. Wszystkie programy są przechowywane w postaci plików. Każdy plik ma unikalną nazwę, która go identyfikuje. Składa się ona z czterech części: nazwy dysku, ścieżki, nazwy zasadniczej i rozszerzenia nazwy. Najprostszym sposobem uruchomienia programu jest podanie tylko jego nazwy zasadniczej. Wtedy system sam uzupełnia brakujące elementy. Jednak tylko pliki z rozszerzeniami .EXE, .COM i .BAT mogą być tak uruchomione. Pliki .EXE i .COM zawierają kod dla procesora 80x86, natomiast pliki typu .BAT są plikami tekstowymi interpretowanymi przez procesor komend. MS-DOS, przeszukując katalogi, próbuje odnaleźć plik z dowolnym z tych trzech rozszerzeń. Przeszukiwanie rozpoczyna się od katalogu bieżącego. Jest to ten katalog, którego zawartość jest widoczna po wydaniu komendy **DIR**. Jeśli pliku tam nie ma, przeszukiwane są katalogi wymienione w zmiennej środowiskowej PATH. (Można je obejrzeć wydając komendę **PATH**). Jeśli plik i tutaj nie zostanie odnaleziony, to MS-DOS wyświetli komunikat *"Bad command or file name"* (błędna komenda lub nazwa pliku).

Jeśli przeszukiwanie zakończy się powodzeniem, wówczas następnym krokiem jest przydział programowi odpowiedniej ilości pamięci. W rzeczywistości przydzielane są dwa obszary: jeden na kopię zmiennych środowiskowych, drugi dla samego programu. Zmienne środowiskowe zawierają dane ustawione przez komendy PROMPT, PATH i SET. Wiele programów korzysta z nich przy ustalaniu konfiguracji systemu.

W programach typu .EXE zawarte są dwie liczby, które określają ilość żądanej pamięci. Są one wyznaczane przez program łączący (ang. *linker*). Jedna z nich określa minimalny rozmiar pamięci potrzebny do uruchomienia programu. Jeśli w systemie nie ma tyle wolnej pamięci, to MS-DOS informuje o tym komunikatem *"Insufficient memory"* (brak pamięci)



Rysunek 1-6. ***Struktura nagłówka PSP***

i zwraca sterowanie do procesora komend. W przeciwnym wypadku sprawdza drugą z liczb, by wyznaczyć maksymalny żądany rozmiar pamięci. Jeśli liczba jest równa zero lub jest większa niż ilość dostępnej pamięci, to MS-DOS przydziela całą wolną pamięć. W trakcie swojego działania program, wykorzystując funkcje systemowe, może modyfikować rozmiar przydzielonej mu pamięci.

Następnie tworzony jest nagłówek programu (PSP). Jest to blok pamięci, który zawiera dane o załadowanym programie. Rysunek 1-6 pokazuje format tego nagłówka. Najczęściej wykorzystuje się go do odczytania parametrów wywołania programu oraz przy ustalaniu wartości zmiennych środowiskowych.

Dopiero teraz MS-DOS jest gotowy do wprowadzenia programu do pamięci. Przy łączeniu programu nie było wiadomo, pod jaki adres zostanie on załadowany. Dlatego linker umieszcza w pliku .EXE informacje o tym, które miejsca w programie zawierają odwołania

do adresów segmentów. Program ładujący wykorzystuje je do zaktualizowania tych odwołań. Natomiast przesunięcia są liczone względem początku segmentu. Nie wymagają więc żadnych modyfikacji.
Następnie MS-DOS inicjuje niektóre rejestry. Rejestry SS i SP są ustawiane tak, by wskazywały koniec segmentu stosu, a DS i ES na przedrostek programu PSP. Na koniec MS-DOS ładuje rejestry CS i IP. Powoduje to uruchomienie załadowanego programu.

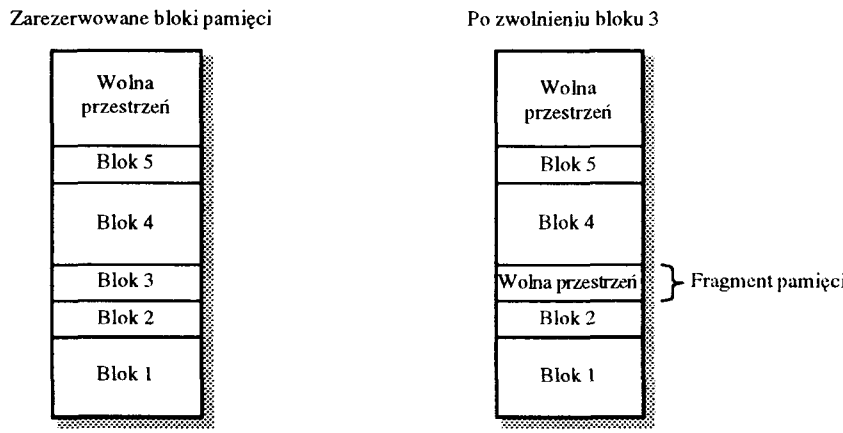
Zarządzanie pamięcią

MS-DOS udostępnia funkcje do zarządzania pamięcią operacyjną: przydzielanie, zwalnianie, zmiana rozmiaru przydzielonej pamięci. Większości aplikacji, podczas ich uruchamiania, zostaje przydzielona cała dostępna pamięć. Wszystko jest w porządku do momentu, w którym dany program chce uruchomić inny. Musi wtedy zwolnić dla niego wystarczający obszar pamięci. Może to zrobić poprzez zmianę rozmiaru przydzielonej dla siebie pamięci. Następujący przykład pokazuje, jak to zrobić:

```
        .MODEL SMALL
        .CODE
Start PROC
        mov  bx, ss       ; Pobierz położenie segmentu
                          ; stosu (ostatni segment
                          ; programu)
        mov  ax, cs       ; Pobierz położenie segmentu
                          ; kodu (pierwszy segment
                          ; programu)
        sub  bx, ax       ; Odejmij położenie początkowe
                          ; od końcowego, wyznaczając
                          ; rozmiar programu (w BX)
        add  bx, 1024/16  ; Dodaj, wyrażony w 16-bajtowych
                          ; paragrafach, rozmiar segmentu
                          ; stosu
        mov  ah, 4Ah      ; Zapisz do AH numer funkcji,
                          ; która zmienia rozmiar
                          ; przydzielonej pamięci
        int  21h          ; Ponieważ położenie bloku
                          ; pamięci jest już w ES,
                          ; wywołaj MS-DOS
        mov  ah, 4Ch      ; Załaduj do AH kod funkcji,
                          ; która kończy program
        int 21h           ; Wywołaj MS-DOS
Start ENDP
        .STACK 1024
END Start
```

Pierwsze trzy linie przykładu są informacją dla asemblera o programie (Rozdział 2 zawiera więcej informacji o formacie programów w asemblerze). Do obliczenia rozmiaru programu wykorzystany został fakt, że segment kodu znajduje się na jego początku, zaś segment stosu na końcu. Rozmiar ten powinien być podany w 16-bajtowych paragrafach. Jest to wygodne,

ponieważ wyrażane są w nich również adresy segmentów. Rejestr ES musi zwierać adres segmentu pamięci, którego rozmiar ma być zmieniony. MS-DOS podczas ładowania programu ustawia ten rejestr tak, by wskazywał na jego początek.
Niektóre programy często dokonują operacji przydziału i zwalniania pamięci. Na przykład, programowi, który odczytuje dane z pliku potrzebne jest miejsce na ich umieszczenie. Gdy informacja zostanie już przetworzona, to bufor nie jest już dłużej potrzebny i może być zwolniony. Częste wykonywanie tych operacji może spowodować fragmentację pamięci. Operacja przydziału może się wtedy nie udać. Jeśli zwalniane obszary pamięci nie sąsiadują ze sobą, to może się zdarzyć, że każdy nie przydzielony blok ma rozmiar mniejszy, niż żądana przez nas liczba bajtów. Problem ten ilustruje rysunek 1-7.



Rysunek 1-7. ***Fragmentacja pamięci***

Rozwiązaniem tego problemu może być wykorzystywanie tak małej ilości bloków, jak to tylko jest możliwe. Programy często ograniczają się do tylko jednego. Gdy aplikacja potrzebuje więcej miejsca, to rozszerza blok. Zwalniając miejsce jednocześnie zmniejsza rozmiar używanego bloku. Technika ta zapobiega fragmentacji, gdyż nigdy nie zostawia wolnego obszaru pomiędzy dwoma przydzielonymi blokami.

Urządzenia zewnętrzne

MS-DOS obsługując urządzenia zewnętrzne, zapewnia do nich jednolity dostęp. Urządzenia w systemie MS-DOS dzielą się na dwa typy. Jeden typ to urządzenia blokowe, takie jak dyski, które mają strukturę wewnętrzną: katalogi i pliki. Drugi to urządzenia znakowe, takie jak drukarki i terminale, które mogą wysyłać i pobierać najwyżej po jednym znaku. Gdy program wysyła komendę do takiego urządzenia, to MS-DOS bezpośrednio kieruje ją do odpowiedniej procedury obsługi. Natomiast polecenie, wysłane do urządzenia blokowego, jest najpierw interpretowane przez podprogramy systemu plików. Dopiero one wywołują odpowiednie programy obsługi.
Każde urządzenie ma swoją nazwę. Nazwy urządzeń blokowych są jednoliterowe. Lecz pisząc lub odczytując dane, program musi podać nie tylko nazwę takiego urządzenia, ale

również nazwę pliku, którego operacja ma dotyczyć. Natomiast nazwy urządzeń znakowych mogą być ośmioliterowe. Odwołując się do tego typu urządzenia, wystarczy tylko podać systemowi nazwę urządzenia.
By uniknąć podawania nazw urządzeń za każdym odwołaniem się do nich, MS-DOS przypisuje plikom i urządzeniom uchwyty (ang. handle). Operacji tej dokonuje funkcja OPEN FILE (otwórz plik) o numerze 3Dh. Od tej pory program, odwołując się do urządzenia, może podawać tylko jego identyfikator. Natomiast po zakończeniu wszystkich operacji należy wywołać funkcję CLOSE FILE (zamknij plik - 3Eh). (Rozdział 13 zawiera więcej informacji o korzystaniu z plików).
MS-DOS zawiera już kilka wbudowanych procedur obsługi urządzeń. Przedstawione są one w tabeli 1-4. Wydając komendę **DEVICE** w pliku CONFIG.SYS można dodawać i zamieniać istniejące już urządzenia.

Tabela 1-4. ***Urządzenia w systemie MS-DOS***

Urządzenie	Opis
CON:	Konsola i klawiatura
AUX:	Dodatkowe urządzenie we/wy. Zwykle COM1:
PRN:	Standardowa drukarka. Zwykle LPT1:
COM1:	Pierwszy port szeregowy
COM2:	Drugi port szeregowy
LPT1:	Pierwszy port równoległy
LPT2:	Drugi port równoległy

MS-DOS udostępnia wiele funkcji do obsługi niektórych urządzeń. Na przykład, istnieje osiem funkcji, które współpracują z klawiaturą i konsolą. Następne trzy pozwalają na dostęp do portów szeregowych (funkcje 3 i 4) i równoległych (funkcja 5). Funkcje, związane z klawiaturą, pozwalają na odczyt zarówno pojedynczego znaku, jak i całego ich ciągu (funkcja 0Ah). Odczytując pojedynczy klawisz można określić, czy ma on być wyświetlony na ekranie (funkcja 1), czy nie (funkcje 7 i 8). Funkcja 0Bh pozwala sprawdzić stan klawiatury, a funkcja 0Ch opróżnia jej bufor. Na ekranie można wyświetlić pojedynczy znak (funkcja 2) lub ciąg znaków (funkcja 9). Ostatnia ósma funkcja pozwala na zapis lub odczyt z konsoli zależnie od zawartości rejestru DL.

Podsumowanie

Środowisko, w którym program ma pracować, jest bardzo ważnym elementem dla projektanta oprogramowania. Różne systemy operacyjne i procesory wymagają innych sposobów programowania. Programy, pisane w Turbo Assemblerze, przeznaczone są do pracy pod kontrolą systemu operacyjnego MS-DOS. System ten zawiera wiele użytecznych funkcji, które dają wygodny dostęp zarówno do plików, jak i do urządzeń wejścia/wyjścia. Poza tym system zarządza pamięcią operacyjną i zapobiega niepożądanemu oddziaływaniu na siebie kilku programów.

Programy w systemie MS-DOS pracują na procesorach 80x86. Turbo Assembler zawiera wszystkie ich rozkazy. Instrukcje te pozwalają na operowanie danymi, które znajdują się w rejestrach i w pamięci, a także na stosie. Stos wykorzystuje się do przechowywania zawartości rejestrów, w czasie gdy one same są używane do innych celów. Zapewnia on także poprawne wywoływanie podprogramów.
Programy użytkownika komunikują się z MS-DOS-em za pośrednictwem przerwań programowych. Są one bardzo podobne do wywołań podprogramów, z tym, że przy ich użyciu nie podaje się bezpośrednio adresu procedury, a numer przerwania. Dopiero procesor na jego podstawie wyznacza ten adres.

BIBLIOTEKA GŁ. AGH

Rozdział 2 *Turbo Assembler*

Pakiet Turbo Assemblera zwiera wszystkie narzędzia niezbędne do tworzenia działającycl programów. Rozdział ten opisuje jak je wykorzystywać przy tworzeniu plików źródłowych ich asemblacji, łączeniu i uruchamianiu. Przedstawiony zostanie również program MAKE - bardzo wygodne narzędzie przy pracy z wieloma plikami źródłowymi.
W następnej części rozdziału opisany jest sposób pracy asemblera. Przedstawione zostan jego instrukcje i dyrektywy czyli komendy, które sterują pracą asemblera.
Ostatnia część to opis instrukcji procesora. Instrukcje te są bezpośrednio wykonywane prze procesor. Przedstawione zostanie kilka przykładów, które pokazują ich działanie i wyjaś niają tryby adresowania procesora 80x86.

Instalacja

Aby posłużyć się Turbo Assemblerem należy go najpierw zainstalować, czyli wprowadzi na dysk wszystkie niezbędne zbiory. Najprostszym sposobem jest uruchomienie program *install*, który znajduje się na dyskietce z Turbo Assemblerem. Program ten przekopiuj zbiory do podkatalogu o standardowej nazwie \TASM. Można mu jednak podać inn dowolną nazwę katalogu. Po zainstalowaniu należy sprawdzić wartość zmiennej PATH przez wydanie MS-DOS-owi polecenia **PATH**. Jednym z wyświetlonych katalogów powi nien być \TASM (lub też ten o podanej nazwie). Powinny znajdować się w nim międz innymi pliki TASM.EXE, TLINK.EXE i MAKE.EXE.
Można je przekopiować również samodzielnie, lecz program instalacyjny ponadto rozpa kuje pliki z przykładami oraz innymi użytecznymi programami jak Turbo Debugger i Turb Profiler. Nie są one konieczne przy pisaniu asemblerowych programów, ale, na przykład Turbo Debugger znakomicie nadaje się do wyszukiwania błędów w programach. Jeg dokładniejszy opis znajduje się w rozdziale 3.
W pakiecie znajduje się wiele przykładowych programów. Ilustrują one ważniejsze cech programowania w asemblerze. Na dyskietkach zapisane są one w zbiorach z rozszerzenien .ZIP. W pojedynczym zbiorze tego typu znajduje się kilka spakowanych plików źródłowych. Program UNZIP przekształca te pliki na zwykłe pliki tekstowe, które następni można już np. obejrzeć. Każdy spakowany zbiór należy umieścić w oddzielnym katalogu Następnie należy wydać polecenie `UNZIP nazwa_pliku`, a to spowoduje rozpakowani podanego zbioru.
Jedynym programem, nie dołączanym wraz z Turbo Assemblerem, jest edytor tekstowy. N szczęście istnieje wiele wysokiej jakości edytorów. Warto upewnić się, czy zmienna PAT zawiera katalog, w którym znajduje się ten edytor. Często komenda PATH umieszczona jes w pliku AUTOEXEC.BAT. Wykonywana jest więc przy każdym restarcie systemu. Jeśl np. pliki Turbo Assemblera znajdują się w katalogu C:\TASM, edytor w C:\EDIT, a progra my usługowe MS-DOS-a w C:\DOS, to możesz użyć następującej komendy:

```
PATH C:\TASM;C:\EDIT;C:\DOS
```

Tworzenie plików źródłowych

Najlepszym sposobem na zapoznanie się z Turbo Assemblerem jest napisanie rzeczywistego programu. Poniższy przykład wyświetla na ekranie tekst *"My First Assembler Program"*.

Wydruk 2-1. ***Prosty program asemblerowy***

```
; FIRST.ASM  Prosty program w asemblerze
        .MODEL SMALL          ; Rozkaz dla asemblera: Użyj
                              ; małego modelu pamięci
Prnstr  EQU  9                ; Symbol Prnstr oznacza
                              ; liczbę 9
ExitPrg EQU  4Ch              ; Symbol ExitPrg oznacza
                              ; liczbę 4Ch

        .DATA
; Następna linia powoduje umieszczenie w pamięci ciągu
; znaków pod adresem identyfikowanym przez Mesg. Liczby
; 0Dh i 0Ah na jego końcu oznaczają znaki powrotu do
; początku i przejścia do następnej linii. Znak dolara
; jest używany przez MS-DOS dla zaznaczenia końca ciągu
; znaków.
Mesg    DB  "My First Assembler Program", 0Dh, 0Ah, '$'
        .CODE
Start   PROC                  ; Początek procedury o
                              ; nazwie Start
        mov  ax, SEG DGROUP   ; Pobierz adres segmentu
                              ; danych
        mov  ds, ax
        mov  dx, OFFSET Mesg  ; Pobierz do DX adres ciągu
                              ; znaków
        mov  ah, Prnstr       ; w AH numer funkcji, która
                              ; wyświetla ciąg znaków
                              ; na ekranie
        int  21h              ; Wywołanie MS-DOS
        mov  ah, ExitPrg      ; w AH numer funkcji, która
                              ; kończy program
        int  21h              ; Wywołanie MS-DOS
Start   ENDP                  ; Koniec procedury Start
        .STACK 512            ; Stwórz stos o wielkości
                              ; 512 bajtów
        END Start             ; Koniec programu, jego
                              ; wykonywanie będzie się
                              ; rozpoczynać od adresu Start
```

Pierwszym znakiem w pierwszej linii programu jest średnik. Oznacza on, że reszta tekstu, znajdująca się za nim, jest komentarzem. Zadaniem komentarzy jest wyjaśnianie, co poszczególne fragmenty kodu mają robić. Asembler je po prostu ignoruje.
Program w asemblerze rozmieszczony jest w trzech kolumnach. Pierwsza zawiera etykiety i symbole, które pozwalają na łatwiejsze zrozumienie programu. Etykiety oznaczają położenie danych w programie. Na przykład na, wydruku 2-1 etykieta **Mesg** oznacza adres ciągu znaków *"My First Assembler Program"*. Instrukcja **mov dx, OFFSET Mesg** kopiuje go do rejestru DX. Używanie etykiet jest więc bardzo wygodne. Pozwala bowiem na operowanie nazwami a nie prawdziwymi wartościami adresów.
Symbole oznaczają pewne stałe wykorzystywane w programie. Na przykład, wydruk 2-1 zawiera symbole **Prnstr** oraz **ExitPrg**, które oznaczają liczby 9 i 4Ch. Każde wystąpienie w programie symbolu **Prnstr** jest zastępowane przez asembler liczbą 9, a **ExitPrg** liczbą 4Ch. Używanie symboli sprawia, że programy są bardziej przejrzyste i czytelne. Jeżeli widzisz w programie liczbę 9, to nie jesteś pewien, czy chodzi o numer funkcji dla MS-DOS-u, czy może o znak ASCII, czy po prostu o liczbę 9. W książce symbole pisane są jednocześnie dużymi i małymi literami - w odróżnieniu od instrukcji (tylko małe) i dyrektyw (tylko duże).
Druga kolumna zawiera dyrektywy i instrukcje asemblera. Dyrektywy są poleceniami dla asemblera. Mówią mu one, jak należy interpretować tekst programu. Przykładem dyrektywy jest EQU, używana do definiowania symboli. Nie generuje ona żadnego kodu wynikowego. Dzięki niej asembler wie tylko co zrobić, gdy w programie znajdzie uprzednio definiowany symbol.
Instrukcje są natomiast przekształcane na kod zrozumiały dla procesora. Każda instrukcja asemblera odpowiada jednemu rozkazowi procesora. Na przykład, pierwszą instrukcją w programie jest MOV. Po jej rozpoznaniu asembler umieści w pliku wynikowym kod 0B8h.
W trzeciej kolumnie znajdują się *argumenty* dla instrukcji i dyrektyw. Każda z nich ma swój własny typ argumentów i ich liczbę. Instrukcja MOV kopiuje dane pomiędzy jej argumentami: **ax, SEG DGROUP**. Pierwszy z nich określa miejsce, do którego będzie przesłana dana, będąca drugim argumentem.
Program ma 37 linii. Podobny, napisany w BASIC-u Pascalu czy C miałby ich około 5. Oto powód, dla którego większość programów jest pisana właśnie w tych językach. Jednakże program ten w postaci wykonywalnej ma tylko 557 bajtów: 45 to właściwy kod, reszta zaś stanowi miejsce zarezerwowane na stos. Rezultat ten jest trudny do osiągnięcia w językach wysokiego poziomu.

Tworzenie pliku wykonywalnego

By stworzyć plik wykonywalny należy najpierw komendą **tasm first** skompilować tekst program do pliku obiektowego (z rozszerzeniem .OBJ):
Na ekranie powinien pojawić się napis:

```
Turbo Assembler Version 2.0 Copyright 1988, 1990 Borland
International

Assembling file: first.ASM
Error messages: None
```

```
Warning messages: None
Passes: 1
Remaining memory: 266K
```

Gdy w programie są pomyłki, to liczba błędów czy ostrzeżeń może być większa. Informacje o pomyłkach zawierają: numer błędnej linii i rodzaj błędu. Linie te należy sprawdzić i poprawić. Jednak możliwe jest, że pomyłka w jednej linii może spowodować wystąpienie błędu w innej. Na przykład, jeśli zdarzy się literówka w symbolu ExitPrg na początku programu, to błąd pojawi się dopiero na końcu procedury **Start**.
Ostatnim krokiem jest *konsolidacja* programu. Pozwala ona na połączenie kilku modułów wynikowych w jeden działający program. Mimo że w naszym przykładzie jest tylko jeden moduł wynikowy, to konsolidacja jest jednak niezbędna. Istnienie tego etapu pozwala bowiem na uproszczenie asemblera. Generuje on tylko zbiory wynikowe, natomiast stworzeniem programu wykonywalnego zajmuje się konsolidator. Konsolidacji przykładowego programu można dokonać komendą:

```
tlink first
```

Konsolidator powinien wyświetlić napis:

```
Turbo Link Version 3.0 Copyright 1987, 1990 Borland
International
```

Po zakończeniu konsolidacji zostanie utworzony zbiór FIRST.EXE. W dalszej części rozdziału znajduje się więcej informacji o asemblerze i programie łączącym.

Asembler

Najprostszym sposobem skompilowania danego programu źródłowego jest podanie asemblerowi jego nazwy. Tak właśnie zrobiliśmy w ostatnim przykładzie. Jednakże w niektórych przypadkach istnieje potrzeba przekazania asemblerowi pewnych specjalnych instrukcji, dotyczących sposobu tłumaczenia plików. Można na przykład podać opcje, z jakimi mają być kompilowane zbiory lub listę zbiorów do przetłumaczenia.

OPCJE

Opcje wybiera się, podając razem z komendą TASM tzw. przełączniki. Przedstawia je tabela 2-1. Może się okazać, że część z nich jest używana niemal zawsze. Na przykład, pisząc procedurę, która będzie wykorzystana w języku wysokiego poziomu, należy użyć przełącznika /ml, który sprawi, że asembler będzie rozróżniał w symbolach i etykietach małe i duże litery. By uniknąć każdorazowego wpisywania często używanych opcji, można umieścić je w pliku konfiguracyjnym TASM.CFG. Powinien on znajdować się w tym samym katalogu co Turbo Assembler. Asembler przed podjęciem kompilacji poszukuje tego pliku i odczytuje z niego opcje. Typowy plik TASM.CFG może wyglądać następująco:

```
/iC:\TASM\INCLUDE
/ml
/zi
```

Pierwsza opcja określa katalog, w którym będą poszukiwane zbiory nagłówkowe. Zbiory te zawierają częściej używane makrodefinicje (rozdział 10) i definicje symboli. Dobrze jest

umieścić wszystkie te pliki w jednym katalogu, a następnie, wykorzystując opcję /i, przekazać jego nazwę asemblerowi.
Drugi przełącznik sprawia, że we wszystkich symbolach rozróżniane są małe i duże litery. Bez niego symbole **ThisSym** oraz **THISSYM** byłyby utożsamione. Opcja ta jest konieczna do konsolidacji programów asemblerowych z programami, napisanymi w językach wysokiego poziomu (rozdział 12).
Ostatni przełącznik nakazuje asemblerowi umieszczenie w zbiorze wynikowym kompletnej informacji o numerach wierszy i symbolach modułu źródłowego. Informacja ta wykorzystywana jest przez Turbo Debugger.

NAZWY PLIKÓW

Nazwy plików do kompilacji podaje się tuż za komendą TASM. Mogą to być dowolne zbiory tekstowe. Jeśli nie zostanie podane rozszerzenie, to asembler przyjmuje je jako .ASM. Można również podać znaki wieloznaczne, by skompilować kilka plików naraz:

```
tasm s*
```

Komenda ta powoduje tłumaczenie wszystkich plików, które zaczynają się od litery "s" i mają rozszerzenie .ASM oraz znajdują się w bieżącym katalogu.

Tabela 2-1. ***Opcje Turbo Assemblera***

Opcja	Opis
/a	Polecenie alfabetycznego uporządkowania segmentów
/c	Umieszczenie w raporcie kompilacji podraportu odwołań skrośnych.
/dSYMBOL[=VAL]	Zdefiniowanie symbolu jako 0 lub VAL
/e	Polecenie generowania rozkazów emulujących operacje zmiennoprzecinkowe
/h,/?	Polecenie wyświetlenia wykazu opcji
/iPATH	Określenie nazwy katalogu zbiorów włączanych jako PATH
/jDIRECTIVE	Określenie dyrektywy, która będzie wykonana przed podjęciem kompilacji
/khNUMBER	Kompilacja programu, który zawiera podaną liczbę symboli
/ksNUMBER	Kompilacja programu, który wymaga podanego obszaru dla ciągów znaków
/l	Polecenie generowania raportu kompilacji
/la	Polecenie generowania rozszerzonego raportu kompilacji
/ml	Polecenie rozróżniania małych i dużych liter
/mNUMBER	Polecenie skompilowania programu w podanej ilości przebiegów
/mu	Polecenie przekształcenia wszystkich małych liter w duże

(kontynuacja)

Opcja	Opis (kontynuacja)
/mvNUMBER	Ustawienie maksymalnej długości symboli
/mx	Polecenie rozróżniania małych i dużych liter, ale tylko w obrębie symboli zewnętrznych
/n	Polecenie wyeliminowania z raportu kompilacji informacji o tablicy symboli
/p	Polecenie sprawdzenia, czy w module źródłowym użyto rozkazów zabronionych w trybie zabezpieczonym
/q	Polecenie usuwania zbędnych rekordów z modułów wynikowych
/s	Polecenie, które powoduje brak porządkowania segmentów
/r	Polecenie generowania rzeczywistych rozkazów arytmetyki zmiennoprzecinkowej
/t	Polecenie zaniechania komunikatu o poprawnej kompilacji
/w[-]WARNING	Polecenie zaniechania generacji podanego ostrzeżenia
/w+WARNING	Polecenie generacji podanego ostrzeżenia
/w0	Polecenie zaniechania generacji wszystkich ostrzeżeń
/w1,/w2	Polecenie generacji ostrzeżeń
/x	Polecenie włączania do raportu kompilacji niekompilowanych gałęzi dyrektyw warunkowych
/zi	Umieszczenie w zbiorze wynikowym kompletnej informacji o numerach wierszy i symbolach modułu źródłowego
/zd	Umieszczenie w zbiorze wynikowym informacji tylko o numerach wierszy modułu źródłowego

Innym sposobem kompilacji kilku zbiorów jest podanie ich nazw rozdzielonych średnikami. Każdy z tych plików może mieć swoje własne opcje:

```
tasm /c zbiór1; /ic:\tasm\include zbiór2
```

Asembler może wygenerować trzy typy zbiorów. Pierwszy z nich to zbiór wynikowy. Zawiera on kod i dane programu. Następnym typem zbioru jest raport kompilacji. Jest to zbiór tekstowy, w którym znajduje się zarówno kod źródłowy, jak i kod wygenerowany przez asembler. Trzecim typem jest raport odwołań skrośnych. Jest to również plik tekstowy. Zawiera on listę wszystkich symboli i etykiet oraz informację o miejscach ich występowania. Standardowo ich nazwa główna pokrywa się z nazwą pliku źródłowego, a rozszerzenia są następujące: odpowiednio .OBJ, .LST i .CRF. Jeśli muszą one mieć inne nazwy, to należy rozdzielić je przecinkami. Na przykład, można w ten sposób skompilować kilka wersji tego samego pliku źródłowego do różnych modułów wynikowych i stworzyć różne raporty kompilacji:

```
tasm /dVersion=1 /l /c PROG.ASM PROG1.OBJ,PROG1.LST,PROG1.CRF
tasm /dVersion=2 /l /c PROG.ASM PROG2.OBJ,PROG2.LST,PROG2.CRF
```

```
tasm /dVersion=3 /l /c PROG.ASM PROG3.OBJ,PROG3.LST,PROG3.CRF
```

Przykład ten tworzy trzy różne wersje programu PROG.ASM. Pierwsza opcja nakazuje asemblerowi zdefiniowanie symbolu **Version** z inną wartością dla każdej wersji programu. Przełącznik /l poleca utworzenie raportu kompilacji, a /c raportu odwołań skrośnych.

Konsolidator

Moduł wynikowy zawiera kod i dane, wygenerowane przez asembler podczas procesu tłumaczenia zbioru źródłowego. Zawiera on także globalne symbole, wykorzystywane przez kilka modułów. Zadaniem konsolidatora jest przekształcenie plików wynikowych w program wykonywalny.
Nazwy modułów wynikowych, niezbędnych do utworzenia programu wykonywalnego, mogą być rozdzielone spacjami lub znakami plus:

```
tlink first second third
```

lub

```
tlink first+second+third
```

Każdy z wymienionych plików zawiera pewną część tworzonego programu. W rozdziale 10 znajduje się opis dyrektyw, które pozwalają na sterowanie wzajemnym położeniem tych części.

OPCJE KONSOLIDATORA

Tabela 2-2. ***Opcje konsolidatora***

Opcja	Opis
/c	Polecenie rozróżniania małych i dużych liter w symbolach zewnętrznych
/d	Polecenie informowania o wystąpieniu tego samego symbolu w kilku bibliotekach
/i	Polecenie inicjacji segmentów
/l	Polecenie włączenia do zbioru, który zawiera program wykonywalny, informacji uruchomieniowych
/m	Polecenie uzupełnienia mapy konsolidacji wykazem symboli publicznych
/n	Polecenie zignorowania bibliotek domniemanych
/s	Polecenie uzupełnienia mapy konsolidacji wykazem symboli publicznych oraz mapą segmentów
/x	Polecenie wymusza brak generacji mapy konsolidacji

Dla większości programów nie trzeba ustawiać żadnego z tych przełączników. Jedynie w programach, które zawierają moduły wynikowe powstałe przy użyciu języków wysokiego poziomu należy wykorzystać opcję /c.

Turbo Debugger wymaga, aby w programie wykonywalnym istniała informacja uruchomieniowa. Zbiory źródłowe muszą więc być skompilowane z opcją /zi asemblera, a konsolidator uruchomiony z opcją /l.
Jeśli program składa się tylko z jednego pliku źródłowego, to możesz przejrzeć raport kompilacji, po to by zobaczyć, jak zostały rozmieszczone jego poszczególne fragmenty. Przy wielu plikach nie jest to już takie proste. Raport dla każdego pliku jest bowiem tworzony tak, jakby program składał się tylko z niego. Potrzebne jest więc globalne spojrzenie na to, jak poszczególne moduły są rozmieszczone. Informację o tym dostarcza mapa konsolidacji, tworzona przez program łączący.
Jeśli nie zostanie użyty żaden z przełączników /m, /s czy /x to konsolidator stworzy mapę konsolidacji, która zawiera nazwy i położenia segmentów oraz adres początku programu. Opcja /m nakazuje dodanie jeszcze wszystkich symboli publicznych. Symbole te są widoczne we wszystkich modułach programu. Opcja /s zapewnia umieszczenie rozszerzonej informacji o segmentach. Natomiast opcja /x powoduje zaniechanie generowania mapy konsolidacji. Może to przyśpieszyć proces łączenia na komputerach, wyposażonych w wolne dyski.

NAZWY PLIKÓW KONSOLIDATORA

Nazwa programu wykonywalnego oraz pliku z mapą konsolidacji jest nazwą pierwszego z modułów wynikowych. Konsolidator przyjmuje dla nich rozszerzenia .EXE i .MAP. Oczywiście nazwę każdego z tych zbiorów można podać również samemu. Muszą one być rozdzielone przecinkami:

```
tlink first second third,myprog,mapfile
```

Komenda ta powoduje konsolidację plików FIRST.OBJ, SECOND.OBJ i THIRD.OBJ do pliku MYPROG.EXE oraz powoduje utworzenie mapy konsolidacji o nazwie MAPFILE.MAP
Istnieje jeszcze jeden typ zbioru, który można wykorzystać w procesie łączenia. Są to biblioteki. Zawierają one wiele modułów wynikowych, w których znajdują się częściej stosowane podprogramy. Do konsolidacji z modułami, powstałymi w językach wysokiego poziomu, będzie prawdopodobnie będzie konieczne dołączenie kilku bibliotek. Na przykład, by skonsolidować asemblerowy podprogram ASMSTUFF z programem napisanym w Turbo C o nazwie CPROG, należy użyć komendy:

```
tlink /c /m C0S CPROG ASMSTUFF,CPROG,CMAP,CS EMU
```

Opcja /c sprawia, że konsolidator rozróżnia małe i duże litery. Przełącznik /m powoduje generowanie mapy konsolidacji. Pierwsze trzy nazwy (C0S, CPROG, ASMSTUFF) są modułami, które mają być połączone. Plik C0S jest modułem startowym, niezbędnym w programach napisanych w Turbo C. Nazwa programu wykonywalnego (CPROG) znajduje się za pierwszym przecinkiem. Następna nazwa (CMAP) to nazwa pliku z mapą konsolidacji. Natomiast ostatnie dwie to biblioteki: standardowa biblioteka języka C (CS) oraz biblioteka podprogramów emulujących operacje zmiennoprzecinkowe.
Z czasem, gdy program się rozrasta, dodawane są coraz to nowe moduły, wydłuża się więc komenda dla konsolidatora. By uniknąć podawania za każdym razem tak długiego polecenia, można umieścić wszystkie instrukcje dla konsolidatora w pliku, a później podawać tylko jego nazwę. Nazwę tą należy tylko poprzedzić znakiem "@". Jeśli wszystkie polecenia

dla konsolidatora są umieszczone w zbiorze LINKFILE.LNK, to komenda, która go wywołuje ma postać:

```
tlink @linkfile.lnk
```

Organizator MAKE

Bez specjalnych narzędzi trudno jest zarządzać wielkimi programami, które składają się z dużej liczby modułów. Po dokonaniu każdej zmiany trzeba na nowo skompilować wszystkie zmienione zbiory i skonsolidować wszystkie moduły wynikowe. Jeśli modyfikacja dotyczy np. zmiennej używanej w kilku modułach należy się upewnić, czy wszystkie zostały ponownie przetłumaczone. Pewnym rozwiązaniem jest napisanie pliku wsadowego, który zawsze skompiluje i skonsoliduje wszystkie zbiory. Jest to jednak czasochłonne.
Program MAKE porównuje czasy utworzenia plików źródłowych i wynikowych. Gdy stwierdzi, że zbiór źródłowy jest nowszy, niż odpowiadający mu zbiór wynikowy, to zarządza jego kompilację. Ale, by program MAKE wiedział, kiedy i jak uruchomić asembler, należy mu podać pewne reguły i instrukcje.
Ponieważ działanie programu opiera się na porównywaniu czasów i dat, to przed podjęciem edycji i kompilacji plików źródłowych, należy się upewnić, czy w systemie ustawiony jest aktualny czas. Służą do tego komendy systemowe TIME oraz DATE. By uzyskać informację o czasie utworzenia zbioru można wykorzystać komendę DIR. Natomiast program TOUCH ustawia datę i czas podanych plików na datę i czas bieżący.
Poniższy przykład zawiera komendy dla organizatora, które to komendy spowodują utworzenie programu BIG.EXE z plików źródłowych FIRST.ASM, SECOND.ASM i THIRD.ASM. Można go napisać pod dowolnym edytorem i umieścić w pliku o standardowej nazwie MAKEFILE.

```
big.exe: first.obj second.obj third.obj
     tlink first second third,big
first.obj: first.asm
     tasm first
second.obj: second.asm
     tasm second
third.obj: third.asm
     tasm third
```

Natomiast, by wywołać organizator, należy wydać polecenie

```
make
```

Jeśli BIG.EXE jest aktualny, to nie zostanie podjęta żadna akcja. Przypuśćmy teraz, że został zmieniony plik SECOND.ASM. Program MAKE swoją działalność rozpoczyna od przeczytania pierwszej linii. Pierwsza nazwa w tej linii określa zbiór, który właśnie należy utworzyć (zbiór wynikowy). Natomiast nazwy, występujące za dwukropkiem, określają pliki, od których zależy jego utworzenie (pliki źródłowe). Organizator sprawdza każdy z nich, określając, czy zbiór wynikowy musi być aktualizowany. Najpierw sprawdzany jest zbiór FIRST.OBJ. Program MAKE przegląda dalsze linie, sprawdzając, czy zbiór ten nie jest zbiorem wynikowym w innej instrukcji organizatora. W naszym przykładzie zostaje odnaleziona linia:

```
first.obj: first.asm
```

Mówi ona, że plik FIRST.OBJ zależy od pliku FIRST.ASM. Ponieważ nie ma instrukcji, w której FIRST.ASM byłby plikiem wynikowym, organizator porównuje daty obu tych plików. Okazuje się, że zbiór FIRST.OBJ jest nowszy niż zbiór źródłowy, nie jest więc podejmowana żadna akcja. Organizator wraca znowu do analizy pierwszej linii i porównuje daty plików FIRST.OBJ oraz BIG.EXE. Jeszcze raz plik wynikowy okazuje się być nowszy niż plik źródłowy, w tej sytuacji nic się nie dzieje.
W ten sposób sprawdzane są wszystkie zależności. Okazuje się więc, że należy utworzyć zbiór SECOND.OBJ. Organizator czyta linię, która występuje za instrukcją i która spowodowała podjęcie tej akcji:

```
tasm second
```

Następnie traktuje ją jako polecenie dla systemu operacyjnego. W naszym przykładzie spowoduje to wywołanie asemblera i kompilację zbioru SECOND.ASM. Podczas tego procesu zmieni się również data zbioru wynikowego SECOND.OBJ. Staje się on nowszy niż BIG.EXE. Organizator sprawdza jeszcze tylko zależności zbioru THIRD.OBJ i przystępuje do wykonania komendy:

```
tlink first second third,big
```

Działanie to, tworząc aktualny program, kończy już wszystkie operacje.
Istnieje jeszcze kilka innych typów komend dla organizatora. Linie, które zaczynają się od znaku "#", traktowane są jako komentarz. W poniższym przykładzie komentarze zawierają opis kilku możliwości programu MAKE:

```
# Plik ten tworzy program o nazwie FANCY.EXE. Poniższa
# linia mówi organizatorowi, by wywoływał kompilator
# tasm przy każdej zależności z rozszerzeniem .ASM.
.asm.obj:
     tasm $*
# Dyrektywa $* oznacza nazwę zbioru bez rozszerzenia.
# w następnej linii tworzone jest makro o nazwie OBJS.
# Odwołanie $(OBJS) będzie zastępowane tekstem z tej
# linii.
OBJS=first.obj second.obj third.obj
# Sprawdź, czy pliki .OBJ lub plik dla konsolidatora nie
# są nowsze od programu wynikowego
fancy.exe: $(OBJS) fancy.lnk
     tlink @fancy.lnk
# Sprawdź zbiory źródłowe. Nie ma potrzeby specyfikowania
# komendy, gdyż została ona już podana powyżej w regule
# dla plików .asm.
first.obj: first.asm
second.obj: second.asm
third.obj: third.asm
# Jeśli plik dla organizatora jest nowszy niż plik dla
# konsolidatora stwórz nowy zbiór dla programu łączącego.
```

```
fancy.lnk: makefile
    echo $(OBJS) > fancy.lnk
    echo fancy.exe >> fancy.lnk
```

Jak pracuje asembler

Podstawowym zadaniem Turbo Assemblera jest tłumaczenie plików z programami asemblerowymi do plików wynikowych. Proces ten zaczyna się od analizy pliku źródłowego. Asembler, po odczytaniu dyrektywy, według niej modyfikuje sposób swego działania. W przykładzie 2-2 pierwszą linią jest:

```
.MODEL SMALL
```

Dyrektywa ta ustala rozmiar etykiet na 16 bitów, a wszystkie wywołania i powroty z funkcji traktowane są jako bliskie. Inną dyrektywą jest:

```
.DATA
```

Powoduje ona umieszczanie przez asembler generowanego kodu i danych w segmencie o nazwie _DATA. W programie można się do nich odwołać poprzez umieszczenie adresu segmentu _DATA w rejestrze segmentowym. Zarówno kod programu jak i jego dane muszą być umieszczone w segmentach. O to troszczy się asembler. Natomiast zadaniem programisty jest załadowanie rejestrów segmentowych odpowiednimi adresami.
Dla każdego segmentu istnieje odpowiadający mu wskaźnik położenia. Umieszczenie danej lub kodu w segmencie powoduje jego uaktualnienie tak, by wskazywał na następną wolną pozycję. Asembler, napotykając etykietę, umieszcza ją w tablicy symboli wraz z wartością wskaźnika, która to wartość jej odpowiada. Potem każde jej wystąpienie zastępowane jest tą wartością. Rysunek 2-1 pokazuje zawartość segmentu danych i tablicy symboli po przetłumaczeniu instrukcji w segmencie _DATA.
Po napotkaniu dyrektywy **.CODE** dane nie są już umieszczane w segmencie _DATA, lecz asembler zaczyna wypełniać segment o nazwie _TEXT. Pierwszą dyrektywą dla tego segmentu jest:

```
Start PROC
```

Oznacza ona, że instrukcje, które za nią występują, są częścią procedury **Start**. W procesorach 80x86 istnieją dwa rodzaje instrukcji powrotu z podprogramu: bliskie (NEAR) i dalekie (FAR). Wszystkie procedury bliskie są w tym samym segmencie kodu, podczas gdy procedury dalekie mogą być rozmieszczone w różnych segmentach. Jeśli dana procedura nie ma jawnie zadeklarowanego typu, to asembler uzupełnia go stosownie do użytej dyrektywy **.MODEL**. W modelu **SMALL** wszystkie procedury są typu NEAR.
Pierwszą instrukcją w procedurze **Start** jest:

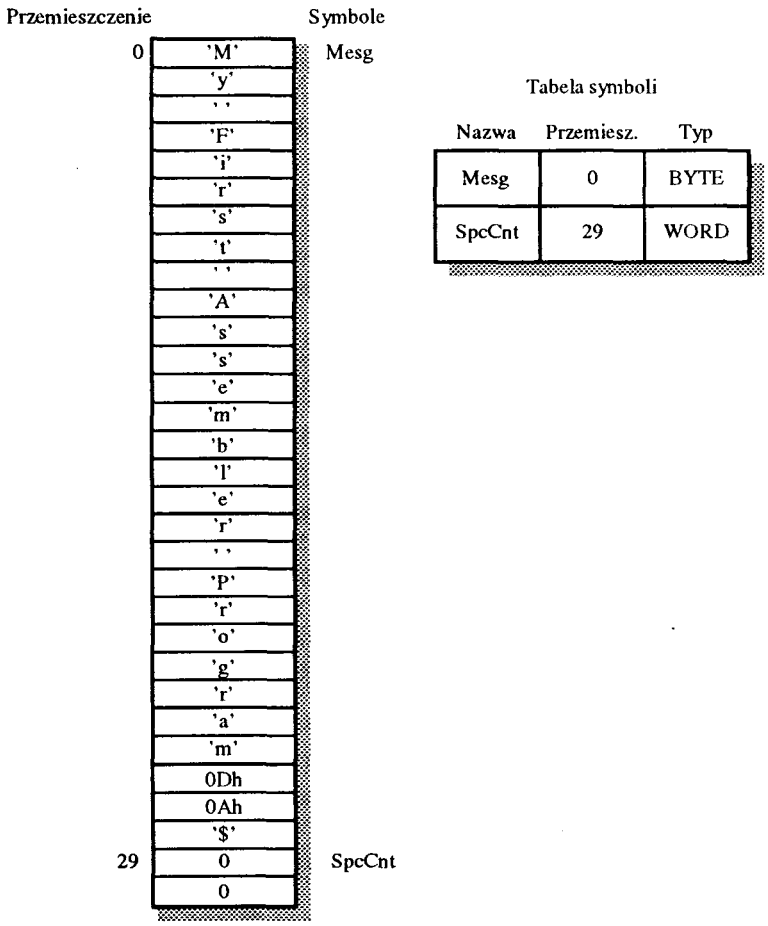
```
mov  ax, SEG DGROUP
```

Drugi argument jest segmentowym adresem etykiety o nazwie DGROUP. Wskazuje ona na strukturę zwaną *grupą*. Grupa to zbiór segmentów traktowanych jako pojedynczy segment (patrz Rozdział 8). W przykładzie segment _DATA jest połączony z segmentem STACK.
Wszystkie instrukcje, które znajdują się w segmencie _TEXT, są albo etykietami, albo instrukcjami języka asemblera. Etykiety są umieszczane przez asembler w pliku z symbolami, a instrukcje są przekształcane na kody zrozumiałe dla procesora 80x86.

Algorytm ten funkcjonuje do czasu, gdy asembler napotka linię:

```
call PrnStr
```

Nie może on znaleźć w tablicy symboli etykiety **PrnStr**, ponieważ nie została ona jeszcze zadeklarowana. Mimo wszystko umieszcza ją w niej, ale zaznacza, że odwołanie to nie zostało zrealizowane. Natomiast w kodzie wynikowym zamiast adresu umieszczane są dwa zerowe bajty. W modelu **LARGE**, gdzie procedury mają typ FAR, umieszczone zostaną cztery zera (rozmiar dalekiej etykiety).
Pewien problem pojawia się również przy instrukcjach skoku (JMP). Istnieją bowiem ich trzy rodzaje: dalekie (FAR), bliskie (NEAR) oraz krótkie (SHORT). Różna jest też długość tych instrukcji. Wynosi ona odpowiednio: 4, 2, i 1 bajt. Jeśli etykieta nie figuruje jeszcze w tablicy symboli, to asembler nie może określić, jakiego rodzaju skoku użyć. W kodzie wynikowym umieszcza więc cztery instrukcje puste NOP. Dopiero po znalezieniu symbolu uzupełnia to miejsce prawdziwym adresem.



Rysunek 2-1. ***Segment danych i tablica symboli***

By zapobiec wstawianiu do kodu wynikowego instrukcji pustych, należy poinformować asembler, jaki rodzaj skoku ma być użyty. Na przykład, by wymusić skok krótki powinna być zastosowana instrukcja:

```
jmp SHORT dest
```

Innym sposobem uniknięcia problemów, które pojawiają się przy korzystaniu z odwołań w przód, jest asemblacja wieloprzebiegowa. Wtedy każdy przebieg dostarcza asemblerowi więcej informacji o tych odwołaniach. Standardowo asembler dokonuje tylko jednego przejścia. Można jednak zmienić liczbę przebiegów, używając do tego celu opcji /mNUMBER. Większość programów może być skompilowana w mniej niż pięciu przebiegach.

Ostatnią dyrektywą w przykładzie 2-1 jest dyrektywa **END**. Oznacza ona koniec zbioru źródłowego. Może również zawierać etykietę, która definiuje punkt wejścia do programu. Tak więc wykonywanie programu z przykładu rozpocznie się od procedury **Start**. Asembler w kodzie wynikowym umieszcza kod, który informuje MS-DOS o adresie tej procedury.

Dyrektywy

Tabela 2-3 zawiera wszystkie dyrektywy Turbo Assemblera. Dzielą się one na cztery kategorie: dyrektywy sterujące segmentami, symbole i wyrażenia, makrodefinicje oraz dyrektywy, dotyczące kompilacji warunkowej. W takiej też kolejności wymienione są one w tabeli. Przy każdej grupie podany został numer rozdziału, w którym można znaleźć więcej informacji o każdej z nich.

Tabela 2-3. ***Dyrektywy Turbo Assemblera***

Dyrektywy segmentowe (Rozdział 8)
.ALPHA, ASSUME, .CODE, CODESEG, .CONST, CONST, .DATA, DATASEG, .DATA?, DOSSEG, .FARDATA, FARDATA, .FARDATA?, GROUP, .MODEL, MODEL, SEGMENT, .STACK, STACK, UDATASEG, UFARDATA
Symbole i wyrażenia (Rozdział 9)
$, @code, @CodeSize, @CPU, @curseg, @data, @DataSize, ??date, @fardata, @fardata?, @FileName, ??filename, @Model, @Startup, ??time, ??version, @WordSize, (), *, +, -, ., /, :, [], AND, ALIGN, BYTE, BYTE PTR, CATSTR, .CONST, CONST, CODEPTR, .CREF, DATAPTR, DB, DD, DF, DP, DQ, DT, DUP, DWORD, DWORD PTR, EQ, EQU, EVEN, EVENDATA, EXTRN, FAR, FAR PTR, FWORD, FWORD PTR, GE, GLOBAL, GT, HIGH, INSTR, LARGE, LE, LENGTH, LOW, LT, MASK, MOD, NE, NEAR, NEAR PTR, NOT, OFFSET, OR, ORG, PTR, PWORD, PWORD PTR, QWORD, QWORD PTR, RECORD, SEG, SHL, SHORT, SHR, SIZE, SIZESTR, SMALL, SUBSTR, SYMTYPE, TBYTE, TBYTE PTR, THIS, .TYPE, TYPE, UNION, UNKNOWN, WIDTH, WORD, WORD PTR, XOR
Makrodefinicje i Procedury (Rozdział 10)
ARG, ENDM, ENDP, EXITM, IRP, IRPC, LOCAL, MACRO, PROC, REPT, USES

Asemblacja warunkowa i sterowanie procesorem

.186, .286, .286C, .286P, .287, .386, .386C, .386P, .387, .8086, .8087, ELSE, ELSEIF, EMUL, END, ENDIF, .ERR, ERR, .ERR1, .ERR2, .ERRDEF, .ERRDIF, .ERRDIFI, .ERRE, .ERRIDN, .ERRIDNI, ERRIF, ERRIF1, ERRIF2, ERRIFB, ERRIFDEF, ERRIFDIF, ERRDIFI, ERRIFE, ERRIFIDN, ERRIFIDNI, ERRIFNB, ERRIFNDEF, .ERRNB, .ERRNDEF, .ERRNZ, JUMPS, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFDIN, IFDINI, IFNB, IFNDEF, INCLUDE, INCLUDELIB, MASM, MASM51, MULTERRS, NOEMUL, NOJUMPS, NOLOCALS, NOMASM51, NOSMART, ORG, P186, P286, P286N, P286P, P287, P386, P386N, P386P, P387, P8086, P8087, PNO87, QUIRKS, SMART

Tryby IDEAL i MASM

Turbo Assembler jest zgodny z asemblerem MASM firmy Microsoft. Oznacza to, że może on skompilować programy napisane w asemblerze MASM. Jednakże, jeśli zgodność ta nie jest konieczna, to można przełączyć Turbo Assembler w tryb IDEAL. Tryb ten ma wiele atrakcyjnych zalet, np. proces kompilacji trwa o 30% krócej. Bardziej zwięzła jest też składnia dyrektyw i wygodniejsze manipulowanie złożonymi typami danych. Wszystkie te możliwości ujawniają się po zinterpretowaniu przez asembler dyrektywy **IDEAL**. Tryb ten jest również wywoływany po podaniu opcji **/jideal**. Rozdział 11 zawiera pełny opis obu trybów.

Opisy dyrektyw, które znajdują się w tej książce, dotyczą zarówno ich wersji w trybie MASM, jak i IDEAL. Niektóre dyrektywy różnią się między sobą tylko występowaniem kropki na początku ich nazwy. W obu trybach ich znaczenie jest takie samo. Polecenia bez kropek istnieją, gdyż w trybie IDEAL dyrektywy nie mogą zaczynać się od kropki.

Segmenty

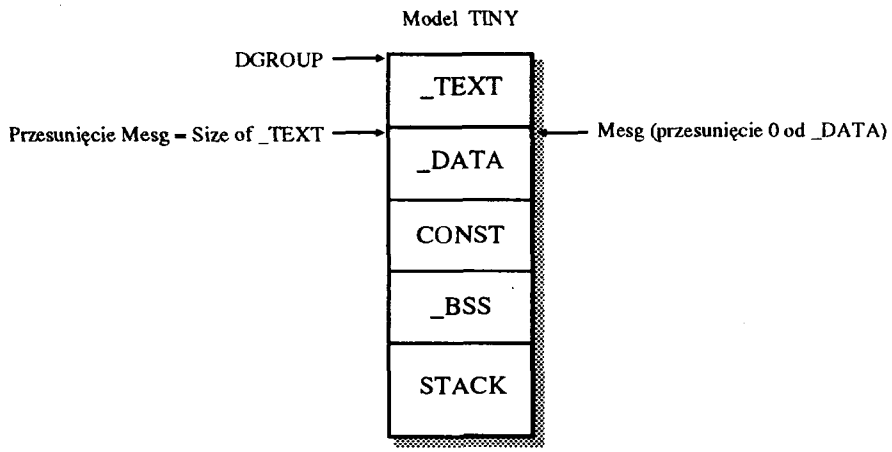
Turbo Assembler wymaga, by wszystkie dane i kod umieszczone były w *segmentach*, które mają własne nazwy. Nazwy te służą do przekazania asemblerowi informacji o zawartości rejestrów segmentowych. Dzięki temu asembler wie, jak obliczać wartości przesunięć danych. Na przykład, jeśli symbol jest zdefiniowany na początku segmentu o nazwie _DATA, to jego przemieszczenie wynosi 0. Jednak gdy poinformujemy asembler, że odpowiedni rejestr segmentowy zawiera wartość o 32 bajty mniejszą niż adres segmentu _DATA, to doda on 32 do wartości przesunięcia tego symbolu. Asembler ma wiele dyrektyw, które sterują rozmieszczeniem segmentów w pamięci oraz informują go o zawartości rejestrów segmentowych. Omówione są one w rozdziale 8.

Program z wydruku 2-1 używa kilku dyrektyw do umieszczenia danych w odpowiednich segmentach. Użyte dyrektywy (**.DATA, .CODE, .STACK**) zwane są uproszczonymi dyrektywami segmentowymi. Deklarują one predefiniowane segmenty _DATA, _TEXT oraz STACK. Program, który używa innych segmentów lub jest napisany dla starszej wersji kompilatora MASM, powinien użyć dyrektywy **SEGMENT**. Wymaga ona kilku dodatkowych informacji. Należy podać sposób łączenia danego segmentu z innymi i jego rozmieszczenie w pamięci. Dyrektywa **SEGMENT**, definiująca segment _DATA, ma postać:

```
_DATA SEGMENT WORD, PUBLIC, 'DATA'
```

Wraz z dyrektywą **SEGMENT** powinny również wystąpić polecenia **ENDS** oraz **ASSUME**. Dyrektywa **ENDS** oznacza koniec segmentu. Polecenie **ASSUME** mówi asemblerowi, jakiego adresu może się on spodziewać w rejestrach segmentowych. Jest to ważne, gdy kilka segmentów zostało połączonych w grupę. Jeśli w rejestrze znajduje się adres grupy, to asembler użyje zupełnie innych wartości przesunięć, niż wówczas gdyby zawierał on adres odpowiedniego segmentu. Ilustruje to rysunek 2-2. Gdyby w przykładzie umieszczonym na początku rozdziału nie użyto dyrektyw uproszczonych, to należałoby podać dyrektywę **ASSUME** o postaci:

```
ASSUME CS:_TEXT, DS:DGROUP, ES:NOTHING, SS:STACK
```



Rysunek 2-2. ***Przesunięcia przy adresowaniu względem grupy oraz segmentu.***

W większości programów wystarczy użycie uproszczonych dyrektyw segmentowych. Nazwy segmentów dobrano w nich tak, by odpowiadały one nazwom, używanym w językach wysokiego poziomu. Upraszcza to sposób łączenia programów, napisanych w asemblerze, z programami powstałymi w innych językach.

Symbole i wyrażenia

Asembler zapewnia wiele dyrektyw do operowania na symbolach. W najprostszej sytuacji podstawia on za symbol odpowiadający mu ciąg znaków lub liczbę. Na przykład, program, który odczytuje z pliku rekordy, może używać symboli, które oznaczają ilość bajtów w rekordzie oraz rozmiar bufora pamięci:

```
RecSize EQU 22          ; Rozmiar rekordu
BuffSize EQU 1024
```

Stała **RecSize** może być użyta do wyznaczania ilości bajtów, jaką należy skopiować z bufora. Symbol **BuffSize** można natomiast wykorzystać przy obliczaniu ilości rekordów, które mieszczą się w buforze. Wprawdzie można również w tym celu zdefiniować nowy symbol o wartości 46, lecz wygodniej jest obliczenia te zlecić asemblerowi. Poniższa instrukcja umieszcza wyliczoną liczbę rekordów w rejestrze CX:

```
mov  cx, BUFFSIZE/RECSIZE
```

Należy zdawać sobie sprawę, że wartości wszystkich wyrażeń obliczane są w procesie kompilacji. Nie można więc używać w wyrażeniach danych, które znajdują się w pamięci czy w rejestrach. Do przeprowadzenia obliczeń konieczne jest wtedy wykorzystanie instrukcji procesora. Na przykład, poniższe wyrażenie jest błędne:

```
mov ax, RECSIZE * bx ; w wyrażeniach nie można używać
                     ; rejestrów
```

Zapisane poprawnie powinno wyglądać następująco:

```
mov  ax, RECSIZE     ; Umieść wartość RECSIZE w AX
mul  bx              ; Pomnóż ax razy bx i umieść
                     ; rezultat w ax
```

Makrodefinicje i procedury

Poniższy przykład wyświetla na ekranie pewien ciąg znaków, a następnie odczytuje dane od użytkownika:

Wydruk 2-2. ***Wywołania procedur***

```
; Plik    Macexam.ASM
         .MODEL SMALL
Prnstr    EQU  9                    ; Funkcja systemowa,
                                    ; która wyświetla
                                    ; ciąg znaków
Inpstr    EQU  0Ah                  ; Funkcja systemowa,
                                    ; która pobiera ciąg
                                    ; znaków
Exitprg   EQU  4Ch                  ; Funkcja systemowa,
                                    ; która kończy program
         .DATA
Prmpt     DB   "Enter string: $"
InputStr  DB   40                   ; Maksymalna liczba
                                    ; odczytywanych znaków
          DB   ?                    ; Odczytana liczba
                                    ; znaków (ustawiana
                                    ; przez GetStr)
          DB   40 DUP('$')          ; Miejsce na
                                    ; odczytywany ciąg
                                    ; znaków
         .CODE
Start     PROC
          mov  ax, SEG DGROUP       ; Pobierz adres
                                    ; segmentu danych
          mov  ds, ax               ; Umieść go w DS
          mov  dx, OFFSET Prmpt     ; Pobierz
                                    ; przemieszczenie
                                    ; wyświetlanego ciągu
```

```
                                ; znaków
        call PrintS             ; Wyświetl go
        mov  dx, OFFSET InputStr ; Pobierz adres danych
                                ; dla funkcji GetStr
        call GetStr             ; Odczytaj ciąg znaków
        mov  dx, OFFSET InputStr ; Pobierz adres
                                ; odczytanego ciągu
                                ; znaków
        call PrintS
        mov  ah, Exitprg
        int  21h
Start   ENDP
; Procedura wyświetlająca ciąg znaków na ekranie
PrintS  PROC
        mov  ah, Prnstr         ; Numer komendy dla
                                ; MS-DOS-a w AH
        int 21h                 ; Wywołanie MS-DOS
        ret                     ; Powrót z podprogramu
PrintS  ENDP
; Procedura do pobierania ciągu znaków
GetStr  PROC
        mov  ah, Inpstr         ; Numer komendy dla
                                ; MS-DOS-a w AH
        int 21h                 ; Wywołanie MS-DOS
        ret                     ; Powrót z podprogramu
GetStr  ENDP
       .STACK 512
END     Start
```

Początek tego programu jest podobny do tego z przykładu 2-1. W segmencie danych zostały zdefiniowane dwie zmienne. Pierwsza z nich - **Prmpt** - zawiera ciąg znaków, który będzie wyświetlony. Druga natomiast - **InputStr** - składa się z trzech części. Pierwszą z nich jest bajt, który określa rozmiar trzeciej. W drugim bajcie zostanie umieszczona liczba faktycznie odczytanych znaków, a część trzecia jest buforem, w którym zostaną umieszczone pobrane znaki. Takiego formatu zmiennej spodziewa się się właśnie funkcja systemowa, która pobiera ciąg znaków.

Kod programu zawiera trzy procedury. Procedura Start ładuje rejestr DS adresem grupy DGROUP. Następnie umieszcza przesunięcie zmiennej **Prmpt** w rejestrze DX i wywołuje funkcję **PrnStr**. Procedura ta, używając funkcji systemowej, wyświetla podany ciąg znaków i zwraca sterowanie do pierwszej instrukcji za rozkazem CALL. W naszym przykładzie instrukcja ta ładuje rejestr DX przemieszczeniem zmiennej **InputStr**. Następnie zostaje wywołana funkcja **GetStr**. W ostatnim kroku program wyświetla pobrany ciąg znaków.

Procedura **Start** mogłaby być napisana bez wykorzystania wywołań funkcji wyświetlającej i pobierającej ciąg znaków. Oznaczałoby to konieczność dwukrotnego powtórzenia kodu funkcji **PrnStr**, a tym samym zwiększenie rozmiarów programu. Natomiast przewagą takiej

wersji byłaby większa szybkość działania wynikająca z opuszczenia instrukcji CALL i RET.
Właśnie makrodefinicje pozwalają włączać wielokrotnie używany kod bez konieczności jego powtórnego wpisywania. Makrodefinicja, równoważna funkcji **PrnStr**, może wyglądać następująco:

```
PrnStr MACRO
       mov  ah, Prnstr ; Numer komendy dla MS-DOS-a w AH
       int  21h        ; Wywołanie MS-DOS
PrnStr ENDM
```

Po napotkaniu symbolu **PrnStr** asembler zastępuje go instrukcjami, które znajdują się w makrodefinicji.

Kompilacja warunkowa

Kompilacja warunkowa pozwala na pisanie kodu źródłowego, który umożliwia uzyskiwanie różnych wersji programu zależnie od wartości pewnych symboli. Można więc stworzyć różne wersje programu z początku rozdziału. Na przykład jedna z nich mogłaby pracować z ekranem 80-kolumnowym, a druga z 40-kolumnowym.
W przykładzie tym, dwie linie nad etykietą **NextLp**, znajduje się instrukcja, która sprawdza, czy linia nie jest zbyt długa. Dla ekranu, który ma 80 kolumn linia jest uznawana za zbyt długą, gdy wartość CX jest większa niż 50, a dla 40-kolumnowego, gdy jest większa od 10. Poniższy fragment programu ustawia długość linii na 50, gdy symbol **SCREEN80** jest zdefiniowany, a na 10, gdy zdefiniowany jest symbol **SCREEN40:**

```
           .
           .
           .
ifdef      SCREEN80
           cmp  cx, 51
elseifdef  SCREEN40
           cmp  cx, 11
else
           display "No Screen size selected"
           err
endif
           .
           .
           .
```

Pierwsza dyrektywa warunkowa (**ifdef SCREEN80**) sprawdza, czy symbol **SCREEN80** został już zdefiniowany. Jeśli tak - kompilowane są linie, znajdujące się pomiędzy tą dyrektywą, a dyrektywą **elseifdef**. Następnie asembler omija wszystkie linie, aż do napotkania dyrektywy **endif**. Jeśli natomiast symbol ten nie został zdefiniowany, to linie pomiędzy **ifdef** a **elseifdef** nie są kompilowane.
Gdy został zdefiniowany symbol **SCREEN40**, to asembler kompiluje tylko instrukcje, zawarte pomiędzy dyrektywami **elseifdef** a **else**. Jeśli natomiast również ten symbol nie został zdefiniowany, wówczas asembler wykonuje instrukcje, które występują po **else**.

Pierwsza z nich powoduje wyświetlenie na ekranie komunikatu, druga zaś powiadamia asembler o zaistnieniu błędu. To powoduje wypisanie na ekranie numeru linii oraz zaniechanie tworzenia pliku wynikowego.
By skompilować nowy program należy więc zdefiniować symbol **SCREEN80** lub **SCREEN40**. Można to zrobić za pośrednictwem dyrektywy **EQU** (tak jak pozostałe symbole) lub też wykorzystując opcję /d. Pozwala ona zdefiniować symbole używane w programie. Na przykład, by zdefiniować symbol **SCREEN80**, należy użyć komendy

```
tasm /dSCREEN80 first
```

Instrukcje

Instrukcje są zamieniane przez asembler na kod zrozumiały dla procesora. Na przykład, pierwszymi instrukcjami w przykładzie 2-1 są:

```
mov  ax, SEG DGROUP
mov  ds, ax
```

Asembler przekształci te instrukcje na kod:

```
B8 00 00 8E D8
```

Każda instrukcja w programie jest pojedynczą instrukcją maszynową. Długość instrukcji maszynowych waha się od 1 do 3 bajtów plus rozmiar argumentów. Tabela 2-4 zawiera wszystkie instrukcje rozpoznawane przez Turbo Assembler. Podzielone są one na cztery kategorie, a przy każdej z nich podana jest informacja o rozdziale, w którym można znaleźć dokładny opis rozkazów.

Tabela 2-4. ***Zbiór rozkazów procesora 80x86***

Rozkazy przesyłania danych (Rozdział 4)
IN, INS(286), LDS, LEA, LES, LFS(386), LGS(386), LODS, LSS(386), MOV, MOVS, MOVSX(386), MOVZX(386), OUT, OUTS(386), POP, POPA(286), POPF, PUSH, PUSHA(286), PUSHF, REP, REPE, REPZ, REPNE, REPNZ, SETA(386), SETNBE(386), SETAE(386), SETNB(386), SETB(386), SETNAE(386), SETBE(386), SETNA(386), SETC(386), SETE(386), SETZ(386), SETG(386), SETNLE(386), SETGE(386), SETNL(386), SETL(386), SETNGE(386), SETLE(386), SETNG(386), SETNC(386), SETNE(386), SETNZ(386), SETNO(386), SETNP(386), SETPO(386), SETNS(386), SETO(386), SETP(386), SETPE(386), SETS(386), STOS, XCHG, XLAT, XLATB

Rozkazy operacji arytmetycznych i logicznych (Rozdział 5)
AAA, AAD, AAM, AAS, ADC, ADD, AND, BSF(386), BSR(386), BT(386), BTC(386), BTR(386), BTS(386), CBW, CDQ(386), CMP, CMPS, CWD, CWDE(386), DAA, DAS, DEC, DIV, IDIV, IMUL, INC, MUL, NEG, NOT, OR, RCL, RCR, ROL, ROR, SAL, SAR, SBB, SCAS, SHL, SHLD(386), SHRD(386), SUB, TEST, XOR

Procedury, pętle, skoki (Rozdział 6)
BOUND(286), CALL, ENTER(386), ESC, HLT, INT, INTO, IRET, JA, JAE, JB, JBE, JC, JCXZ, JE, JECXZ(386), JG, JGE, JL, JLE, JMP, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JP, JPE, JPO, JS, JZ, LEAVE(386), LOCK, LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ, NOP, RET, RETF, RETN
Sterowanie procesorem (Rozdział 7)
ARPL(286), CLC, CLD, CLI, CLTS(286), CMC, ESC, HLT, LAHF, LAR(286), LGDT(286), LIDT(286), LLDT(286), LMSW(286), LOCK, LSL(286), LTR(286), SAHF, SGDT(286), SIDT(286), SLDT(286), SMSW(286), STC, STD, STI, STR(286), VERR(286), VERW(286), WAIT

Przesyłanie danych

Procesor 80x86 ma kilka instrukcji, które przesyłają dane. Instrukcja MOV może przesłać dane pomiędzy dwoma rejestrami lub rejestrem a pamięcią. Po napotkaniu instrukcji MOV asembler sprawdza jej argumenty. Jeśli oba są rejestrami asembler od razu generuje odpowiedni kod. Jeśli jeden z nich zawiera odwołanie do pamięci, to asembler najpierw ustala tryb adresowania i dopiero wtedy generuje odpowiedni kod wynikowy.
Niektóre instrukcje operują na wielu słowach pamięci. Zwane są one instrukcjami łańcuchowymi, gdyż przetwarzają łańcuchy (ciągi) danych.

TRYBY ADRESOWANIA

W procesorach 80286 i starszych istnieje dziewięć sposobów podania adresu danej, która znajduje się w pamięci operacyjnej. Procesory 80386 mają jeszcze dodatkowe trzy tryby. Wszystkie one są przedstawione w tabeli 2-5.

Tabela 2-5. ***Tryby adresowania pamięci***

Tryb adresowania	Opis	
Domyślny	Kilka instrukcji (CBW, MUL, LAHF...) operuje tylko na pewnych rejestrach. Najczęściej jest to 8- lub 16-bitowy akumulator (AL lub AX).	
	cbw	; Przekształć bajt w AL na słowo w AX
Natychmiastowy	Dana zawarta jest bezpośrednio w rozkazie mikroprocesora. Tryb ten używany jest do ładowania rejestrów.	
	mov ax, 1234h	; Załaduj rejestr AX stałą 1234h
Bezpośredni	Adres danej zawarty jest bezpośrednio w rozkazie.	
	mov ax, Symbol	; Załaduj daną spod adresu Symbol ; do AX
Indeksowy	Adres danej znajduje się w rejestrze DI lub SI.	
	mov [si], ax	; Prześlij zawartość rejestru AX ; pod adres dany w SI

Tryb adresowania	Opis
Bazowy	Adres danej znajduje się w rejestrze BX lub BP. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS. mov ax, [bx] ; Prześlij daną o adresie podanym ; w BX do AX.
Indeksowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru SI lub DI oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem. mov ax, [si+4] ; Dodaj 4 do SI i prześlij daną zawartą ; pod tym adresem do AX
Bazowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru BX lub BP oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS. mov [bp-2], ax ; Prześlij AX pod adres BP-2
Bazowy indeksowany	Adres danej jest sumą zawartości rejestru BX lub BP oraz zawartości rejestru DI lub SI. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS. mov ax, [bx+di] ; Dodaj BX i DI, a następnie spod tego ; adresu prześlij daną do AX
Bazowy indeksowany z przemieszczeniem	Adres danej jest sumą zawartości rejestru BX lub BP, zawartości rejestru DI lub SI oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS. mov ax, [bp+si+2] ; Dodaj BP, SI i 2, a następnie spod tego ; adresu prześlij daną do AX
Skalowany indeksowy z przemieszczeniem	Adres danej jest sumą zawartości podanego rejestru pomnożonej przez współczynnik (2, 4, 8) oraz przemieszczenia (bajt, słowo lub podwójne słowo), które występuje za rozkazem (tylko 80386). mov [ebx*2+Table],ax ; Prześlij AX pod adres będący sumą ; przemieszczenia Table oraz zawartości ; rejestru EBX pomnożonej przez 2
Skalowany bazowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru bazowego oraz zawartości innego rejestru pomnożonej przez współczynnik (2, 4, 8) (tylko 80386). mov eax,[ebx+ecx*4] ; Prześlij do EAX daną spod adresu ; będącego sumą rejestru EBX oraz ; zawartości rejestru ECX pomnożonej ; przez 4

(kontynuacja)

Tryb adresowania	Opis (kontynuacja)
Skalowany bazowy indeksowany z przemieszczeniem	Adres danej jest sumą zawartości rejestru bazowego, zawartości innego rejestru pomnożonej przez współczynnik (2, 4, 8) oraz przemieszczenia (bajt, słowo lub podwójne słowo), które występuje za rozkazem (tylko 80386). mov [ebx*8+ecx+4],cx ; Prześlij CX pod adres będący sumą ; ECX, EBX*8 oraz przemieszczenia 4.

Adresowanie domyślne nie wymaga podania żadnych argumentów, gdyż instrukcje operują tylko na określonych rejestrach. Takimi instrukcjami są CBW (przekształć bajt na słowo) oraz CWD (przekształć słowo na podwójne słowo). Operują one tylko na rejestrach AH i AX (CBW) lub AX i DX (CWD).

W trybie *adresowania natychmiastowego* argument zawarty jest bezpośrednio w rozkazie. Może on być tylko argumentem źródłowym o stałej wartości. Tryb ten pozwala załadować 16-bitowy rejestr 8-bitową stałą. Asembler automatycznie użyje odpowiedniego rodzaju instrukcji, zależnie od wartości tej stałej.

Następnym trybem jest *adresowanie bezpośrednie*. W samym rozkazie zawarte jest tylko przemieszczenie danej. Standardowo adres segmentu pobierany jest z rejestru DS. Przyporządkowanie to można jednak zmienić, podając w rozkazie nazwę żądanego rejestru segmentowego:

```
mov  es:Storage, ax        ; Dana zawarta jest w segmencie
                           ; wskazywanym przez ES.
```

Procesory 80x86 zawierają dwa rejestry indeksowe DI i SI. Używane są one w *indeksowym trybie adresowania*. Tryb ten bywa najczęściej stosowany przy odwoływaniu się do tablic danych. By pobrać daną z komórki tablicy należy najpierw obliczyć jej adres. Na przykład, by odwołać się do trzeciego elementu tablicy słów, zaczynającej się od adresu **IArray**, można użyć następującego kodu:

```
mov  si, 3                 ; w SI numer elementu
shl  si, 1                 ; Mnożenie przez rozmiar
                           ; elementu tablicy
add  si, OFFSET IArray     ; Dodanie adresu pierwszego
                           ; elementu
mov  ax, [si]              ; Pobranie danej
```

Podobnie jak rejestry SI i DI mogą być wykorzystane rejestry BX i BP. Tryb ten zwany jest *bazowym trybem adresowania*. Jest on wolniejszy od *trybu indeksowego*. Poza tym, używając rejestru BP odwołujemy się do danych zawartych w segmencie, wskazywanym przez SS a nie DS.

W przykładzie, dotyczącym adresowania indeksowego, przemieszczenie pierwszego elementu tablicy musi być dodane do rejestru SI. Adresowanie indeksowe z przemieszczeniem pozwala ominąć ten krok. W tym trybie do rejestru indeksowego dodawany jest adres, który występuje za rozkazem. Po wykorzystaniu tego trybu przykład przyjmie postać:

```
mov  si, 3                 ; w SI numer elementu
```

```
shl  si, 1                ; Mnożenie przez rozmiar
                          ; elementu tablicy
mov  ax, [IArray+si]      ; Pobranie danej
```

W podobny sposób w trybie *adresowania bazowego z przemieszczeniem* mogą być wykorzystane rejestry BX i BP. *Adresowanie bazowe z indeksowaniem* sprowadza się do dodania do siebie zawartości rejestru bazowego i indeksowego oraz, w trybie adresowania bazowego z indeksowaniem i przemieszczeniem, dodatkowo dodania przemieszczenia.
W procesorach 80386 rolę rejestru indeksowego może pełnić dowolny rejestr danych. Poza tym jego zawartość może być pomnożona przez współczynnik 2, 4, lub 8. Nasz przykład redukuje się wtedy do dwóch linii:

```
mov si, 3                 ; w SI numer elementu
mov ax, [IArray+si*2]     ; Pobranie danej z tablicy
```

Jednakże w każdym z trybów jednym z argumentów musi być rejestr. Nie można więc przesłać danej między dwoma miejscami pamięci bez wykorzystania jakiegoś rejestru. Jedynym wyjątkiem jest możliwość przesłania danej natychmiastowej bezpośrednio do pamięci.

Podsumowanie

Pisanie programów składa się z kilku etapów. Pierwszym jest edycja zbioru źródłowego, później następuje kompilacja, konsolidacja i na koniec testowanie powstałego programu. W czasie testowania zostają wykryte błędy, należy więc wtedy wrócić do pierwszego kroku. Pakiet Turbo Assemblera zawiera narzędzia do kompilacji i konsolidacji programów. Starano się napisać tak je, by były wygodne i łatwe w użyciu. W najprostszym przypadku należy tylko podać nazwy plików do tłumaczenia czy konsolidacji. Każda z tych aplikacji zawiera zestaw wielu opcji, których listę można przejrzeć, wywołując dany program z opcją /h. Organizator MAKE może pomóc w tworzeniu gotowego programu. Sprawdza on, czy wszystkie zmienione pliki zostały skompilowane i dołączone już do programu wykonywalnego. Odbywa się to poprzez porównanie czasów i dat ostatniej modyfikacji plików źródłowych i wynikowych. Jeśli zbiór źródłowy jest nowszy niż wynikowy, to przeprowadzana jest jego ponowna kompilacja.
Istnieją dwa rodzaje komend, z których składa się program asemblerowy. Dyrektywy sterują pracą asemblera. Nie generują żadnego kodu wynikowego, aczkolwiek mogą mieć wpływ na jego postać. Na przykład, dyrektywy kompilacji warunkowej same nie tworzą kodu, lecz mają wpływ na to, czy blok nimi objęty będzie włączony do programu, czy też nie.
Reszta programu to instrukcje dla procesora. Asembler przekształca je na kody zrozumiałe dla procesora 80x86. Każda instrukcja programu stanowi pojedynczą instrukcję maszynową. Daje to pełną kontrolę nad pracą procesora. Nie ma żadnych zbędnych instrukcji, chyba że programista sam je w programie umieści.

Rozdział 3 *Turbo Debugger*

W każdym programie mogą wystąpić błędy. Mogą być one wynikiem niewłaściwego zrozumienia problemu, niewiedzy o działaniu danej instrukcji, efektów ubocznych podprogramów czy też po prostu zwykłej pomyłki. Jednym ze sposobów ich usunięcia z programu jest analiza tekstu źródłowego. Okazuje się jednak, że metoda ta może być źródłem podobnych błędów. Inną techniką usuwania błędów jest dodanie do programu kodu, który wyświetla wartości pewnych interesujących nas zmiennych czy rejestrów. Sposób ten pozwala również na stwierdzenie, które fragmenty programu się wykonują, a które nie. Jednakże dodany kod może zmieniać działanie programu.
Na szczęście istnieje jeszcze jedna alternatywa: oprogramowanie, które ułatwia śledzenie uruchamianych programów. Pozwala ono na uruchomienie pewnego fragmentu kodu i na analizę zawartości dowolnych rejestrów i obszarów pamięci. Istnieje wiele tego rodzaju aplikacji. Rozdział ten zawiera opis uruchamiacza Turbo Debugger oraz jego zastosowanie przy usuwaniu błędów z programów, napisanych w Turbo Assemblerze.
Sposób działania Turbo Debuggera zostanie przedstawiony w przykładzie, w którym znajduje się kilka błędów.

Użycie Turbo Debuggera

Turbo Debugger pozwala na krokowe wykonywanie programu oraz analizę jego kodu źródłowego, zawartości rejestrów i pamięci. Uruchamiacz umieszcza na ekranie kod, który jest aktualnie wykonywany. Program może być uruchamiany instrukcja po instrukcji, można też ustawić punkty kontrolne i tylko w nich obserwować jego zachowanie się.
Turbo Debugger zawiera wiele komend, które ułatwiają śledzenie programów. Są one zorganizowane w postaci rozwijanych menu. Program zawiera także wbudowany system pomocy, który zapewnia informację o dowolnie wybranej komendzie. Po nabyciu pewnego doświadczenia używanie Turbo Debuggera nie będzie sprawiało wielu kłopotów.

Przygotowanie programu do uruchomienia

Informacja o nazwach symbolicznych programu i numerach jego wierszy nie jest częścią zwykłego programu w systemie MS-DOS. Jednakże jest ona wymagana przez Turbo Debugger. Należy ją więc do programu wykonywalnego dołączyć, używając odpowiednich opcji asemblera (/zi) i konsolidatora (/v). By skompilować i skonsolidować program o nazwie MISTAKE.ASM tak, by był gotowy do uruchomienia przez Turbo Debugger, należy podać następujące komendy:

```
tasm /zi mistake
tlink /v mistake
```

Powodują one umieszczenie w programie wykonywalnym informacji o nazwach symbolicznych i numerach linii kodu źródłowego. Programy mogą być uruchamiane również bez tych danych. Jednak wiele użytecznych możliwości uruchamiacza staje się wówczas niedostępne.

Istnieją trzy wersje Turbo Debuggera, które różnią się między sobą typem procesora, na jakim mają być uruchamiane. Wersja standardowa działa na wszystkich komputerach zgodnych z IBM PC. Na komputerze z procesorem 80286 lub 80386 można wykorzystać wersje, które pracują w *trybie chronionym* (TD286 i TD386). Tryb chroniony oferuje kilka sprzętowych możliwości, które Turbo Debugger może wykorzystać. Jednakże wszystkie trzy wersje mają takie same komendy. Dzięki temu poznanie jednej wersji zapewnia znajomość pozostałych. By rozpocząć sesję uruchomieniową z programem MISTAKE.EXE należy użyć polecenia:

```
td mistake
```

Ekrany Turbo Debuggera

Po uruchomieniu Turbo Debuggera powinien ukazać się ekran podobny do rysunku 3-1. W najwyższej jego linii znajduje się menu główne. Linia najniższa informuje o funkcjach niektórych klawiszy. Obie te linie ułatwiają wyszukanie potrzebnej komendy. Największe okienko na ekranie to okienko modułów. Znajduje się w nim właśnie uruchamiany zbiór źródłowy. Należy zauważyć, że okienko to jest obwiedzione podwójną linią, co coznacz, że jest okienkiem aktywnym. Wszystkie polecenia z klawiatury są więc kierowane właśnie do niego.

Poniżej okienka modułów znajduje się okienko obserwacyjne. Mogą być w nim umieszczone nazwy zmiennych programu, które mają być analizowane. Okienko to ma linię pojedynczą. Nie jest więc ono okienkiem aktywnym. By je uaktywnić, należy znać numer tego okienka. Podany jest on w prawym górnym rogu każdego okienka. Jeśli teraz zostanie wciśnięty klawisz ALT i jednocześnie klawisz, odpowiadający numerowi okienka, stanie się ono okienkiem bieżącym.

W czasie sesji uruchomieniowej będzie się zmieniać zarówno liczba okienek, jak i ich położenie na ekranie. Zawierają one bardzo różne rodzaje informacji. Wykorzystanie myszy może więc bardzo ułatwić ich organizację. Operacje zmiany położenia i rozmiaru dowolnego okienka można również przeprowadzić, korzystając z poleceń menu Window.

Rysunek 3-1. ***Początkowy ekran Turbo Debuggera***

Komendy

Turbo Debugger nie ma, tak jak program DEBUG, wiersza poleceń. W dodatku, do systemu menu i klawiszy funkcyjnych, komendy można podawać w specjalnych okienkach. Na przykład, wciśnięcie klawisza Enter w okienku obserwacyjnym powoduje edycję wyrażenia, które znajduje się pod kursorem. Niektóre funkcje, znajdujące się w menu, można aktywować poprzez wciśnięcie odpowiednich kombinacji klawiszy np. funkcji Quit z menu File odpowiada kombinacja ALT-X. Jeśli funkcji odpowiada pewna kombinacja klawiszy, to informacja o tym znajduje się w menu obok jej nazwy.
W każdym momencie, gdy nie jesteś pewien działania jakiegoś polecenia, możesz nacisnąć klawisz F1. Pojawi się wtedy okienko podpowiedzi, które zawiera informację o poleceniu lub okienku, w jakim znajduje się właśnie kursor. Można również wyszukać potrzebną komendę, a następnie wyświetlić o niej informację.

KLAWISZE FUNKCYJNE

Wiele funkcji, częściej stosowanych, może być wywołanych przez wciśnięcie odpowiedniej kombinacji klawiszy. Na przykład, by prześledzić działanie dowolnej instrukcji, możesz użyć klawisza F7. Mimo że większości klawiszy funkcyjnych odpowiada opcja w menu, to jednak korzystanie z nich jest prostsze i szybsze niż wyszukiwanie w menu odpowiedniej opcji. Ponadto, funkcje klawiszy funkcyjnych są wyświetlone w najniższej linii ekranu. Natomiast, by otrzymać podpowiedź, dotyczącą wybranego polecenia, musisz odnaleźć je w menu, ustawić nań kursor i wcisnąć klawisz F1.
Wciśnięcie i przytrzymanie klawisza ALT ujawnia więcej możliwości klawiszy funkcyjnych. Rozszerza to listę częściej używanych funkcji. Tak jak i poprzednio, również i tym klawiszom funkcyjnym odpowiada pewne polecenie w menu. Mimo że komendy te nie są tak często wykorzystywane, to zapoznanie się z nimi może okazać się przydatne.
Trzecia grupa klawiszy funkcyjnych różni się trochę od dwóch pozostałych. Tak jak i poprzednio po wciśnięciu klawisza CTRL ukaże się ich lista. Natomiast, by wywołać żądane polecenie, należy użyć pierwszej litery nazwy funkcji, zamiast klawisza funkcyjnego. Można również zauważyć, że lista ta jest zależna od rodzaju bieżącego okienka. Na przykład, gdy bieżącym okienkiem jest okienko modułów, to dostępne są komendy takie jak *Module*, które pozwalają na zmianę załadowanego zbioru. Natomiast, gdy aktywne jest okienko obserwacyjne, to by zmienić zawartość tego okienka, można wywołać polecenie *Edit*.
Funkcje te zwane są *menu lokalnym*. Można zobaczyć to menu w innym formacie po wciśnięciu klawiszy ALT-F10 lub prawego przycisku myszy. Gdy menu lokalne zostanie wywołane tym sposobem, to można umieścić kursor nad dowolną jego opcją, a następnie, po wciśnięciu F1, zapoznać się z opisem wybranej komendy.

MENU GŁÓWNE

Najprościej korzystać jest z menu głównego, używając myszy. Wystarczy tylko umieścić jej kursor nad żądaną opcją i wcisnąć lewy przycisk. Oczywiście, mimo braku myszy również można korzystać z menu. Po wciśnięciu klawisza F10 uaktywnia się menu główne. W tym momencie klawisze strzałkowe pozwalają na przemieszczanie się pomiędzy pozy-

cjami menu. Klawisz Enter powoduje natomiast rozwinięcie się podmenu. Należy teraz wybrać żądaną opcję i wywołać ją wciskając Enter.
W każdym menu znajduje się pewien typ komend. Tabela 3-1 zawiera krótki opis każdego z nich. Również klawisz F1 pozwala zapoznać się z informacją o każdym menu.

Tabela 3-1. ***Menu główne Turbo Debuggera***

System	Odświeżenie aktualnego ekranu
File	Wybranie programu do uruchamiania
View	Dodawanie nowych okienek
Run	Wykonywanie fragmentów kodu
Breakpoints	Ustawianie pułapek
Data	Edycja i wyświetlanie danych
Options	Ustalanie opcji Turbo Debuggera
Window	Zmiana rozmiarów i położenia okienek
Help	Wyświetlenie informacji o uruchamiaczu

Przykład

Poniższy program powinien wypisać zachętę, odczytać odpowiedź od użytkownika, zapisać ją do pliku, a następnie zakończyć swoje działanie. Jednakże poprawnie wykonywane są tylko trzy pierwsze czynności. Natomiast zamiast zakończenia swojej działalności program wraca znowu do zadania pytania.

Wydruk 3-1. ***Program, który zawiera błędy***

```
; File: MISTAKE.ASM
; Program demonstrujący zastosowanie uruchamiacza
.MODEL SMALL, C
.DATA
NmPrmt DB "Name: $"       ; Zachęta
FileNm DB "RECORD.DAT",0  ; Nazwa pliku
; Następne trzy pola muszą być umieszczone obok siebie
InStrSz DB 40             ; Zmienne dla MS-DOS, umieszczony
                          ; zostanie w nich odczytany ciąg
                          ;znaków
InStrCt DB 0
InStrDt DB 40 dup(?)
CREATFILE EQU 3Ch
WRTDATA EQU 40h
CLOSEFILE EQU 3Eh
PROGEXIT EQU 4Ch
PRNTSTR EQU 9h
GETSTR EQU 0Ah
CHAROUT EQU 2h
LF EQU 0Ah
```

```
CR EQU 0Dh
.CODE
Start PROC
  mov ax, DGROUP          ; Ustal adres segmentu danych
  mov ds, ax              ; Załaduj nim DS
  mov es, ax              ; oraz ES
  mov dx, OFFSET FileNm   ; Ustal parametry dla funkcji,
                          ; która tworzy plik
  mov ah, CREATFILE       ; Numer funkcji systemowej,
                          ; która tworzy plik
  sub cx, cx              ; Stwórz plik zwykły
  int 21h                 ; Wywołaj system
  mov bx, ax              ; Skopiuj deskryptor pliku do BX
  call GetData            ; Pobierz odpowiedź
  mov ah, WRTDATA         ; Numer funkcji systemowej,
                          ;która zapisuje dane do pliku
  mov dx, OFFSET InStrDt  ; Pobierz adres danych
  mov ch, 0                ; Rozmiar danych
  mov cl, InStrCt
  int 21h                 ; Wywołaj system
  mov ah, CLOSEFILE       ; Numer funkcji systemowej,
                          ; która zamyka plik
  int 21h                 ; Wywołaj system
  mov ah, PROGEXIT        ; Numer funkcji systemowej,
                          ; która kończy program
  int 21h                 ; Wywołaj system
Start ENDP
; Wyświetl zachętę i pobierz odpowiedź
GetData PROC
  push bx                 ; Zachowaj BX (deskryptor pliku)
  mov dx, OFFSET NmPrmt   ; Pobierz adres zachęty
  mov ah, PRNTSTR         ; Numer funkcji systemowej,
                          ;która wyświetla ciąg znaków
  int 21h                 ; Wywołaj system
  mov dx, OFFSET InStrSz  ; Pobierz adres zmiennych
                           ; wejściowych
  mov ah, GETSTR          ; Numer funkcji systemowej,
                          ; która pobiera ciąg znaków
  int 21h                 ; Wywołaj system
  mov ah, CHAROUT         ; Numer funkcji systemowej,
                          ; która wyświetla znak
  mov dl, CR              ; Znak przejścia do początku linii
  int 21h                 ; Wywołaj system
  mov dl, LF              ; Znak przejścia do następnej linii
  int 21h                 ; Wywołaj system
  ret                     ; Powrót z podprogramu
```

```
GetData ENDP
.STACK 512
END Start
```

W programie zdefiniowanych zostało pięć zmiennych. Pierwsza z nich (**NmPrmt**) jest tekstem zachęty dla użytkownika. Następna (**FileNm**) jest nazwą pliku, do którego zostaną zapisane dane. Kolejne trzy zmienne (**InStrSz, InStrCt, InStrDt**) definiują dane, żądane przez funkcję systemową, która odczytuje ciąg znaków.

Segment kodu zawiera dwie procedury. Pierwsza (**Start**) ładuje rejestr DS, otwiera plik, wywołuje funkcję **GetData**, zapisuje dane, zamyka plik i zwraca sterowanie do systemu. Natomiast procedura **GetData**, używając funkcji systemowych, wypisuje zachętę, pobiera od użytkownika odpowiedź, przenosi kursor do nowej linii i zwraca sterownie do programu, który ją wywołał.

Program po skompilowaniu, skonsolidowaniu i uruchomieniu albo ciągle pyta się użytkownika o jego imię, albo zawiesza się. Dopiero wciśnięcie klawiszy CTRL-C powoduje jego zakończenie. Program więc nie pracuje tak, jak tego oczekiwaliśmy. Pora więc na skorzystanie z Turbo Debuggera.

Najpierw należy upewnić się, czy program został skompilowany i skonsolidowany z odpowiednimi opcjami (/zi i /v). Teraz można uruchomić Turbo Debugger. Powinien pojawić się ekran podobny do tego z rysunku 3-1.

Teraz, używając klawisza F7, można wykonywać przykładowy program instrukcja po instrukcji. Funkcja ta powoduje wykonanie bieżącego rozkazu i zatrzymanie się przed następnym. Brakuje jeszcze tylko możliwości obejrzenia skutków wykonanej instrukcji. W większości asemblerowych programów wygodne jest obejrzenie chociażby tylko zawartości rejestrów. Menu View zajmuje się sterowaniem ekranu, a więc to właśnie w nim powinna znajdować się potrzebna komenda. Najbardziej odpowiednim wydaje się być polecenie *Registers*. Po jego wybraniu otwiera się okienko, w którym wyświetlone są zawartości wszystkich rejestrów (patrz rysunek 3-2). Teraz, używając myszy albo poleceń menu Windows, można dopasować jego rozmiar i położenie.

Rysunek 3-2. ***Okienko rejestrów***

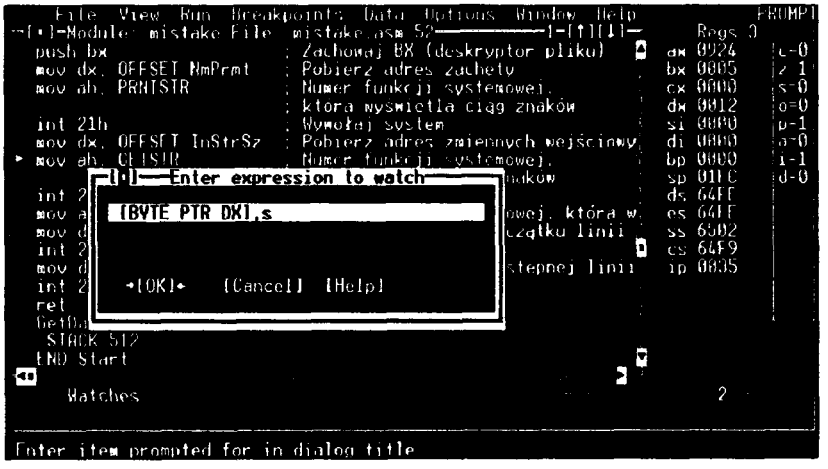
Lewa część okienka Registers pokazuje trzynaście rejestrów procesora 80x86. Natomiast po prawej stronie wyświetlony jest stan ośmiu flag procesora. Są to znaczniki: przeniesienia, zera, znaku, nadmiaru, parzystości, przeniesienia pomocniczego, zezwolenia na przerwanie i kierunku dla operacji łańcuchowych.

W trakcie śledzenia programu należy obserwować zmiany poszczególnych rejestrów. Po osiągnięciu instrukcji **call GetData** można wybrać dwie możliwości. Pierwszą z nich jest kontynuacja procesu śledzenia. Powoduje to wejście do procedury GetData. Natomiast druga możliwość pozwala procesorowi na wykonanie całej procedury, a uruchamiacz przejmuje sterowanie dopiero po wykonaniu instrukcji powrotu. Proces ten zwany jest *pracą krokową*, a wywoływany jest klawiszem F8. Jeśli tak właśnie postąpisz w naszym przykładzie, to program zacznie znowu żądać podania nazwy. Można z tego wnioskować, że błąd ukryty jest w procedurze **GetData**.

Niestety, obecnie jest już za późno, by wykryć przyczynę błędu. Proces uruchamiania należy zacząć od początku. W tym celu należy wydać polecenie Program Reset z menu Run. Powoduje to przywrócenie warunków początkowych.

Teraz można użyć klawisza F4, czyli wywołać komendę *Go To Cursor*. Jednak najpierw należy przemieścić kursor na pierwszą linię procedury **GetData**. Po wciśnięciu klawisza F4 program jest wykonywany, aż do napotkania wiersza, w którym znajduje się kursor. Wtedy sterowanie znowu przejmuje uruchamiacz.

Pierwszą instrukcją jest **push bx**. W okienku rejestrów można sprawdzić, czy rejestr ten zawiera wartość 5. Jeżeli wszystko jest w porządku, wówczas przechodzimy do następnej instrukcji (**mov dx, OFFSET NmPrmt**). Po jej wykonaniu stwierdzamy, że rzeczywiście rejestr DX zmienił swoją zawartość. Niestety, wartość ta nie mówi nam zbyt wiele. Ale, używając okienka obserwacyjnego, można zobaczyć, co znajduje się pod adresem zawartym w DX. By je uaktywnić należy wcisnąć klawisze ALT-2. Następnie klawiszami CTRL-W można dodawać nowe wyrażenia. Rysunek 3-3 przedstawia wygląd ekranu po wprowadzeniu żądanego wyrażenia. Łańcuch **[BYTE PTR DX],s** jest poleceniem dla uruchamiacza, które powoduje wyświetlenie ciągu znaków, wskazywanego przez DX. Efektem zadziałania tego polecenia będzie równoczesne wyświetlenie ciągów znaków **Name: $** i **RECORD.DAT**. Powodem wyświetlenia przez uruchamiacz również napisu **RECORD.DAT**



Rysunek 3-3. ***Okienko obserwacyjne***

jest to, że Turbo Debugger rozpoznaje koniec ciągu znaku po zerowym bajcie. Spoglądając do kodu źródłowego można przekonać się, że pierwszym zerowym bajtem, który znajduje się za tekstem zachęty, jest bajt kończący nazwę pliku.

MS-DOS wypisuje natomiast poprawny tekst, gdyż do zaznaczenia końca ciągu znaków używa znaku dolara ($). Można to sprawdzić, po wykonaniu instrukcji **int 21h**, przełączając się na ekran użytkownika. W tym celu należy wydać polecenie ALT-F5. Powoduje ono zmianę ekranu na ekran użytkownika. Stan ten utrzymuje się do chwili naciśnięcia dowolnego klawisza. Po przełączeniu się, przy lewej krawędzi ekranu powinien pojawić się napis **Name:**.

Po powrocie, używając klawisza F7, kontynuujemy śledzenie. Po dojściu do następnej instrukcji **int 21h** można wcisnąć F7 lub F8. Należy zauważyć, że uruchamiacz automatycznie przełączył się na ekran użytkownika. Stało się tak dlatego, że wywołanie MS-DOS wymaga dostępu do wejścia/wyjścia. Po otrzymaniu odpowiedzi uruchamiacz znowu przejmuje sterowanie, proces śledzenia może być więc kontynuowany.

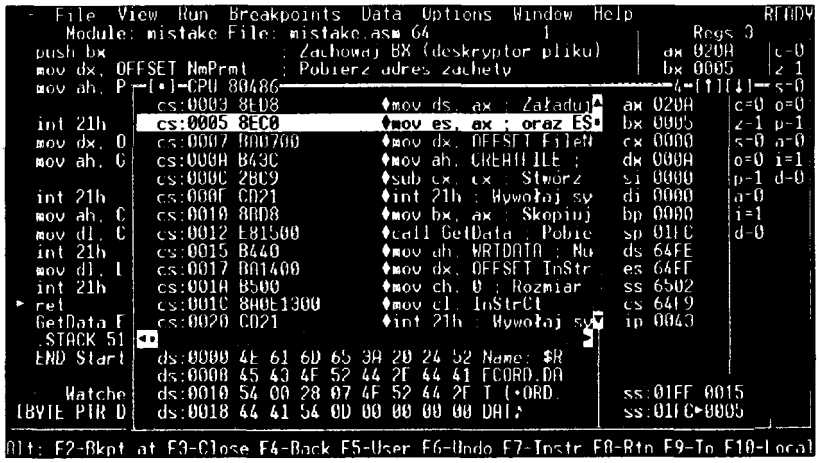
Używając klawiszy F7 lub F8 dochodzimy wreszcie do instrukcji powrotu. Wiadomo już, że powrót z tej procedury nie odbywa się poprawnie. Należy więc znaleźć wyjaśnienie, dlaczego instrukcja **ret** nie działa prawidłowo. Sprawdźmy najpierw zawartość stosu. Na jego szczycie powinien znajdować się adres powrotu. Okienko Dump pozwala na obejrzenie zawartości dowolnego obszaru pamięci, choćby właśnie stosu. Należy więc teraz wydać komendę Dump, która znajduje się w menu View. Wybranie tej komendy powoduje pojawienie się okienka takiego, jakie jest przedstawione na rysunku 3-4.

Należy teraz podać adres szczytu stosu. Poleceniem CTRL-G można zmienić adres wyświetlanego w okienku Dump obszaru pamięci. Adres ten może być podany w formacie segment, przesunięcie np. **1234:0002** lub też może być wyrażeniem, które można użyć w programie asemblerowym np. wyrażenie **ss:sp** określa szczyt stosu. Po jego wprowadzeniu widzimy, że pierwszą wartością jest 05.

By sprawdzić, co znajduje się w segmencie kodu pod adresem 05 (miejsce, pod które zostanie przekazane sterowanie po wykonaniu instrukcji powrotu), użyjemy okienka procesora (rysunek 3-5). Odpowiednia komenda znajduje się w menu View. Górna, lewa jego

```
 - File  View  Run  Breakpoints  Data  Options  Window  Help                    READY
 ------Module: mistake File: mistake.asm 64 ----------1----------Regs-3-------
   push bx                  ; Zachowaj BX (deskryptor pliku)     ax 020H   c=0
   mov dx, OFFSET NmPrmt    ; Pobierz adres zachety              bx 0005   z=1
   mov ah, PRNTSTR          ; Numer funkcji systemowej,          cx 0000   s=0
                            ; która wyswietla ciąg znakow        dx 0000   o 0
   int 21h                  ; Wywołaj system                     si 0000   p=1
   mov dx, OFFSET InStrSz   ; Pobierz adres zmiennych wejsciowy  di 0000   a=0
   mov ah, ┌f*1-Dump──────────────────────────────6-IIIIII─      bp 0000   i 1
           │ ss 01FC 05 00 15 00 FB 52 10 03 ↕ $ nR►♥ ▲          sp 01FC   d=0
   int 21h │ ss 0204 48 00 00 00 09 00 21 00 H   = !  ▼          ds 641E
   mov ah, │ ss 020C 00 00 07 00 00 00 01 00   ·      tóra w     es 64FF
   mov dl, │ ss:0214 00 00 00 00 1F 00 01 00     ▼ :  ▼linii     ss 6502
   int 21h ◄►                                        ►            cs 6419
   mov dl, 1F               ; Znak przejścia do nastepnej linii  ip 0043
   int 21h                  ; Wywołaj system
 ► ret                      ; Powrot z podprogramu
   GetData ENDP
   .STACK 512
   END Start
 ------Watches-----------------------------------------------------------2-------
 BYTE PTR DX1.s                "ORD.DAT"

Alt-F2 Bkpt at F3 Close F4 Back F5 User F6 Undo F7 Instr F8 Rtn F9 To F10 Local
```

Rysunek 3-4. ***Okienko Dump***



Rysunek 3-5. ***Okienko procesora (CPU)***

część zawiera kod programu, który znajduje się w pamięci. Za pomocą klawiszy kursora możemy się po nim przemieszczać. Sprawdźmy więc, co znajduje się pod adresem 05.
Jest to instrukcja **mov es, ax**. Nie jest to więc rozkaz, który występuje po wywołaniu procedury **GetData** (**mov ah, 40h**). Musimy się więc dowiedzieć, co spowodowało ten bałagan na stosie. Wciśnięcie klawiszy ALT-1 uaktywnia okienko modułów, możemy więc przeanalizować tekst źródłowy. Używając klawiszy kursora wyświetlmy na ekranie procedurę **GetData**, a następnie poszukajmy w niej instrukcji, które operują na stosie. Pierwszą z nich jest **push bx**. Powoduje ona odłożenie na stos zawartości rejestru BX, by po zakończeniu podprogramu mogła ona zostać odtworzona. Okazuje się jednak, że nasza funkcja nie zawiera rozkazu **pop bx**, który zdejmuje ze stosu zawartość BX. Więc to jest szukany błąd. Używając klawiszy ALT-X należy opuścić Turbo Debugger i przy pomocy dowolnego edytora dodać instrukcję POP tuż przed rozkazem powrotu z procedury **GetData**. Po ponownym skompilowaniu i skonsolidowaniu program działa już poprawnie.

Podsumowanie

Turbo Debugger jest silnym narzędziem, które pozwala śledzić sposób pracy programu. I to nie tylko programów napisanych przez nas samych. Dzięki niemu można zrozumieć działanie również programów stworzonych przez innych programistów. Oczywiście, wtedy nie śledzi się ich linia po linii, lecz wykonuje całe fragmenty kodu i analizuje wyniki ich działania.
Główną zaletą Turbo Debuggera jest możliwość śledzenia programów na poziomie kodu źródłowego. Zamiast więc rozszyfrowywania binarnych liczb, z których się on składa, można go obejrzeć ze wszystkimi symbolami i komentarzami. Poza tym, nie trzeba operować na adresach danych czy kodu. Wystarczy podać nazwę odpowiedniego symbolu.
Turbo Debugger pozwala również na oglądanie zawartości pamięci. Zawartość ta może być interpretowana na wiele sposobów. Obszar pamięci może być widziany jako lista szesnastkowych liczb w okienku Dump lub też wyświetlony jako ciąg znaków, liczba całkowita czy znak w okienku Watch.

Turbo Debugger posiada również wiele komend, które pozwalają uruchamiać dowolne fragmenty programu. Program może być śledzony instrukcja po instrukcji, wykonany tylko do pewnego punktu lub też pewien jego fragment może być powtórzony określoną ilość razy. Pozwala to na ominięcie mało interesujących fragmentów kodu. Ponieważ programy będą stawały się coraz większe i większe, pozwoli to zaoszczędzić dużo czasu i umożliwi szybsze pisanie poprawnych programów.

II Instrukcje Procesora

- » Rozkazy transmisji danych
- » Instrukcje arytmetyczne i logiczne
- » Procedury, pętle i skoki
- » Sterowanie procesorem i operacje w trybie chronionym

Rozdział 4 Rozkazy transmisji danych

Przesyłanie danych jest jedną z ważniejszych operacji wykonywanych przez procesor 80x86. Wiele instrukcji procesora 80x86 czy funkcji MS-DOS wymaga, by dane znajdowały się w określonych rejestrach. Najczęściej dane te są pobierane ze zmiennych, które znajdują się w pamięci. Pierwsza część rozdziału opisuje rozkazy, które przesyłają dane między pamięcią i rejestrami.
Wiele funkcji do przechowania danych używa zmiennych tymczasowych. Jeśli jest ich tylko kilka, to można je umieścić w rejestrach. Jednakże zmiennych może być więcej niż dostępnych rejestrów. Poza tym co zrobić, jeśli funkcja, korzystająca ze zmiennych tymczasowych, wywoła inną, która również zawiera zmienne tymczasowe. Druga część rozdziału pokazuje sposób wykorzystania stosu do przechowywania zmiennych tymczasowych.
Ważnym źródłem danych są porty wejścia/wyjścia. To właśnie za ich pośrednictwem procesor 80x86 komunikuje się ze światem zewnętrznym. Do obsługi klawiatury, ekranu monitora, portów równoległych i szeregowych, potrzebne są instrukcje wejścia/wyjścia, opisane w kolejnej części rozdziału.
Ostatnia część przedstawia instrukcje, które operują na blokach danych. Mimo że nazywają się one rozkazami łańcuchowymi, to nie ma żadnego powodu, by ograniczać ich zastosowanie tylko do łańcuchów znaków. Mogą one być użyte do każdej tablicy danych dowolnego typu.

Przesyłanie danych

Instrukcja MOV kopiuje dane z jednego miejsca w drugie. Format tego rozkazu jest następujący:

```
mov   przeznaczenie, źródło
```

Zarówno przeznaczenie, jak i źródło danych, mogą być rejestrami lub też jedno z nich może być miejscem w pamięci. Ponadto źródło może być daną natychmiastową. Natomiast niemożliwe jest przesłanie danej pomiędzy dwoma komórkami w pamięci.
Za pomocą instrukcji MOV można załadować i przechować zawartości rejestrów segmentowych. Jednak, gdy jednym z argumentów jest rejestr segmentowy, wówczas zabroniony jest natychmiastowy tryb adresowania. Poza tym ciekawa rzecz dzieje się przy ładowaniu segmentu stosu:

```
mov   ss, seg newstack
mov   sp, offset stackpoint
```

Jeśli między tymi dwoma instrukcjami pojawi się przerwanie, to będzie podjęta próba odłożenia znaczników oraz adresu powrotu na stos newstack. Jednakże wskaźnik stosu wskazuje w tym momencie nieokreślone miejsce w pamięci. Problem ten w procesorach 80x86 rozwiązano, blokując przerwania na czas wykonania instrukcji, która występuje za rozkazem MOV, dotyczącym rejestru SS. Jednak w starszych wersjach układów 8088 lub

8086 powyższe instrukcje muszą zostać otoczone parą rozkazów CLI (zablokuj przerwania) i STI (odblokuj przyjmowanie przerwań).

Tryby adresowania

W procesorach 80x86 istnieje dwanaście sposobów na przesłanie danych z rejestru lub do rejestru. Zwane są one trybami adresowania, które określają sposób obliczania adresu danej, znajdującej się w pamięci. Jest to jej adres efektywny (EA). Tabela 4-1 zawiera wszystkie dostępne tryby adresowania.

Tabela 4-1. Tryby adresowania pamięci.

Tryb adresowania	Opis
Domyślny	Kilka instrukcji (CBW, MUL, LAHF...) operuje tylko na pewnych rejestrach. Najczęściej jest to 8- lub 16-bitowy akumulator (AL lub AX).
`cbw`	`; Przekształć bajt w AL na słowo w AX`
Natychmiastowy	Dana zawarta jest bezpośrednio w rozkazie mikroprocesora. Tryb ten używany jest do ładowania rejestrów.
`mov ax, 1234h`	`; Załaduj rejestr AX stałą 1234h`
Bezpośredni	Adres danej zawarty jest bezpośrednio w rozkazie.
`mov ax, Symbol`	`; Załaduj daną spod adresu Symbol do AX`
Indeksowy	Adres danej znajduje się w rejestrze DI lub SI.
`mov [si], ax`	`; Prześlij zawartość rejestru AX` `; pod adres dany w SI`
Bazowy	Adres danej znajduje się w rejestrze BX lub BP. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS.
`mov ax, [bx]`	`; Prześlij daną o adresie podanym` `; w BX do AX.`
Indeksowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru SI lub DI oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem.
`mov ax, [si+4]`	`; Dodaj 4 do SI i prześlij daną zawartą` `; pod tym adresem do AX`
Bazowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru BX lub BP oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS.
`mov [bp-2], ax`	`; Prześlij AX pod adres BP-2`
Bazowy indeksowany	Adres danej jest sumą zawartości rejestru BX lub BP oraz zawartości rejestru DI lub SI. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS.

Tryb adresowania	Opis
`mov ax, [bx+di]`	`; Dodaj BX i DI, a następnie spod tego` `; adresu prześlij daną do AX`
Bazowy indeksowany z przemieszczeniem	Adres danej jest sumą zawartości rejestru BX lub BP, zawartości rejestru DI lub SI oraz przemieszczenia (bajt lub słowo), które występuje za rozkazem. Użycie rejestru BP powoduje przyjęcie rejestru segmentowego SS.
`mov ax, [bp+si+2]`	`; Dodaj BP, SI i 2, a następnie spod` `; tego adresu prześlij daną do AX`
Skalowany indeksowy z przemieszczeniem	Adres danej jest sumą zawartości podanego rejestru pomnożonej przez współczynnik (2, 4, 8) oraz przemieszczenia (bajt, słowo lub podwójne słowo), które występuje za rozkazem (tylko 80386).

`mov [ebx*2+Table],ax`	`; Prześlij AX pod adres będący sumą`
	`; przemieszczenia Table oraz`
	`; zawartości rejestru EBX pomnożonej`
	`; przez 2`

Skalowany bazowy z przemieszczeniem	Adres danej jest sumą zawartości rejestru bazowego oraz zawartości innego rejestru pomnożonej przez współczynnik (2, 4, 8) (tylko 80386).

`mov eax,[ebx+ecx*4]`	`; Prześlij do EAX daną spod adresu`
	`; będącego sumą rejestru EBX oraz`
	`; zawartości rejestru ECX pomnożonej`
	`; przez 4`

Skalowany bazowy indeksowany z przemieszczeniem	Adres danej jest sumą zawartości rejestru bazowego, zawartości innego rejestru pomnożonej przez współczynnik (2, 4, 8) oraz przemieszczenia (bajt, słowo lub podwójne słowo), które występuje za rozkazem (tylko 80386).
`mov [ebx*8+ecx+4],cx`	`; Prześlij CX pod adres będący sumą` `; ECX, EBX*8 oraz przemieszczenia 4.`

Ładowanie adresu efektywnego do rejestru

Im bardziej złożony jest tryb adresowania, tym większa jest ilość czasu, którego wymaga proces dostępu do danej. Jeśli do pewnego miejsca w pamięci istnieje potrzeba wielokrotnego odwoływania się, to dobrze by było obliczyć najpierw skomplikowany adres efektywny, zapamiętać go w rejestrze i później wykorzystać przy odwoływaniu się do tej komórki pamięci. Dokonuje tego instrukcja LEA (załaduj adres efektywny) o formacie:

lea rejestr, pamięć

W poniższym przykładzie wyznaczana jest suma czterech kolejnych słów, które znajdują się w tablicy pod indeksem SI+BX. Wykonanie tego programu zajmuje 184 takty.

```
   xor   ax, ax            ; wyzeruj rejestr ax
   mov   cx, 4
aloop:
   add   ax, table[si+bx]  ; dodaj wartość
   inc   bx                ; następne słowo
   inc   bx
   loop  aloop             ; powtórz cx razy
```

Użycie instrukcji LEA skraca czas wykonania programu do 173 cykli:

```
   xor   ax, ax
   mov   cx, 4
   lea   di, table[si+bx]  ; wyznacz w DI adres efektywny
aloop:
   add   ax, [di]
   inc   di
   inc   di
   loop  aloop
```

Prefiksowanie

W każdym z trybów adresowania adres efektywny jest tylko przemieszczeniem danej w standardowym segmencie. Na przykład, jeśli w trybie indeksowym zostanie użyty rejestr BP, to standardowym rejestrem będzie segment stosu (SS); do pozostałych trybów przypisany jest segment danych (DS). Przyporządkowanie to można zmienić, poprzedzając rozkaz prefiksem. By odwołać się do danej, która znajduje się w segmencie dodatkowym, można użyć instrukcji:

```
   mov   ax, es:data
```

Prefiksowanie jest często spotykane w programach, które mają kilka segmentów danych tzn. W modelach large, huge lub compact. W programach tych najczęściej używane dane umieszczone są w segmencie wskazywanym przez DS, a rejestr ES zawiera adres segmentu aktualnie potrzebnej danej.
Każdy prefiks dodaje dwa dodatkowe cykle zegara przy wyliczaniu adresu efektywnego. Nie jest to wielkim problemem w programach, które używają prefiksowania niewiele razy. Jednakże przy wielokrotnym prefiksowaniu należy rozważyć możliwość załadowania adresu używanego segmentu do rejestru DS. Na przykład:

```
   push  ds             ; Zachowaj bieżący segment danych
   mov   ax, ExtraSeg   ; Pobierz adres segmentu
   mov   ds, ax         ; Załaduj nim rejestr DS
   .                    ; Operacje na danych w segmencie
                        ; ExtraSeg
   .
   .
   pop   ds             ; Odtworzenie DS
```

Dalekie wskazania

Turbo Assembler zapamiętuje zarówno adres segmentu jak i przesunięcie każdego symbolu w programie. Jednakże dane, które znajdują się w segmencie danych, są identyfikowane przez podanie tylko ich przemieszczenia. Taki adres zwany jest wskazaniem bliskim. Natomiast, gdy dana znajduje się w innym segmencie, to program musi zmodyfikować zawartość rejestru segmentowego. Razem, adres segmentu i przesunięcie, tworzą wskazanie dalekie. Niektóre podprogramy zachowują wskaźniki w pamięci, po to by mogły być one wykorzystane w dalszej części programu. Jeśli są to wskazania dalekie, wówczas kod mógłby wyglądać następująco:

```
.DATA
FarDataOffset     DW    OFFSET DataItem ; Miejsce na
                                        ; przechowanie
                                        ; przemieszczenia
FarDataSegment    DW    SEG DataItem    ; Miejsce na
                                        ; przechowanie
                                        ; adresu segmentu
            .CODE
            .
            .
            .
            mov   bx, FarDataOffset
            mov   es, FarDataSegment
            .
            .
            .
```

Istnieje prostszy sposób osiągnięcia efektu z poprzedniego przykładu. W tym celu należy wykorzystać instrukcję LES:

```
            .DATA
FarDataPtr    DD    DataItem    ; Miejsce na przechowanie
                                ; adresu segmentu
                                ; i przemieszczenia
            .CODE
            .
            .
            .
            les   bx, FarDataPtr
            .
            .
            .
```

Instrukcja ta powoduje przesłanie podwójnego słowa spod adresu **FarDataPtr** do rejestrów ES i BX. Należy zauważyć, że zmienna **FarDataPtr** musi znajdować się w bieżącym segmencie danych. Natomiast zmianę adresu segmentu danych można osiągnąć, używając instrukcji LDS. Umieszcza ona daną w rejestrze przeznaczenia oraz DS.

Procesor 80386 ma dwa rejestry segmentowe więcej. Posiada też więcej instrukcji, które służą do ładowania dalekich wskazań. Nowe rejestry to FS i GS, a dodatkowe rozkazy to LFS (ładuj dalekie wskazanie do FS), LGS (ładuj dalekie wskazanie do GS), LSS (ładuj dalekie wskazanie do SS).

Zamienianie danych

Czasami istnieje potrzeba zamienienia dwóch wartości. Na przykład, konieczność ta zachodzi podczas sortowania danych. Operacja ta wykonana przy użyciu instrukcji MOV wymaga czterech rozkazów oraz dwóch dodatkowych rejestrów:

```
mov   ax, Data1      ; Przechowaj dane
mov   bx, Data2      ; w rejestrach
mov   Data1, bx      ; Odtwórz dane
mov   Data2, ax      ; w odwrotnej kolejności
```

Kod ten można skrócić, używając instrukcji XCHG o formacie:

```
XCHG  reg/mem, mem/reg
```

Reg/mem oznacza albo rejestr, albo miejsce w pamięci. Podobnie jak w innych instrukcjach tylko jeden z argumentów może być odwołaniem do pamięci.
Po zastosowaniu instrukcji XCHG przykład przyjmie postać:

```
mov   ax, Data1      ; Pierwszy argument w AX
xchg  ax, Data2      ; Zamiana wartości
mov   Data1, ax      ; Prześlij nową wartość AX do
                     ; pierwszego argumentu
```

Ładowanie akumulatora wartością z tablicy

Istnieje wiele sposobów zakodowania znaków w komputerze. W komputerach, które pracują pod systemem MS-DOS, zastosowano kod ASCII, ale inne systemy mogą używać innych sposobów kodowania np. EBCDIC lub BAUDOT. Instrukcja XLAT może okazać się pomocną przy konwersji jednego systemu kodowania na drugi. Rejestr BX musi wskazywać na tablicę, za pośrednictwem której odbywa się ta konwersja, a w rejestrze AL powinien znajdować się indeks elementu w tej tablicy. Po wykonaniu instrukcji będzie on zawierał wartość tego właśnie elementu.
Podobny efekt, i to nawet w dwa razy krótszym czasie, można uzyskać, stosując instrukcję MOV i indeksowy tryb adresowania. Jednakże wymaga to dwukrotnie większej ilości bajtów pamięci.
W następnym przykładzie przyjmujemy, że istnieje tablica, która zawiera kody znaków w standardzie EBCDIC, umieszczone w kolejności odpowiadającej systemowi ASCII. Procedura ta jest wywoływana ze znakiem w kodzie ASCII, umieszczonym w rejestrze AL, a zwraca kod EBCDIC tego znaku.

```
mov   bx, offset ASC2EBC  ; Pobierz adres tablicy
                          ; konwersji
xlat
```

Przesyłanie mniejszych danych do większych rejestrów

W procesorach 80386 istnieją rejestry 32-bitowe. Można bezpośrednio przesłać do nich daną 32-bitową. Jednak czasami istnieje potrzeba przesłania danej z rejestru 16-bitowego do rejestru 32-bitowego. Wydaje się, że można to zrobić tak:

```
        .DATA
dblspace   dd  1 dup(?)                ; Rezerwacja miejsca
                                       ; na daną 32-bitową
        .CODE
        mov   word ptr dblspace, ax    ; Prześlij rejestr AX
                                       ; do młodszego słowa
                                       ; zmiennej
        mov   word ptr dblspace+2, 0   ; Wyzeruj starsze
                                       ; słowo zmiennej
        mov   eax, dblspace            ; Załaduj rejestr
                                       ; 32-bitowy
```

Istniejąca w procesorach 80386 instrukcja MOVZX (prześlij uzupełniając zerami) bardzo ułatwia tego typu przesłania. W rozkazie tym rejestr źródłowy może być 8- lub 16-bitowy, natomiast rejestr przeznaczenia - 16- lub 32-bitowy. Przy użyciu tej instrukcji powyższy kod redukuje się do jednego rozkazu:

```
movzx  eax, ax
```

Powyższe przesłanie staje się jeszcze trudniejsze w przypadku danych ze znakiem. Należałoby do starszego słowa zmiennej 32-bitowej przesyłać 0 lub 0ffffh zależnie od znaku danej, znajdującej się w mniejszym rejestrze. Na szczęście istnieje instrukcja MOVSX, która rozszerza daną, zachowując jej znak.

Warunkowe ustawianie wartości bajtów

Jedną z wielu nowych instrukcji procesora 80386 jest instrukcja SET. Na podstawie znaczników stwierdza ona, czy pewien warunek jest spełniony, a jeśli to tak ustawia wartość bajtu, który jest jej argumentem, na 1. Istnieje 18 różnych warunków. Przedstawione są one w tabeli 4-2.

Tabela 4-2. Instrukcje SET w procesorze 80386

Nazwa	Opis
	Testy po operacjach arytmetycznych na liczbach bez znaku
SETB/SETNAE	Jeśli mniejsze/nie wieksze i nie równe (CF=1)
SETAE/SETNB	Jeśli większe lub równe/nie mniejsze (CF=0)
SETBE/SETNA	Jeśli mniejsze lub równe/nie wieksze (CF=1 ZF=1)
SETA/SETNBE	Jeśli większe/nie mniejsze i nie równe (CF=0 ZF=0)
	Testy po operacjach arytmetycznych na liczbach ze znakiem
SETL/SETNGE	Jeśli mniejsze/nie większe i nie równe (SF<>OF)

(kontynuacja)

niepotrzebnych już wartości. Działa on na zasadzie algorytmu LIFO (Last-In-First-Out). Ostatnie zapisane słowo jest pierwszym, które będzie pobrane.
Istnieje specjalny rejestr zwany wskaźnikiem stosu. Przy zapamiętywaniu nowej danej na stosie jego wartość jest najpierw zwiększana o dwa (rozmiar słowa), a następnie dana ładowana jest pod adres SS:SP. Pobieranie danej ze stosu jest procesem dokładnie odwrotnym. Najpierw jest ona odczytywana spod adresu SS:SP, a następnie zawartość SP zostaje zmniejszona o dwa.

Zapisywanie i pobieranie danych ze stosu

Instrukcje PUSH i POP nawzajem się uzupełniają. Jeśli w programie zostanie użyta instrukcja PUSH, to w dalszej jego części powinien pojawić się odpowiadający jej rozkaz POP. Brak uzupełniającej instrukcji POP jest częstym błędem, który pojawia się w programach asemblerowych.
Instrukcja PUSH zapisuje daną na stos. Jej format jest następujący:

```
PUSH  rejestr/pamięć/stała
```

Rejestr może być jednym z 16-bitowych rejestrów lub rejestrem segmentowym. Dana, znajdująca się w pamięci, traktowana jest również jako wartość 16-bitowa. Dopiero procesor 80286, może przesłać stałą na stos, a procesor 80386 ma dodatkowo możliwość przesyłania danych 32-bitowych.
Format instrukcji POP jest taki sam, jak instrukcji PUSH, a jej działanie ma dokładnie odwrotny skutek do działania rozkazu PUSH. Instrukcje te są bardzo wygodne do tymczasowego przechowywania wartości pewnych rejestrów, podczas gdy one same są wykorzystywane do innych celów. Na przykład, poniższy fragment kodu używa rejestru CX jako licznika w obydwu pętlach. Instrukcja LOOP może operować tylko na tym właśnie rejestrze. Dlatego wartość CX używana w pętli zewnętrznej musi być zachowana przed uruchomieniem pętli wewnętrznej.

```
        sub   ax, ax          ; Wyzeruj AX
        mov   cx, 20          ; Pętla zewnętrzna
                              ; będzie wykonana 20 razy
outer:
        push  cx              ; Przechowaj rejestr CX
        mov   cx, 15          ; Pętla wewnętrzna będzie
                              ; wykonana 15 razy
inner:
        inc   ax
        loop  inner           ; Wykonuj pętlę wewnętrzną
        pop   cx              ; Odtwórz CX
        loop  outer           ; Wykonuj pętlę zewnętrzną
                              ; Po jej zakończeniu
                              ; AX=300 ( 20 * 15 )
```

Jeśli podprogram ma być wywoływany z kilku różnych miejsc, to dobrą zasadą jest przechowanie rejestrów w nim wykorzystywanych, a po zakończeniu ich odtworzenie. Zapewnia to, że program wywołujący, uruchamiając ten podprogram, nie straci żadnych danych.

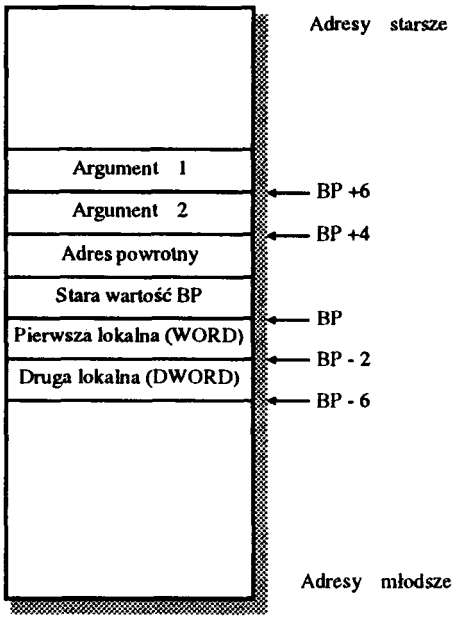
Jeśli procedura używa wszystkich rejestrów, to ich zapamiętanie na stosie wymaga 8 bajtów kodu oraz 24 cykli zegara. W procesorach 80286 i 80386 wprowadzono więc do tego celu pojedynczą instrukcję. PUSHA przesyła na stos wszystkie rejestry 16-bitowe, a PUSHAD 32-bitowe. By je odtworzyć należy użyć odpowiednio rozkazu POPA lub POPAD. Instrukcje te nie zachowują rejestrów segmentowych.
Ostatnim rejestrem, który można odłożyć na stos jest rejestr znaczników. Służy do tego instrukcja PUSHF, a w procesorze 80386, w którym rejestr ten jest 32-bitowy instrukcja PUSHFD. Rozkazy, zdejmujące wartość ze stosu do rejestru flag, mają postać: POPF i POPFD dla procesora 80386.

Przesyłanie danych na stosie

Bazowy tryb adresowania oraz adresowanie bazowe z przemieszczeniem w rejestrze BP odwołują się do segmentu stosu. Fakt ten wykorzystany został przy operowaniu na zmiennych lokalnych funkcji. Obszar stosu przeznaczony na dane podprogramu nazywany jest ramką stosu. Poniższy przykład ilustruje sposób przydzielenia, użycia i zwolnienia obszaru zmiennych lokalnych o rozmiarze trzech słów.

```
push  bp                  ; Zachowaj starą wartość BP
mov   bp, sp              ; BP wskazuje na szczyt stosu
sub   sp, 6               ; Rezerwacje miejsca na trzy
                          ; słowa
mov   word ptr [bp-2],1   ; Pierwsza zmienna ma
                          ; wartość 1
mov   word ptr [bp-4],2   ; Druga zmienna ma wartość 2
mov   word ptr [bp-6],3   ; Trzecia zmienna ma wartość 3
.
.
mov   sp, bp              ; Zwolnienie obszaru
                          ; zmiennych lokalnych
pop   bp                  ; Odtworzenie BP
ret
```

Ramki stosu mają duże znaczenie przy programowaniu modułowym. Zapewniają one możliwość przekazania argumentów do danej funkcji oraz miejsce na przechowanie zmiennych lokalnych. By przekazać parametry, program wywołujący odkłada je na stos, a następnie uruchamia podprogram. Może się on odwołać do swoich argumentów w podobny sposób jak do zmiennych lokalnych. Po zakończeniu działania funkcja przywraca oryginalną wartość rejestrowi SP, co powoduje zwolnienie przydzielonego miejsca i możność jego ponownego wykorzystania przez inne procedury.
Rysunek 4-1 pokazuje stos z umieszczonymi na nim kilkoma ramkami. Należy zauważyć, że aby odwołać się do zmiennych lokalnych przemieszczenie jest odejmowane od rejestru BP; natomiast odwołując się do argumentów wywołania jest ono dodawane.



Rysunek 4-1. ***Zmienne lokalne.***

Operacje łańcuchowe

Przy użyciu poprzednich rozkazów można się było odwołać do bajtu, słowa lub podwójnego słowa. Wygodnie byłoby jednak mieć możliwość np. skopiowania całego bloku danych. Procesor 80x86 ma kilka instrukcji, które operują na blokach danych w pamięci. Są to rozkazy łańcuchowe.

Rozkazy te wykorzystują rejestry indeksowe DI i SI. Ogólnie rejestr SI wskazuje na tablicę, która znajduje się w segmencie danych i jest źródłem danych. Natomiast rejestr DI identyfikuje tablicę w dodatkowym segmencie danych, w której będzie umieszczony wynik działania instrukcji. Podanie prefiksu ma wpływ tylko na segment źródła.

W trakcie działania rozkazy łańcuchowe modyfikują rejestry indeksowe. Wielkość poprawki zależy od rodzaju danych, na których instrukcja operuje. i tak, dla instrukcji bajtowych poprawka wynosi jeden, działających na słowach dwa, a na podwójnych słowach cztery. Natomiast, to czy będzie ona dodana czy odjęta od rejestru określa znacznik kierunku. Jeśli jest on wyzerowany, to rozmiar argumentu jest dodawany do rejestru indeksowego, a w przeciwnym razie odejmowany.

Kopiowanie tablic

Instrukcja MOVS przesyła bajt, słowo, lub podwójne słowo (tylko w 80386) z miejsca wskazywanego przez DS:SI pod adres ES:DI. Format tego rozkazu jest następujący:

```
MOVS  przeznaczenie, [seg:]źródło
```

W instrukcji tej argumenty podaje się tylko w celu wyznaczenia rozmiaru pojedynczego elementu. Mimo ich podania w instrukcji zostanie użyte standardowe adresowanie.

Innym sposobem, pozwalającym ustalić rozmiar przesyłanych elementów, jest dołączenie do nazwy instrukcji litery B (bajt), w (słowo) lub D (podwójne słowo).
Fakt, że rejestry segmentowe są każdorazowo uaktualniane, sprawia, że rozkaz ten może być użyty kilka razy z rzędu do skopiowania całej tablicy. Poniższy fragment kodu kopiuje 20 bajtów ze **Str1** do **Str2**:

```
   cld                      ; Zwiększanie adresów
   mov   si, OFFSET Str1    ; w SI adres Str1
   mov   di, OFFSET Str2    ; w DI adres Str2
   push  ds                 ; Skopiuj rejestr DS
   pop   es                 ; do ES
   mov   cx, 20             ; Liczba przesyłanych bajtów
mloop:
   movsb                    ; Prześlij bajt
   loop  mloop              ; Powtórz 20 razy
```

Przedrostek REP

W poprzednim przykładzie przesyłanie danych było zorganizowane za pomocą instrukcji LOOP. Użycie przedrostka REP pozwala uczynić kod jeszcze bardziej efektywnym. Jest on używany przed niektórymi instrukcjami, a jego wystąpienie powoduje powtórzenie rozkazu CX razy. W powyższym przykładzie pętla mogłaby być zastąpiona instrukcją:

```
   rep   movsb
```

Działanie przedrostka deklarującego powtórzenie uległo poprawie wraz z nowymi wersjami procesorów. W procesorach 8088 i 8086 było szybszym użycie pętli, a zajmowało tyle samo miejsca. Natomiast w procesorach 80286 i 80386 rozmiar pozostał ten sam, lecz skrócił się znacznie czas wykonania. Pewien problem pojawia się przy jednoczesnym użyciu przedrostka REP i prefiksu segmentu. Dotyczy on wszystkich procesorów z wyjątkiem 80386. Okazuje się, że rezultaty działania instrukcji łańcuchowych mogą być całkowicie przypadkowe, gdy w czasie jej działania pojawi się przerwanie. Należy więc być ostrożnym.

Pobieranie i umieszczanie danych w łańcuchach

Czasami istnieje potrzeba dokonania bardziej złożonej operacji niż tylko skopiowanie danych z jednego miejsca w drugie. Można, np. dodać do każdego elementu pewną wartość. Przesłanie będzie więc musiało zostać rozbite na dwa oddzielne kroki (pobranie i zapisanie danej). Etapy te realizowane są przez instrukcje LODS i STOS. Format ich jest bardzo podobny do formatu rozkazu MOVS, z tą różnicą, że wymagane jest podanie tylko jednego argumentu:

```
      LODS   [seg:], źródło
```

Argument ten determinuje rozmiar danych źródła. Prefiks segmentu jest opcjonalny. Należy jednak zauważyć, że rozbicie instrukcji MOVS na dwie daje możliwość poprzedzenia prefiksem obu argumentów przesłania.
Zarówno LODS jak i STOS używają akumulatora. Dla operacji 8-bitowych jest to rejestr AL, 16-bitowych AX, a 32-bitowych (tylko 80386) EAX.

Jednym z zastosowań instrukcji LODS i STOS jest zamiana małych liter na duże w trakcie kopiowania. W poniższym przykładzie łańcuch **Str1** jest kopiowany do **Str2** przy jednoczesnej zamianie małych liter na duże.

```
   cld                          ; Zwiększanie adresów
   mov   si, OFFSET Str1        ; w SI adres Str1
   mov   di, OFFSET Str2        ; w DI adres Str2
   push  ds                     ; Skopiuj rejestr DS
   pop   es                     ; do ES
Uloop:
   lodsb                        ; Pobranie bajtu
   or    al, al                 ; Ustawienie znaczników
   jz    Done
   cmp   al, 'a'                ; Czy jest to mała litera
   jb    NotLow                 ; Jeśli al < 'a', to nie
   cmp   al, 'z'
   ja    NotLow                 ; Jeśli al > 'z', również nie
   sub   al, 20h                ; Zamień na dużą literę
NotLow:
   stosb                        ; Zapisz bajt
   jmp   Uloop
Done:
```

Instrukcja STOS wraz z przedrostkiem REP może być użyta do wypełniania obszaru pamięci stałą wartością. W poniższym przykładzie 18-bajtowy blok danych zostanie wypełniony spacjami (kod ASCII 20h). Zastosowanie instrukcji STOSW powoduje, dzięki redukcji liczby obiegów pętli, zmniejszenie czasu potrzebnego na wykonanie programu.

```
   cld                          ; Zwiększanie adresów
   mov   di, OFFSET Spaces      ; w DI adres bloku
   mov   ax, 2020h              ; w AX dwie spacje
   mov   cx, 9                  ; Liczba słów do przesłania
   rep   stosw
```

Sprawdzanie zakresów tablic

Dobrą praktyką programowania jest sprawdzanie, czy podany adres mieści się w granicach tablicy. Zaniechanie tego może spowodować zmianę zawartości komórki pamięci, która to komórka nie powinna zostać zmieniona i często jest to przyczyną nieokreślonego zachowania się programu. Tego rodzaju błąd jest później bardzo trudny do znalezienia.
W procesorach 8088 i 8086 sprawdzenie poprawności odwołania musi być zrealizowane przy użyciu instrukcji porównujących i skoków warunkowych:

```
   cmp   si, Lowlimit           ; Porównaj wskaźnik
                                ; z ograniczeniem dolnym
   jb    OutBounds              ; Jeśli jest mniejszy, skocz
                                ; do procedur obsługi błędu
   cmp   si, Upperlim           ; Porównaj wskaźnik
                                ; z ograniczeniem górnym
```

```
    ja      OutBounds           ; Jeśli jest za duży, skocz
                                ; do procedur obsługi błędu
    .                           ; Adres jest poprawny
    .
    .
OutBounds:                      ; Np. wyświetl komunikat
                                ; o błędzie
```

W procesorach 80286 i 80386 operacja ta została uproszczona. Wykonuje ją pojedyncza instrukcja BOUND. Jej format jest następujący:

```
BOUND  rejestr, zakresy
```

Rejestr zawiera sprawdzany adres, natomiast drugi argument jest wskazaniem miejsca pamięci, w którym znajdują się dolny i górny zakres. Trudność użycia tej instrukcji wynika ze sposobu, w jaki informuje ona o błędzie. Zamiast ustawienia odpowiednich flag generowane jest przerwanie o numerze 5. Sposób ten nadaje się więc dla programu, który posiada kilka ogólnych działań, podejmowanych po wykryciu przekroczenia zakresu.
Poza tym w systemie MS-DOS przerwanie 5 jest używane do drukowania zawartości ekranu na drukarce. Wprawdzie można, korzystając z systemowych funkcji o numerach 25h i 35h, zachować stary wektor i zastąpić go adresem procedury obsługi błędów, lecz wtedy, również przy każdym wciśnięciu klawisza Print Screen, będzie ona wywoływana. Dlatego też stosowanie tej instrukcji w systemie MS-DOS nie jest polecane.

Operacje wejścia/wyjścia

Wykonując operację wejścia/wyjścia procesor wysyła specjalny sygnał, który oznacza, że żąda on dostępu do portów wejścia/wyjścia, a nie do pamięci. Następnie na magistrali adresowej umieszcza numer żądanego portu i czyta lub wysyła daną na magistralę danych. W komputerach kompatybilnych z IBM PC, wiele z tych portów ma specjalne zastosowanie. Na przykład, porty szeregowe i równoległe mają zawsze te same adresy. Inne są zdefiniowane tylko w niektórych modelach komputerów. Porty, używane przy dostępie do zegara czasu rzeczywistego, są różne na różnym sprzęcie. Przed użyciem instrukcji wejścia/wyjścia powinieneś przeczytać dokumentację, która dotyczy twojego komputera.

Instrukcje wejścia/wyjścia

Instrukcje IN i OUT pozwalają na dostęp do portów wejścia/wyjścia. Istnieją dwa formaty tych rozkazów. Jeśli numer portu zawiera się w przedziale od 0 do 255, to można podać go jako daną natychmiastową:

```
in      acc, port
out     port, acc
```

Acc jest rejestrem akumulatora (AL lub AX), natomiast port jest 8-bitowym adresem portu wejścia/wyjścia. Jeśli jest on większy niż 255, należy użyć innego formatu:

```
in      acc, dx
out     dx, acc
```

Acc jest, jak poprzednio, rejestrem akumulatora (AL lub AX). Natomiast numer portu musi być umieszczony w rejestrze DX.
Ponieważ instrukcje wejścia/wyjścia służą do komunikacji ze światem zewnętrznym, to trochę uwagi poświęcimy zależnościom czasowym. Rozkaz IN pobiera daną, która jest w porcie, w czasie, gdy jest on wykonywany. Możliwe jest, że nie będzie to dana właściwa. Różne urządzenia stosują różne metody informowania o gotowości do przesłania danych. Jedną z nich jest umieszczenie w porcie wartości zero, która oznacza brak danych do odczytu. Technika ta może być stosowana wtedy, gdy zero nie jest poprawną wartością. Poniższa pętla czeka na pojawienie się danych:

```
iloop:
   in     al,123h            ; Odczytaj port
   or     al, al             ; Ustaw znaczniki
   jz     iloop              ; Kontynuuj pętlę, jeśli
                             ; dana jest 0
```

Podobnym sposobem jest wyzerowanie najstarszego bitu portu, aż do pojawienia się właściwej danej. W tym przypadku możesz również użyć powyższej pętli, zmieniając tylko warunek skoku na JNS (skocz, gdy dodatni).
Bardzo często stosowaną metodą jest zastosowanie rejestru statusu urządzenia. Program tak długo oczekuje w pętli, aż status urządzenia zezwoli na przesłanie danej.
Niektóre urządzenia wykorzystują rejestr indeksowy, by przesłać różne dane tym samym portem. Na przykład układy zegara czasu rzeczywistego mają wiele rejestrów, które są dostępne poprzez ten sam port. Najpierw program musi poinformować zegar, którego rejestru zamierza użyć. Dopiero teraz może nastąpić przesłanie danej. Często urządzenie potrzebuje pewnego opóźnienia między ustawieniem rejestru indeksowego, a odczytem lub zapisem danej. Najprostszym sposobem wstawienia krótkiej przerwy jest użycie kilku instrukcji JMP $+2. Poniższy przykład pokazuje, jak odczytać rejestr 10h układu zegara czasu rzeczywistego:

```
   mov    al, 10h            ; Numer żądanego rejestru
   out    70h, al            ; Wyślij ten numer do
                             ; portu indeksowego
   jmp    $+2                ; opóźnienie
   jmp    $+2                ; opóźnienie
   in     al, 71h            ; Odczyt portu danych
```

Wejście/wyjście w procesorze 80386

Oprócz możliwości korzystania z 32-bitowych rejestrów w instrukcjach wejścia/wyjścia procesor 80386 posiada dwa specjalne, łańcuchowe rozkazy wejścia/wyjścia oraz możność zablokowania dostępu do portów wejścia/wyjścia dla programów, które mają niski priorytet. Łańcuchowe instrukcje wejścia/wyjścia to INS i OUTS. Ich format podobny jest do pozostałych rozkazów łańcuchowych:

```
INS    [es:]przeznaczenie, DX
   INSB                      ; Odczyt bajtu
   INSW                      ; Odczyt słowa
   INSD                      ; Odczyt podwójnego słowa
```

W pierwszym formacie podaje się adres pamięci, by asembler mógł wyznaczyć rozmiar przesyłanej danej. Dane są zawsze pobierane spod adresu DS:SI, a umieszczane pod adresem ES:DI. W celu przesłania ciągu danych obie instrukcje mogą być poprzedzone przedrostkiem REP.
Jest bardzo niewiele sytuacji, kiedy mogą być one przesyłane całymi ciągami bez żadnej kontroli transmisji. Dlatego też, stosując przedrostek REP w instrukcjach wejścia/wyjścia, powinieneś być bardzo ostrożny.
Natomiast połączenie użycia instrukcji IN do sprawdzania rejestru statusowego oraz instrukcji INS do pobierania i zapamiętywania danych jest rozwiązaniem wygodnym i efektywnym. Ilustruje to poniższy przykład:

```
   mov   di, OFFSET buffer  ; Ustal adres bufora danych
   push  ds                 ; Skopiuj rejestr DS
   pop   es                 ; do ES
   mov   cx, 22             ; Liczba bajtów do
                            ; przeczytania
rloop:
   mov   dx, 123h           ; Numer portu statusowego
sloop:
   in    al, dx             ; Odczytaj ten port
   test  al, status         ; Sprawdź odpowiedni bit
   jz    sloop              ; Kontynuuj sprawdzanie
   mov   dx, 331h           ; Numer portu danych
   insb                     ; Pobierz i zapisz daną
   loop  rloop              ; Kontynuuj odczyt
```

Podsumowanie

Przed wykonaniem operacji na danej, musi być ona umieszczona w rejestrze. Jednak trzeba pamiętać, że niektóre instrukcje wykorzystują tylko pewne rejestry.
Dane w pamięci umieszczone są w segmentach. Niektóre z nich, jak segment kodu, danych i stosu, przechowują specjalne rodzaje danych. Dzięki istnieniu trybów adresowania dostęp do nich może odbywać się na wiele sposobów.
Instrukcje łańcuchowe pozwalają operować na blokach danych. W pewnych sytuacjach użycie przedrostka REP może sprawić, że cała operacja będzie wykonana w jednym rozkazie.
Procesor komunikuje się ze światem zewnętrznym za pośrednictwem instrukcji wejścia/wyjścia. Używane są one do zarządzania sprzętem, dołączonym do twojego komputera.

Input

88/86	286	386
●	●	●

IN

ZASTOSOWANIE

Wprowadzenie danej z układu wejściowego

SKŁADNIA

`IN acc, stała`	*Umieść daną z portu o numerze* ***stała*** *w akumulatorze;* ***stała*** *może mieć wartość od 0 do 255*
`IN acc, dx`	*Umieść daną z portu o numerze zawartym w DX w akumulatorze; rejestr może zawierać wartość od 0 do 65535*

ZNACZNIKI

Bez zmian

KOMENTARZ

Używając tej instrukcji należy zapoznać się z urządzeniem, z którym będziemy się komunikować. Na przykład, te same urządzenia mogą używać różnych portów na różnych komputerach. Należy także dokładnie zapoznać się z protokołem przesyłania danych.

PATRZ

OUT	*By wyprowadzić dane*
INS/OUTS	*Łańcuchowe wejście/wyjście*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
stała	1110010w port	in ax,3f8h	14/10	5	12
DX	1110110w	in al,dx	12/8	5	13

PRZYKŁAD

Poniższy przykładowy program sprawdza, czy port szeregowy zawiera nie odczytaną jeszcze wartość. Jeśli tak, to dana ta jest pobierana.

```
   mov    dx, 3f8h          ; Numer portu statusowego
                            ; dla COM1
sloop:
   in     al, dx            ; Pobierz bajt statusowy
   test   al, 40h           ; Czy jest gotowy znak
                            ; do odbioru
   jz     sloop
   add    dx,               ; Adres portu danych
   in     al, dx
```

INS/INSB/INSW/INSD

Input from Port to a String

88/86	286	386
	●	●

INS/INSB/INSW/INSD

ZASTOSOWANIE

Pobranie ciągu danych z portu wejściowego i umieszczanie ich w kolejnych komórkach pamięci, wskazywanych przez parę rejestrów ES i DI.

SKŁADNIA

`INS [es:]przeznaczenie,dx`	*Pobierz daną z portu o numerze zawartym w DX; pierwszy argument jest podawany w celu wyznaczenia rozmiaru danej*
`INSB`	*Pobierz bajt*
`INSW`	*Pobierz słowo*
`INSD`	*Pobierz podwójne słowo (tylko 80386)*

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja INS odczytuje daną z portu o numerze zawartym w DX. Następnie dana zostaje umieszczona w komórce pamięci o adresie ES:DI. Po operacji odczytu zawartość rejestru DI jest odpowiednio modyfikowana. Sposób modyfikacji zależy od rozmiaru pobieranych danych i od stanu znacznika kierunku. Rozkaz ten może być poprzedzony przedrostkiem REP.

PATRZ

IN, OUT	*Rozkazy wejścia/wyjścia*
OUTS	*Wyjście łańcuchowe*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	0110110w	insw	-	5	15

PRZYKŁAD

Odczyt 20 bajtów z portu 123h

```
mov   dx, 123h          ; Ustal adres portu
mov   cx, 20            ; Liczba odczytanych danych
rep   insb
```

Load Far Pointer to DS

88/86	286	386
●	●	●

LDS

ZASTOSOWANIE

Załadowanie dalekiego adresu do rejestru DS i podanego rejestru

SKŁADNIA

`LDS reg, pamięć` — *Pamięć zwiera daleki adres ładowany do rejestru segmentowego i do rejestru podanego w argumencie*

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcje, które ładują dalekie wskazania, upraszczają ich wykorzystanie w języku asemblera. Zamiast użycia dwóch rozkazów do przesłania numeru segmentu oraz przemieszczenia, można wykonać to samo zadanie w pojedynczym rozkazie.
Jeśli instrukcja ta używana jest na procesorze 80386, to należy znać długość przemieszczenia w segmencie. Jeśli bowiem w danym segmencie używane są przesunięcia 32-bitowe, wówczas należy również podać 32-bitowy rejestr.

PATRZ

LES, LFS, LGS, LSS — *By załadować daleki adres do innych rejestrów*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślnie DS	11000101 mod,reg,r/m	LDS bx, fpoint	24/16+EA	7	7
Domyślnie ES	11000100 mod,reg,r/m	LES di, fpoint	24/16+EA	7	7
Domyślnie FS	00001111 10110100 mod,reg,r/m	LFS edi, fpoint	-	-	7
Domyślnie GS	00001111 10110101 mod,reg,r/m	LGS dx, fpoint	-	-	7
Domyślnie SS	00001111 10110010 mod,reg,r/m	LSS bp, fpoint	-	-	7

LES

PRZYKŁAD

Przykładowy program kopiuje łańcuch, wskazywany przez FarPoint1 do łańcucha FarPoint2

```
   .MODEL LARGE
   .DATA; Segment danych
FarPoint1   DD   String1
FarPoint2   DD   String2
   .FARDATA                    ; Inny segment danych
String1     DB   "This is the source string",0
String2     DB   40 dup(?)      ; Miejsce na nowy łańcuch
   .CODE; Segment kodu
StrCpy   PROC
   cld                         ; Zwiększanie adresów
   push  ds                    ; Zachowaj bieżący segment
                               ; danych
   les   di, FarPoint2         ; Pobierz adres nowego
                               ; łańcucha
   lds   si, FarPoint1         ; Pobierz adres łańcucha
                               ; źródłowego. Instrukcja ta
                               ; musi być wykonana za
                               ; poprzedzającym ją
                               ; rozkazem. Wynika to stąd,
                               ; że same wskaźniki mogą
                               ; znajdować się w innym
                               ; segmencie niż łańcuch
                               ; wskazywany przez FarPoint1.
cloop:
   lodsb                       ; Pobierz bajt z łańcucha
                               ; źródłowego
   stosb                       ; Prześlij go do nowego
                               ; łańcucha
   cmp   al, 0                 ; Czy już koniec łańcucha
   jnz   cloop
   pop   ds
   ret
StrCpy   ENDP
```

Load Far Pointer to the ES Register

88/86	286	386
●	●	●

LES

Załaduj dalekie wskazanie do rejestru ES i rejestru podanego w argumencie.
Patrz LDS.

	88/86	286	386
Load Far Pointer to the FS Register			•

LFS

Załaduj dalekie wskazanie do rejestru FS i rejestru podanego w argumencie.
Patrz LDS.

	88/86	286	386
Load Far Pointer to the GS Register			●

LGS

Załaduj dalekie wskazanie do rejestru GS i rejestru podanego w argumencie.
Patrz LDS.

	88/86	286	386
Load Far Pointer to the SS Register			●

LSS

Załaduj dalekie wskazanie do rejestru SS i rejestru podanego w argumencie.
Patrz LDS.

	88/86	286	386
Load Effective Address	●	●	●

LEA

ZASTOSOWANIE

Załadowanie adresu efektywnego do rejestru

SKŁADNIA

`LEA reg, pamięć`

Zmienna pamięć jest komórką w pamięci, której adres efektywny będzie wyliczony i przesłany do rejestru reg.

ZNACZNIKI

Bez zmian

LEA

KOMENTARZ

Instrukcja ta może być użyta po to, aby zaoszczędzić czas potrzebny na obliczanie złożonego adresu efektywnego. Można bowiem najpierw obliczyć ten złożony adres, a następnie wykorzystać go w prostszym trybie adresowania.

PATRZ

MOV *Transmisja danych między rejestrem a pamięcią*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg,mem	10001101 mod,reg,r/m	lea bx, [si+bx+4]	2+EA	3	2

PRZYKŁAD

Procedura dodaje 20 słów w tablicy Table począwszy od indeksu SI+BX.

```
        .MODEL SMALL
        .DATA                   ; Segment danych
Table DW    100 dup(?)          ; Tablica danych
        .CODE                   ; Segment kodu
; Na wejściu procedura otrzymuje rejestry SI i BX, które
; wskazują na część tablicy poddawaną sumowaniu
AddEm PROC
        lea     bx,Table[si+bx] ; wyznacz w BX adres
                                ; efektywny pierwszego
                                ; słowa
        xor     ax,ax
        mov     cx,20
aloop:
        add     ax,[bx]         ; Dodaj słowo wskazywane
                                ; przez BX
        inc     bx              ; Następne słowo
        inc     bx
        loop    aloop
        ret                     ; Wynik sumowania w AX
AddEm ENDP
```

Load From a String

88/86	286	386
●	●	●

LODS

ZASTOSOWANIE

Załadowanie bajtu, słowa lub podwójnego słowa łańcucha do akumulatora

SKŁADNIA

`LODS [seg:]źródło`	*Argument jest podawany w celu wyznaczenia rozmiaru danej. Seg jest opcjonalnym prefiksem segmentu.*
`LODSB`	*Pobierz bajt*
`LODSW`	*Pobierz słowo*
`LODSD`	*Pobierz długie słowo (tylko 80386)*

ZNACZNIKI

Bez zmian

KOMENTARZ

Dane są pobierane z miejsca w pamięci o adresie DS:SI (chyba, że został podany prefiks). Po każdym pobraniu rejestr DI jest modyfikowany zależnie od stanu znacznika kierunku i rozmiaru danej.

Rozkaz może być poprzedzony przedrostkiem deklarującym powtórzenie, ale przydatność tego działania jest wątpliwa.

PATRZ

STOS	*Przesłanie danej do łańcucha*
MOVSB	*Kopiowanie łańcuchów*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	1010110w	lodsb	16/12	5	5

PRZYKŁAD

Podprogram wykonuje operację "XOR" na 512 bajtach bufora. Obliczenie takie jest często wykonywane, gdy sprawdzamy sumę kontrolną bloku danych.

```
            .MODEL SMALL
            .DATA
Buffer      DB  512 dup(0)          ; Miejsce na dane
            .CODE
CheckSum    PROC
            cld                     ; Zwiększanie
                                    ; adresów
```

```
            mov   si, offset Buffer ; Pobierz adres
                                    ; bufora
            mov   cx, 512           ; Rozmiar bufora
                                    ; w CX
            xor   bl, bl
xloop:
            lodsb                   ; Pobranie bajtu
            xor   bl,al             ; Wykonanie operacji
                                    ; "XOR"
            loop  xloop             ; Powtarzaj 512 razy
            ret                     ; Suma kontrolna w BL
CheckSum    ENDP
```

Move Data

88/86	286	386
●	●	●

MOV

ZASTOSOWANIE

Przesłanie wartości danej między rejestrem a pamięcią

SKŁADNIA

MOV	reg, reg	*Przesłanie danej pomiędzy rejestrami*
MOV	mem, reg	*Przesłanie danej z rejestru do pamięci*
MOV	reg, mem	*Przesłanie danej z pamięci do rejestru*
MOV	reg, stała	*Załadowanie rejestru stałą*
MOV	mem, stała	*Załadowanie pamięci stałą*
MOV	reg, seg	*Przesłanie danej z rejestru segmentowego do rejestru 16-bitowego*
MOV	mem, seg	*Przesłanie danej z rejestru segmentowego do pamięci*
MOV	seg, reg	*Przesłanie danej z rejestru 16-bitowego do rejestru segmentowego*
MOV	seg, mem	*Przesłanie danej z pamięci do rejestru segmentowego*
MOV	reg, spcl	*Przesłanie danej z rejestru specjalnego procesora 80386 do re jestru 32-bitowego*
MOV	spcl, reg	*Przesłanie danej z rejestru 32-bitowego do rejestru specjalnego procesora 80386*

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja MOV przesyła daną pomiędzy dwoma miejscami. Oba argumenty muszą być tego samego rozmiaru.

By zapewnić ochronę stosu w czasie pojawienia się przerwania, przerwania są wyłączane na czas wykonania następnego rozkazu za instrukcją, która ładuje rejestr SS. Jednak niektóre starsze wersje procesorów 8088 i 8086 nie zapewniają takiego działania.

PATRZ

MOVSX, MOVZX *Przesłanie z rejestru 16-bitowego do 32-bitowego*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	1000101w 11,reg,r/m	mov ax,bx	2	2	2
mem, reg	1000101w mod,reg,r/m przemieszczenie	mov temp, dx	13/9+EA	3	2
reg, mem	1000100w mod,reg,r/m przemieszczenie	mov ax, value	12/8+EA	5	4
reg, stała	1100011w mod,000,r/m stała	mov bx,4	4	2	2
mem, stała	1100011w mod,000,r/m przemieszczenie stała	mov data,21	14/10+EA	3	2
mem, accum	1010001w przemieszczenie	mov value, ax	14/10	3	2
accum,mem	1010000w przemieszczenie	mov ax, value	14/10	5	4
seg,reg	10001110 mod,sreg,r/m	mov ds,ax	2	2	2
seg, mem	10001110 mod,sreg,r/m przemieszczenie	mov es, extra	12/8+EA	5	5
reg, seg	10001100 mod,sreg,r/m	mov ax,ss	2	2	2
mem, seg	1000100 mod,sreg,r/m przemieszczenie	mov extra, ds	13/9+EA	3	2
reg, cntlreg	00001111 00100000 11,creg,r/m	mov eax,cr0	-	-	6
cntlreg, reg	00001111 00100010 11,creg,r/m	mov cr5,ebx	- CR0=10, CR2=4, CR3=5		

MOV

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, debugreg	00001111 00100001 11,dreg,r/m	mov edx,dr0	- DR0-3=22, DR6-7=14		
debugreg, reg	00001111 00100011 11,creg,r/m	mov dr4,eax	- DR0-3=22, DR6-7=14		
reg, testreg	00001111 00100100 11,treg,r/m	mov ecx,tr4	-	-	12
testreg, reg	00001111 00100110 11,treg,r/m	mov tr3,eax	-	-	12

PRZYKŁAD

W większości przykładów wykorzystane są instrukcje przesłania danych. Oto jeszcze jeden program przykładowy, który zapamiętuje i odtwarza rejestry AX, BX, CX, DX.

```
          .MODEL SMALL
          .DATA
RegBuf    DW   4 dup(?)          ; Miejsce na cztery
                                 ; rejestry
          .CODE
SaveRegs  PROC
          mov   [RegBuf+0], ax ; Zachowaj rejestr AX
          mov   [RegBuf+2], bx ; Zachowaj rejestr BX
          mov   [RegBuf+4], cx ; Zachowaj rejestr CX
          mov   [RegBuf+6], dx ; Zachowaj rejestr DX
          ret
SaveRegs  ENDP
RestRegs  PROC
          mov   ax, [RegBuf+0] ; Odtwórz rejestr AX
          mov   bx, [RegBuf+2] ; Odtwórz rejestr BX
          mov   cx, [RegBuf+4] ; Odtwórz rejestr CX
          mov   dx, [RegBuf+6] ; Odtwórz rejestr DX
          ret
RestRegs  ENDP
```

Move String Data

88/86	286	386
●	●	●

MOVS

ZASTOSOWANIE

Przepisuje w pamięci dane wskazywane przez DS:SI pod adres ES:DI, a następnie modyfikuje rejestry indeksowe

SKŁADNIA

```
MOVS [es:]przeznaczenie,[seg:]źródło
```

Argumenty są podawane po to, aby wyznaczyć rozmiar danej. Seg jest opcjonalnym prefiksem segmentu, w którym znajduje się źródło danych.

MOVSB	*Prześlij bajt*
MOVSW	*Prześlij słowo*
MOVSD	*Prześlij długie słowo (tylko 80386)*

ZNACZNIKI

Bez zmian

KOMENTARZ

Po przesłaniu danej rejestry indeksowe są modyfikowane odpowiednio do stanu znacznika kierunku i rozmiaru danej. Jeśli przesyłane są bajty, to rejestry zmieniane są o jeden, jeśli słowa - o dwa, a o cztery przy transmisji podwójnych słów. Poza tym, jeśli znacznik kierunku jest ustawiony, to są one zmniejszane o tę wartość, a w przeciwnym wypadku są zwiększane.

Użycie przedrostka REP powoduje skopiowanie całego bloku danych. Rejestr CX powinien wtedy zawierać określoną ilość elementów do przesłania.

PATRZ

REP	*Przedrostek deklarujący powtórzenie*
LODS	*Pobranie danej z łańcucha*
STOS	*Zapisanie danej do łańcucha*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślny	1010010w	movsw	26/18	5	7

PRZYKŁAD

Odczytanie z dysku bloku o rozmiarze 8192 bajtów jest o wiele bardziej efektywne niż odczytywanie dużej liczby małych rekordów. Poniższa procedura pokazuje, jak skopiować żądany rekord z tego bloku.

MOVS

```
            .MODEL   SMALL
            .DATA
Buffer      DB   8192 dup(?)        ; Bufor dyskowy
FileHand    DW   ?                  ; Deskryptor pliku
            .CODE
; Na wejściu procedury w rejestrze AX musi znajdować się
; numer rekordu, w CX jego rozmiar, a w DI adres, pod
; który będzie on skopiowany. Plik musi być już otwarty,
; a jego uchwyt powinien znajdować się w zmiennej FileHand.
GetRec      PROC
   mul   cx                  ; w rejestrach AX i DX
                             ; przesunięcie żądanego rekordu
                             ; w pliku
   push  ax                  ; Zachowaj młodsze słowo tego
                             ; przesunięcia
   push  cx                  ; Zachowaj rozmiar rekordu
   and   ax, 0E000h          ; Zaokrąglij do najbliższej
                             ; granicy 8K
   mov   cx, dx              ; Prześlij starsze słowo do CX
                             ; wymagane przez MS-DOS
   mov   dx, ax              ; Młodsze słowo w rejestrze DX
   mov   ax, 4200h           ; Numer funkcji systemowej, która
                             ; przesuwa wskaźnik pliku.
   int   21h                 ; Wywołanie systemu
ReadLp:
   mov   dx, OFFSET Buffer ; Wskazanie na bufor danych
   mov   cx, 8192            ; Liczba bajtów do odczytania
   mov   ah, 3Fh             ; Numer funkcji systemowej,
                             ; która odczytuje dane z pliku
   int   21h                 ; Wywołanie systemu
   pop   cx                  ; Odtworzenie rozmiaru rekordu
   pop   ax                   ; oraz młodszego słowa
                              ; przesunięcia
   and   ax, 1FFFh            ; Wyliczenie adresu rekordu
                              ; w buforze
   mov   si, OFFSET Buffer    ; Wskazanie na bufor
   add   si, ax               ; Dodanie przemieszczenia
   add   ax, cx               ; Obliczenie adresu końca
                              ; rekordu
   cmp   ax, 8192             ; Czy cały rekord zmieścił się
                              ; w buforze
   ja    skip1
   sub   ax, ax               ; Wyzeruj AX
   jmp   SHORT skip2
skip1:
   sub   ax, 8192             ; Sprawdź, ile bajtów pozostało
```

```
   sub    cx, ax              ; Sprawdź, ile bajtów jest
                              ; w bloku
skip2:
   cld
   rep    movsb               ; Skopiuj rekord
   or     ax, ax
   jz     done                ; Koniec jeśli przesłano już
                              ; wszystkie bajty
   push   cx
   push   ax                  ; Odłóż liczbę bajtów do
                              ; przeczytania
   jmp    ReadLp              ; Odczytaj następny blok
done:
   ret
GetRec     ENDP
```

Move and Sign Extend — 88/86 286 386 ●

MOVSX

ZASTOSOWANIE

Rozszerzenie 8- lub 16-bitowej danej ze znakiem na daną 16- lub 32-bitową

SKŁADNIA

`MOVSX reg, reg`	*Przesłanie między rejestrami*
`MOVSX reg, mem`	*Przesłanie między rejestrem a pamięcią*

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja ta jest bardziej uniwersalna niż rozkaz CBW, ponieważ jej argumentem może być dowolny rejestr, podczas gdy CBW operuje tylko na akumulatorze AL i AX.

PATRZ

CBW, CWD, CDQ, CWDE	*Rozszerzanie danej ze znakiem*
MOVZX	*Rozszerzenie z uzupełnieniem zerami*

MOVSX

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	00001111 1011111w mod,reg,r/m	movsx eax, cx	-	-	3
reg, mem	00001111 1011111w mod,reg,r/m przemieszczenie	movsx ax, signb	-	-	6

PRZYKŁAD

Poniższa procedura sumuje liczby, które znajdują się w tablicy, a wynik umieszcza w rejestrze AX. Wszystkie elementy tej struktury leżą w zakresie od -128 do 127. Dla zaoszczędzenia pamięci tablica ta składa się z bajtów, a instrukcja MOVSX użyta jest do rozszerzenia bajtów na słowa. Jeśli liczby leżałyby w zakresie 0..255 należałoby użyć rozkazu MOVZX.

```
                .MODEL SMALL
                .386                        ; Udostępniane są
                                            ; instrukcje
                                            ; procesora 80386
                .DATA
Table           DB   128 dup(?)
                .CODE
SumUp           PROC
                mov   cx, 128               ; Rozmiar tablicy
                sub   ax, ax                ; Wyzeruj AX
                mov   si OFFSET Table       ; Adres pierwszego
                                            ; elementu
SLoop:
                movsx bx, BYTE PTR [si]     ; Prześlij bajt
                                            ; rozszerzając go
                                            ; ze znakiem
                add   ax, bx                ; Dodaj następny
                                            ; element
                loop  SLoop
                ret
SumUp           ENDP
```

Move and Zero Extend 88/86 286 386 ●

MOVZX

ZASTOSOWANIE

Rozszerzenie 8- lub 16-bitowej danej na daną 16- lub 32-bitową z uzupełnieniem zerami

SKŁADNIA

`MOVZX reg, reg`	*Przesłanie między rejestrami*
`MOVSX reg, mem`	*Przesłanie między rejestrem a pamięcią*

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja ta używana jest do przesłania danej z mniejszego do większego rejestru. Dzięki niej nie musisz sam uzupełniać zerem starszej części rejestru docelowego. Ponieważ nie ma dostępu do obu słów rejestru 32-bitowego, nie byłoby to zadanie proste.

PATRZ

MOVZX *Rozszerzenie z uzupełnieniem zerami*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	00001111 1011011w mod,reg,r/m	mov zx ax, bl	-	-	3
reg, mem	00001111 1011011w mod,reg,r/m przemieszczenie	movsx ecx, count	-	-	6

PRZYKŁAD

Patrz MOVSX

Output Data

88/86	286	386
●	●	●

OUT

ZASTOSOWANIE

Wyprowadzenie bajtu, słowa lub podwójnego słowa do układu wyjściowego

SKŁADNIA

`OUT port, acc`	*Wyślij daną z akumulatora (AL, AX LUB EAX) do podanego portu*
`OUT dx, acc`	*Wyślij daną z akumulatora do portu o numerze, który znajduje się w rejestrze DX*

ZNACZNIKI

Bez zmian

KOMENTARZ

Używając tej instrukcji należy zapoznać się z urządzeniem, z którym będziemy się komunikować. Pisząc programy bezpośrednio operujące na sprzęcie powinieneś zapoznać się z dokumentacją, która dotyczy twojego komputera,.
W trybie chronionym użycie tej instrukcji, gdy priorytet operacji wejścia/wyjścia jest większy niż priorytet programu, spowoduje powstanie błędu.

PATRZ

OUTS	*Wyprowadzanie łańcuchowe*
IN/INS	*Instrukcje wejściowe*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
port, accum	1110011w port	out 20h,al	14/10	3	10
DX,accum	1110111w	out dx,ax	12/8	3	11

PRZYKŁAD

W wielu komputerach zgodnych z IBM PC układ zegara czasu rzeczywistego ma kilka rejestrów, które są dostępne dla programisty. By móc się do nich odwołać, należy zapisać numer żądanego rejestru do portu indeksowego.

```
; Poniższy kod wysyła 4 starsze i młodsze bity rejestru
; BL do rejestrów 10h i 11h układu RTC. Port indeksowy
; znajduje się pod adresem 70h, a port danych 71h.
    mov   al, 10h          ; Pierwszy rejestr układu RTC
```

```
out   70h, al
mov   al, bl          ; Pobierz daną
and   al, 0fh         ; Maskuj dolne 4 bity
out   71h, al         ; Wyślij daną
mov   al, 11h         ; Następny port układu RTC
out   70h, al
mov   al, bl
shr   al, 1           ; Przesuń starsze 4 bity do
                      ; młodszej części bajtu
shr   al, 1
shr   al, 1
shr   al, 1
out   71h, al         ; Wyślij bajt
```

Output a String

88/86	286	386
	●	●

OUTS

ZASTOSOWANIE

Wysłanie danej z miejsca pamięci wskazywanego przez DS:SI, a następnie aktualizacja rejestru indeksowego

SKŁADNIA

`OUTS dx, [seg:]źródło`	*Argument jest podawany w celu wyznaczenia rozmiaru danej. Seg jest opcjonalnym prefiksem segmentu.*
`OUTSB`	*Wyślij bajt*
`OUTSW`	*Wyślij słowo*
`OUTSD`	*Wyślij podwójne słowo (tylko 80386)*

ZNACZNIKI

Bez zmian

KOMENTARZ

Przy pobieraniu danej do wysłania instrukcja ta używa rejestru SI. Po dokonaniu operacji rejestr jest modyfikowany odpowiednio do stanu znacznika kierunku i rozmiaru danej. Instrukcja ta może być również poprzedzona przedrostkiem REP. Pozwala to na wysłanie całego łańcucha danych. Jednak istnieje bardzo niewiele urządzeń, które potrafią odebrać taki strumień danych.

PATRZ

OUT	*Wysłanie do portu wartości, która znajduje się w akumulatorze*
IN, INS	*Instrukcje wejściowe*

OUTS

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	0110111w	OUTSB	-	5	14

PRZYKŁAD

Instrukcja OUTS może być wykorzystana do wysłania do portu szeregowego ciągu znaków. Jednak przed każdą operacją wysłania powinien być sprawdzany stan układu, w celu stwierdzenia czy jest on gotowy na przyjęcie następnej danej.

```
; Wyślij ciąg znaków, który znajduje się w zmiennej
; SerialData, do portu szeregowego. Pierwszy znak tego
; łańcucha zawiera liczbę transmitowanych bajtów.
; Przed wysłaniem danej do portu 3f8h sprawdzany jest
; port statusowy układu o adresie 3f9h.
SendStr   PROC
          cld
          mov   si,  OFFSET SerialData
          lodsb              ; Pobranie długości łańcucha
          mov   cl, al       ; Przepisanie jej do
                             ; rejestru licznikowego
          xor   ch, ch
Sloop:
          mov   dx, 3f9h     ; Port statusowy
Tloop:
         in    al, dx        ; Pobranie bajtu statusu
         test  al, 8         ; Czy można wysłać następną
                             ; daną
         jz    Tloop         ; Jeśli nie gotowy,
                             ; kontynuuj testowanie
         dec   dx            ; Ustal w DX adres portu
                             ; danych
         outsb               ; Wyślij znak
         loop  Sloop
         ret
SendStr  ENDP
```

Pop Data Off of the Stack

88/86	286	386
●	●	●

POP

ZASTOSOWANIE

Pobranie słowa lub podwójnego słowa (tylko 80386) ze szczytu stosu, a następnie modyfikacja wskaźnika stosu.

SKŁADNIA

`pop reg/mem` — *Pobranie danej ze szczytu stosu do rejestru albo miejsca w pamięci*

ZNACZNIKI

Bez zmian

KOMENTARZ

Stos jest używany do przechowania zmiennych tymczasowych oraz adresów powrotu z funkcji. Jedynie w trybie chronionym sprawdzane jest jego przepełnienie. Programista powinien więc upewnić się, czy na stosie jest wystarczająco dużo miejsca dla potrzeb programu.

PATRZ

POPA, POPF	*Inne instrukcje do pobierania elementów ze stosu*
PUSH, PUSHA, PUSHD	*Umieszczanie danych na stosie*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg	01011reg	pop ax	12/8	5	4
mem	10001111 mod,000,r/m przesunięcie	pop data	25/17+EA	5	5
segreg	000sreg111	pop es	12/8	5	7
80386 segreg	00001111 10,sreg,001	pop fs	-	-	7

PRZYKŁAD

Stos jest często wykorzystywany do zapamiętywania wartości rejestrów. Mogą one być następnie użyte do innych celów. Jest to szczególnie ważne w wypadku rejestrów, które mają specjalne zastosowanie.

```
; Poniższa procedura przepisuje łańcuch Source do Dest.
; Rozmiar łańcucha jest podany w jego pierwszym bajcie.
; Rejestry CX, SI, DI i ES są odkładane na stos
```

```
; i zdejmowane z niego po dokonaniu przesłania.
SafeCpy  PROC
         push  cx                  ; Zachowaj rejestr CX
         push  si                  ; Zachowaj rejestr SI
         push  di                  ; Zachowaj rejestr DI
         push  es                  ; Zachowaj rejestr ES
         mov   si, offset Source
         mov   di, offset Dest
         push  ds                  ; Skopiuj rejestr DS
         pop   es                  ; do ES
         mov   cl, [si]            ; Pobierz długość
                                   ; łańcucha
         xor   ch, ch
         cld
         rep   movsb               ; Skopiuj łańcuch
         pop   es                  ; Pobierz ze stosu
                                   ; zapamiętane rejestry
                                   ; w kolejności
         pop   di                  ; odwrotnej niż były
                                   ; one odkładane
                                   ; na stos
         pop   si
         pop   cx
         ret
SafeCpy  ENDP
```

Pop All Registers	88/86	286	386
		●	●

POPA, POPAD

ZASTOSOWANIE

Pobranie ze stosu rejestrów DI, SI, BP, SP, DX, CX, BX, AX.

SKŁADNIA

POPA	*Pobranie rejestrów 16-bitowych*
POPAD	*Pobranie rejestrów 32-bitowych (tylko 80386)*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja ta pozwala na zaoszczędzenie czasu i pamięci wtedy, gdy istnieje potrzeba odtworzenia wszystkich rejestrów.

Mimo że wartość rejestru SP jest również umieszczona na stosie, to jest ona ignorowana przez ten rozkaz.

PATRZ

PUSHA	*Zapisanie na stos wszystkich rejestrów*
POP, POPF	*Inne instrukcje do pobierania elementów ze stosu*
PUSH, PUSHF	*Inne instrukcje do umieszczania elementów na stosie*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	01100001	popa	-	19	24

PRZYKŁAD

Podprogramy, np. funkcje systemowe, które korzystają z wielu rejestrów, a nic nie wiedzą o programie, który je wywołuje, muszą zachować wszystkie rejestry. Zapobiega to wzajemnemu oddziaływaniu podprogramów na siebie. Instrukcje PUSHA i POPA sprawiają, że jest to bardzo prosta operacja.

```
; Początek i koniec procedury obsługi przerwania
Routine:
        pusha               ; Zachowaj wszystkie rejestry
        .
        .
        .
        popa                ; Odtwórz ich wartości
        iret
```

Pop the Flags Register

88/86	286	386
●	●	●

POPF/POPFD

ZASTOSOWANIE

Pobranie znaczników ze szczytu stosu

SKŁADNIA

POPF	*Pobranie wartości 16-bitowej*
POPFD	*Pobranie wartości 32-bitowej*

ZNACZNIKI

Znaczniki ustawiane są według wartości zdjętej ze stosu

POPF/POPFD

KOMENTARZ

Rozmieszczenie flag w rejestrze znaczników jest następujące:

Znacznik	Numer bitu w słowie
Znacznik przeniesienia	0
Znacznik parzystości	2
Znacznik przeniesienia pomocniczego	4
Znacznik zera	6
Znacznik znaku	7
Znacznik pracy krokowej	8
Znacznik zezwolenia na przerwanie	9
Znacznik kierunku	10
Znacznik nadmiaru	11

PATRZ

PUSHF, PUSHFD	*Zapisanie na stos rejestru znaczników*
POP, POPA, POPAD	*Inne instrukcje do pobierania elementów ze stosu*
PUSH, PUSHA, PUSHAD	*Inne instrukcje do umieszczania elementów na stosie*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011101	popf	12/8	5	5

PRZYKŁAD

Instrukcje PUSHF/POPF mogą być użyte do zapamiętania wyniku operacji arytmetycznej lub logicznej w celu jej późniejszego wykorzystania.

```
; Porównaj rejestr AX z liczbą 21. Wykonaj pozostałe
; obliczenia. Być może korzystają one z AX i zmieniają
; stan znaczników. Następnie, jeśli rejestr AX był
; mniejszy niż 21 dodaj do niego 15.
     cmp   ax, 21
     pushf              ; Zapisz rezultat porównania
     .                  ; Wykonaj pozostałe obliczenia
     .
     .
     popf               ; Odtwórz rejestr
                        ; znaczników
     jge   skip         ; Nie dodawaj, jeśli AX
                        ; był większy niż 21
     add   ax, 15
skip:
```

Push Data on the Stack

88/86	286	386
●	●	●

PUSH

ZASTOSOWANIE

Przesłanie słowa na szczyt stosu

SKŁADNIA

`PUSH reg/mem`	*Prześlij 16-bitową wartość z rejestru lub pamięci na szczyt stosu*
`PUSH stała`	*Prześlij stałą na stos*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Stos jest używany do przechowania zmiennych tymczasowych oraz adresów powrotu z funkcji. Jedynie w trybie chronionym sprawdzane jest jego przepełnienie. Programista powinien więc upewnić się, czy na stosie jest wystarczająco dużo miejsca dla potrzeb programu.

W procesorach 8088 i 8086 instrukcja PUSH SP najpierw zmniejsza wartość rejestru SP i dopiero tę wartość umieszcza na stosie. Procesory 80286 i 80386 przesyłają natomiast zawartość SP przed dekrementacją.

PATRZ

POP	*Pobranie danej ze szczytu stosu*
PUSHF, PUSHFD, PUSHA, PUSHAD	*Inne instrukcje do umieszczania elementów na stosie*
POPF, POPFD, POPA, POPAD	*Inne instrukcje do pobierania elementów ze stosu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg	01010reg	push ax	15/11	3	2
mem	11111111 mod,110,r/m przesunięcie	push data	24/16+EA	5	5
segreg	00sreg110	push es	14/10	3	2
80386 segreg	00001111 10,sreg,000	push gs	-	-	2
stała	011010s0	push 12	-	3	2

PUSHA/PUSHAD

PRZYKŁAD

Jednym ze sposobów skopiowania rejestru segmentowego jest przesłanie jego wartości na stos, a następnie pobranie jej do innego rejestru. Zaletą tej metody jest to, że nie wymaga ona użycia żadnego dodatkowego rejestru ani miejsca w pamięci.

```
; Używając stosu prześlij zawartość rejestru DS do ES
    .
    .
    .
    push  ds        ; Zapisz rejestr DS na stosie
    pop   es        ; Odczytaj tę wartość do rejestru ES
                    ; teraz ES=DS
    .
    .
    .
```

Push All Register — 88/86 | 286 ● | 386 ●

PUSHA/PUSHAD

ZASTOSOWANIE

Przesłanie na stos rejestrów DI, SI, BP, SP, DX, CX, BX, AX.

SKŁADNIA

PUSHA	*Przesłanie rejestrów 16-bitowych*
PUSHAD	*Przesłanie rejestrów 32-bitowych (tylko 80386)*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Podprogramy, np. funkcje systemowe, które korzystają z wielu rejestrów, a nic nie wiedzą o programie, który je wywołuje, mogą użyć tej instrukcji do ochrony zawartości rejestrów.

PATRZ

POPA, POPAD	*Pobranie ze stosu wszystkich rejestrów*
PUSH, PUSHF, PUSHFD	*Inne instrukcje do umieszczania elementów na stosie*
POP, POPF, POPFD	*Inne instrukcje do pobierania elementów ze stosu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	01100000	pusha	-	17	18

PRZYKŁAD

Po przesłaniu wszystkich rejestrów na stos, można odwoływać się do ich wartości. Wystarczy w tym celu załadować rejestr BP wartością SP i wykorzystać bazowy tryb adresowania z przemieszczeniem. Po wykonaniu instrukcji POPA do rejestrów będą wpisane zmodyfikowane dane.

```
; Procedura zachowuje na stosie wszystkie rejestry.
; Następnie w miejsce, które odpowiada rejestrowi
; AX wpisuje wartość 1. Taką też wartość przyjmie
; ten rejestr po wyjściu z tego podprogramu.
  pusha                        ; Zachowaj wszystkie rejestry
  mov   bp, sp
  mov   word ptr [bp+14], 1    ; Prześlij w miejsce
                               ; odpowiadające położeniu
  .                            ; rejestru AX wartość 1
  .
  .
  popa                         ; Odtwórz wszystkie rejestry
  ret
```

Push the Flags

88/86	286	386
●	●	●

PUSHF/PUSHFD

ZASTOSOWANIE

Przesłanie znaczników na szczyt stosu

SKŁADNIA

```
PUSHF
PUSHFD
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

W procesorze 80386 dodano dwa nowe znaczniki. Dlatego też rejestr flag musiał zostać rozszerzony do 32 bitów. Instrukcja PUSHFD zachowuje całe słowo stanu procesora, a PUSHF tylko znaczniki, które były w poprzednich układach.

REP

PATRZ

POPF, POPFD	*Pobranie ze stosu rejestru znaczników*
POP, POPA, POPAD	*Inne instrukcje do pobierania elementów ze stosu*
PUSH, PUSHA, PUSHAD	*Inne instrukcje do umieszczania elementów na stosie*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011100	pushf	14/10	3	4

PRZYKŁAD

Jednym z zastosowań instrukcji PUSHF jest symulacja przerwania. Po pojawieniu się przerwania, na stos jest odkładany rejestr znaczników, następnie rejestry CS i IP, oraz wywoływana jest procedura obsługi tego przerwania. Niektóre programy przechwytują przerwania. Najpierw uruchamiają oryginalny podprogram obsługi, a następnie wykonują swoje własne czynności. By jednak wywołać procedurę obsługi, należy najpierw przesłać na stos rejestr znaczników. Instrukcja PUSHF, która poprzedza rozkaz CALL, ma taki sam skutek jak pojawienie się przerwania.

```
; Procedura przechwytuje przerwania wywołane
; wciśnięciem klawisza. Adres oryginalnej
; procedury obsługi tego przerwania znajduje się
; w zmiennej OldKeys
NewKey   PROC   FAR
   pushf                        ; Prześlij na stos rejestr flag
                                ;w celu zasymulowania
                                ; przerwania
   call   DWORD cs:[OldKeys]    ; Wywołaj funkcję,której adres
                                ; znajduje się w zmiennej
                                ; OldKeys
   .                            ; Obsłuż przerwanie
   .
   iret                         ; Powrót z przerwania
```

Repeat String

88/86	286	386
●	●	●

REP

ZASTOSOWANIE

Zmniejszenie o jeden zawartości rejestru CX, a następnie wykonanie instrukcji, która znajduje się za tym przedrostkiem. Proces odbywa się aż do wyzerowania rejestru CX.

SKŁADNIA

```
REP   instrukcja
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Przedrostek deklarujący powtórzenie może poprzedzać instrukcje MOVS i STOS. Może on być również użyty wraz z rozkazem LODS, ale przydatność tego działania jest wątpliwa. Na procesorach 80286 i 80386 przedrostek ten może także poprzedzać instrukcje INS oraz OUTS.

PATRZ

REPNE, REPE — *Powtarzaj, gdy rejestr CX i znacznik zera*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
rep MOVS	11110010 1010010w	rep movsb	9+25/17n	5+4n	8+4n
rep STOS	11110010 1010101w	rep stosw	9+14/10n	4+3n	8+4n
rep INS	11110010 0110110w	rep insb	-	5+4n	13+6n
rep OUTS	11110010 0110111w	rep outsb	-	5+4n	12+5n

PRZYKŁAD

Patrz przykład dla MOVSB

Set a Byte Conditionally — 88/86 286 386 ●

SET Warunek

ZASTOSOWANIE

Ustawienie podanego rejestru lub miejsca w pamięci na wartość 1, jeśli warunek jest spełniony, lub wstawienie tam 0, jeśli warunek nie jest spełniony.

SKŁADNIA

SETAE reg/mem — *Ustaw rejestr lub pamięć, gdy CF=0*
SETNB reg/mem — *Ustaw rejestr lub pamięć, gdy CF=0*

SET Warunek

`SETBE reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=1 lub ZF=1*
`SETNA reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=1 lub ZF=1*
`SETA reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=0 lub ZF=0*
`SETNBE reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=0 lub ZF=0*
`SETE reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=1*
`SETZ reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=1*
`SETNE reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=0*
`SETNZ reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=0*
`SETL reg/mem`	*Ustaw rejestr lub pamięć, gdy SF<>OF*
`SETNGE reg/mem`	*Ustaw rejestr lub pamięć, gdy SF<>OF*
`SETGE reg/mem`	*Ustaw rejestr lub pamięć, gdy SF=OF*
`SETNL reg/mem`	*Ustaw rejestr lub pamięć, gdy SF=OF*
`SETLE reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=1 lub SF<>OF*
`SETNG reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=1 lub SF<>OF*
`SETG reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=0 lub SF=OF*
`SETNLE reg/mem`	*Ustaw rejestr lub pamięć, gdy ZF=0 lub SF=OF*
`SETS reg/mem`	*Ustaw rejestr lub pamięć, gdy SF=1*
`SETNS reg/mem`	*Ustaw rejestr lub pamięć, gdy SF=0*
`SETC reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=1*
`SETNC reg/mem`	*Ustaw rejestr lub pamięć, gdy CF=0*
`SETO reg/mem`	*Ustaw rejestr lub pamięć, gdy OF=1*
`SETNO reg/mem`	*Ustaw rejestr lub pamięć, gdy OF=0*
`SETP reg/mem`	*Ustaw rejestr lub pamięć, gdy PF=1*
`SETPE reg/mem`	*Ustaw rejestr lub pamięć, gdy PF=1*
`SETNP reg/mem`	*Ustaw rejestr lub pamięć, gdy PF=0*
`SETPO reg/mem`	*Ustaw rejestr lub pamięć, gdy PF=0*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja ta jest podobna do rozkazów skoków warunkowych. Jednakże zamiast wykonania (lub nie) skoku, instrukcja ustawia wartość argumentu na 1 lub 0, w zależności od spełnienia warunku.

Instrukcja SET ułatwia programowanie złożonych wyrażeń warunkowych. Na przykład, by sprawdzić, czy wartość rejestru AX mieści się w podanym zakresie, należałoby użyć wyrażenia:

```
AX > LowLimit AND AX < HighLimit
```

Wartość tego wyrażenia można obliczyć na dwa sposoby. Jeden polega na dokonaniu pierwszego porównania (AX > LowLimit) i jeśli jest ono prawdziwe na wykonaniu drugiego (AX < HighLimit). Metoda ta jest dobra w wypadku wyrażeń prostych, natomiast przy bardziej złożonych warunkach może powodować bałagan. Dzięki instrukcji SET można natomiast zapamiętać wynik każdego porównania w oddzielnym bajcie, a końcowy rezultat otrzymamy po przeprowadzeniu na nich operacji AND.

PATRZ

Jwarunek *Skoki warunkowe*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg	00001111 1001cond mod,000,r/m	setb al	-	-	4
mem	00001111 1001cond mod,000,r/m	setcflag	-	-	5

PRZYKŁAD

Procedura sprawdza, czy wartość rejestru AX znajduje się w zakresie wyznaczonym przez zmienne LowLimit i HighLimit. Zwraca 1, jeśli tak i 0 w przeciwnym wypadku.

```
            .MODEL SMALL
            .DATA
LowLimit    DW  ?
HighLimit   DW  ?
Flag1       DB  ?          ; Ustawiany na 1, gdy AX > LowLimit
Flag2       DB  ?          ; Ustawiany na 1, gdy AX < HighLimit
            .CODE
LimTest     PROC
   cmp   ax, LowLimit      ; Porównaj rejestr AX
                           ; z ograniczeniem dolnym
   seta  Flag1             ; Ustaw Flag1, jeśli AX > LowLimit
   cmp   ax, HighLimit     ; Porównaj rejestr AX
                           ; z ograniczeniem górnym
   seta  Flag1             ; Ustaw Flag2, jeśli AX < HighLimit
   mov   ax, Flag1         ; Wynik pierwszego porównania w AX
   and   ax, Flag2         ; Koniunkcja z drugim rezultatem
   ret                     ; Wynik końcowy w AX
LimTest     ENDP
```

STOS

Store to a String

88/86	286	386
●	●	●

STOS

ZASTOSOWANIE

Przesłanie z akumulatora danej do miejsca w pamięci wskazywanego przez rejestry ES i DI, a następnie modyfikacja zawartości rejestru DI.

SKŁADNIA

`STOS [es:]przeznaczenie`	*Argument jest podawany w celu wyznaczenia rozmiaru danej.*
`STOSB`	*Prześlij bajt*
`STOSW`	*Prześlij słowo*
`STOSD`	*Prześlij podwójne słowo (tylko 80386)*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja STOS może być poprzedzona przedrostkiem REP. Jest to wygodne przy wypełnianiu obszaru pamięci daną wartością.

PATRZ

CLD, STD	*Ustawianie znacznika kierunku*
LODS	*Pobranie danej z łańcucha*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	1010101w	stosb	15/11	3	4

PRZYKŁAD

Instrukcja STOS używana jest do wypełniania obszaru pamięci stałą wartością. Przesyłanie dwóch bajtów naraz jest rozwiązaniem bardziej efektywnym.

```
; Przesłanie 20 zer do obszaru danych DataBlk.
; Dla zaoszczędzenia czasu przesyłanych jest 10 zerowych
; słów, zamiast 20 zerowych bajtów.
Set0    PROC
        cld
        mov   di, OFFSET DataBlk
        mov   ax, SEG DataBlk
        mov   es, ax
        xor   ax, ax          ; AX=0
        mov   cx, 10          ; Liczba przesyłanych słów
```

```
        rep   stosw          ; Wypełnienie zerami
        ret
Set0    ENDP
```

Exchange Data Values

88/86	286	386
●	●	●

XCHG

ZASTOSOWANIE

Zamiana miejscami danych

SKŁADNIA

```
XCHG reg/mem, mem/reg
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja XCHG przeprowadza operację, która przy użyciu instrukcji MOV zajęłaby trzy rozkazy, a ponadto wymagałaby użycia dodatkowego rejestru. Na przykład, zamiana zawartości rejestrów AX i BX przy wykorzystaniu instrukcji przesłania mogłaby być zrealizowana w następujący sposób:

```
mov   cx, ax          ; Zapisz rejestr AX do
                      ; rejestru pomocniczego
mov   ax, bx          ; Prześlij BX do AX
mov   bx, cx          ; Skopiuj wartość z rejestru
                      ; pomocniczego do BX
```

PATRZ

MOV	*Przesyłanie danych*
XLAT	*Pobranie danej z tablicy*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	1000011w 11,reg,r/m	xchg cx, si	4	3	3
mem, reg	1000011w mod,reg,r/m przesunięcie	xchg Numb, si	17+EA	5	5
accum, reg	10010reg	xchg ax, bx	3	3	3

XLAT/XLATB

PRZYKŁAD

Procesory 80x86 przechowują dane w pamięci, umieszczając pod mniejszymi adresami mniej znaczące bajty. Inne natomiast np. 68000 umieszczają jako pierwszy bajt najbardziej znaczący. Poniższy przykład pokazuje, jak przekształcić obszar danych, zapisany w jednym formacie, w drugi. Uruchomienie drugi raz tej funkcji na tej samej tablicy spowoduje przywrócenie jej poprzedniej postaci.

```
          .MODEL SMALL
          .CODE
; Zamień słowa w tablicy wskazywanej przez DS:SI.
; Ich liczba znajduje się w rejestrze CX.
FlipEm    PROC
          mov   ax, WORD PTR [si] ; Pobierz słowo
          xchg  ah, al
          mov   WORD PTR [si], ax ; Zapisz zamieniony
                                  ; element
          inc   si                ; Następne słowo
          inc   si
          loop  FlipEm
          ret
FlipEm    ENDP
```

Translate Data — 88/86 ● 286 ● 386 ●

XLAT/XLATB

ZASTOSOWANIE

Zwrócenie wartości elementu tablicy wskazywanej przez DS:BX o indeksie zawartym w AL.

SKŁADNIA

`XLAT [[seg:]pamięć]`	*Wykorzystywany jest tylko prefiks segmentu. Tablica znajduje się pod adresem seg:BX.*
`XLATB [[seg:]pamięć]`	*Jest to synonim XLAT*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja ta jest użyteczna przy przekształcaniu jednego systemu kodowania w drugi.

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	11010111	xlat	11	5	5

PRZYKŁAD

Instrukcja XLAT może być użyta do odkodowania danych. Wykorzystana została tablica wypełniona odpowiednio wszystkimi znakami ASCII. Dzięki niej można dokonywać podstawień.

```
; Tablica CodeTable zawiera sposób kodowania znaków.
; Znak do odkodowania znajduje się w rejestrze AL.
        .DATA
CodeTable  DB  061, 010, 107, 114, 021, 110, 097, 066
           DB  009, 002, 098, 123, 068, 119, 023, 109
           DB  065, 025, 081, 050, 005, 077, 091, 070
           DB  018, 063, 027, 054, 029, 085, 031, 115
           DB  033, 112, 024, 015, 090, 038, 039, 040
           DB  041, 127, 043, 044, 045, 111, 047, 080
           DB  113, 020, 051, 124, 053, 028, 055, 056
           DB  057, 120, 059, 017, 001, 062, 026, 106
           DB  060, 008, 125, 013, 069, 035, 071, 072
           DB  121, 074, 075, 076, 022, 078, 079, 048
           DB  019, 082, 083, 099, 030, 086, 128, 088
           DB  089, 037, 036, 092, 093, 094, 095, 096
           DB  007, 011, 084, 100, 101, 102, 103, 104
           DB  105, 064, 003, 108, 016, 006, 046, 034
           DB  049, 004, 032, 126, 117, 118, 114, 058
           DB  073, 012, 123, 052, 067, 116, 042, 087
Encrypt    PROC
    mov    bx, OFFSET CodeTable    ; Wskazanie na tablicę
                                   ; konwersji w BX
    xlat                           ; Dokonanie konwersji
    ret
Encrypt    ENDP
```

Rozdział 5 Instrukcje arytmetyczne i logiczne

Instrukcje arytmetyczne są używane do wyznaczania wartości wyrażeń, do obliczania adresów zmiennych, modyfikowania ramki stosu itd. Procesor 80x86 ma rozkazy dodawania, odejmowania, zwiększania i zmniejszania o jeden, mnożenia i dzielenia. Instrukcje te używają liczb, które mogą mieć różne formaty: liczby ze znakiem, liczby bez znaku, liczby dziesiętne kodowane dwójkowo. Opis tych rozkazów znajduje się w pierwszej części rozdziału.

Programy używają instrukcji logicznych do wyznaczania wartości wyrażeń logicznych, manipulowania na poszczególnych bitach, symulacji pewnych operacji arytmetycznych. Wiele z nich nie jest bezpośrednio dostępna w językach wysokiego poziomu. Właśnie dlatego są niedoceniane przez początkujących programistów. W drugiej części rozdziału przedstawiono, jak wykorzystać te instrukcje do zwiększenia efektywności pisanych programów.

Operacje arytmetyczne

Procesor 80x86 może operować na 8-, 16- lub 32-bitowych liczbach, które są umieszczone w rejestrach albo w pamięci operacyjnej. Ogólny format instrukcji arytmetycznych jest następujący:

```
add   mem/reg, reg/mem/stała
```

W rozkazie tym **mem/reg** może być rejestrem lub miejscem pamięci, a **reg/mem/stała** rejestrem, miejscem w pamięci lub daną natychmiastową. Pierwszy argument zwany jest *argumentem docelowym*. Będzie w nim umieszczony rezultat wykonanej operacji. Drugi argument dostarcza tylko wartości i jest zwany *argumentem źródłowym*. Oba natomiast muszą być takich samych rozmiarów. Nie można dodać liczby 8-bitowej do 16-bitowej. Również tylko jeden z nich może wskazywać na miejsce w pamięci. Jeśli argument docelowy jest daną w pamięci, to argument źródłowy może być tylko rejestrem lub wartością stałą. Jeśli natomiast argument źródłowy jest daną w pamięci, wówczas argument docelowy może być tylko rejestrem.

Wyświetlanie liczb

Przed wyświetleniem liczby na ekranie czy też wysłaniem jej na drukarkę, musi być ona zamieniona na łańcuch znaków ASCII. Gdy użytkownik wprowadza ją z klawiatury, to jest ona również zapamiętywana w postaci łańcucha znaków ASCII. Wtedy, przed podjęciem jakiejkolwiek operacji na niej, musi być ona zamieniona na liczbę binarną. Poniższy przykład odczytuje liczbę z klawiatury, zamienia ją na postać dwójkową, z powrotem przekształca na ciąg znaków i następnie wyświetla na ekranie. Składa się on z dwóch funkcji

(**GetNumb** oraz **PrntNumb**). Możesz je wykorzystać w swoich własnych doświadczeniach, które dotyczą operacji arytmetycznych.

```
; Pobierz i wyświetl liczby
            .MODEL   SMALL
            .DATA
Prompt      DB   0Dh, 0Ah, ">>$"
TmpSize     DB   20
TmpCnt      ?
TmpStr      20   dup(?)
            .CODE
; Procedura testuje funkcje GetNumb oraz PrntNumb
FuncTest    PROC
            mov   ax, SEG DGROUP    ; Pobierz adres
                                    ; segmentu danych
            mov   ds, ax            ; Prześlij go do DS
            mov   es, ax            ; i do ES
            call  GetNumb           ; Pobierz od
                                    ; użytkownika liczbę
            call  PrntNumb          ; Wyświetl ją
            mov   ah, 4Ch           ; Numer funkcji
                                    ; systemowej, która
                                    ; kończy program
            int   21h
FuncTest    ENDP
; Pobierz liczbę i umieść ją w AX
GetNumb     PROC
            mov   dx, OFFSET Prompt; Adres łańcucha
                                    ; zachęty
            mov   ah, 9             ; Numer funkcji
                                    ; systemowej, która
                                    ; wyświetla ciąg
                                    ; znaków
            int   21h
            mov   dx, OFFSET TmpSize; Adres bufora
                                     ; wejściowego
            mov   ah, 0Ah            ; Numer funkcji
                                     ; systemowej, która
                                     ; pobiera łańcuch
                                     ; znaków
            int   21h
            sub   ax, ax             ; Wyzeruj AX
            sub   cx, cx             ; Wyzeruj CX
            sub   bx, bx             ; Wyzeruj BX
            mov   cl, TmpCnt         ; Liczba odczytanych
                                     ; znaków
```

```
            mov   si, OFFSET TmpStr ; Adres pobranego
                                    ; łańcucha
            mov   di, 10            ; System dziesiętny
ConvLp:
            mul   di                ; Pomnóż rejestr AX
                                    ; przez 10
            mov   bl, BYTE PTR [si] ; Pobierz znak
                                    ; łańcucha
            inc   si                ; Następny znak
            and   bl, 0Fh           ; Zamień znak na cyfrę
            add   ax, bx            ; Dodaj ją do wyniku
            loop  ConvLp            ; Powtórz dla każdego
                                    ; znaku
            push  ax
            mov   dl, 0Dh           ; Wyświetl znak CR
            mov   ah, 2
            int   21h
            mov   dl, 0Ah           ; Wyświetl znak LF
            mov   ah, 2
            int   21h
            pop   ax
            ret
GetNumb     ENDP
; Wyświetl liczbę z rejestru AX na ekranie
PrntNumb    PROC
            mov   di, 10            ; System dziesiętny
            sub   cx, cx            ; Wyzeruj CX
PrntLp:
            sub   dx, dx            ; Wyzeruj DX
            div   di                ; Podziel liczbę
                                    ; przez 10
            push  dx                ; Odłóż resztę
                                    ; z dzielenia na stos
            inc   cx
            or    ax, ax            ; Czy koniec konwersji
            jnz   PrntLp
            cld
            mov   di, OFFSET TmpStr ; Adres łańcucha
StrLp:
            pop   ax
            add   al, '0'           ; Przekształć cyfrę
                                    ; na znak ASCII
            stosb                   ; Prześlij do
                                    ; łańcucha
SK1:
        loop      StrLp
```

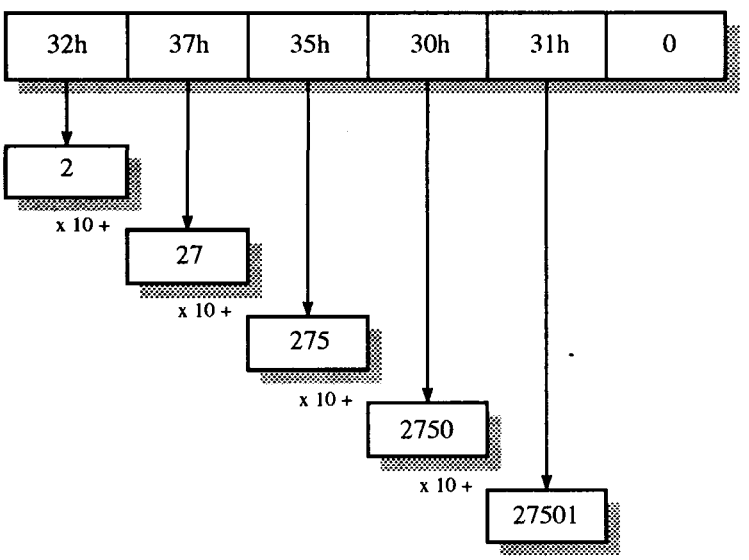
```
PNDone:
          mov   al, '$'              ; Znak końca łańcucha
          stosb
          mov   dx, OFFSET TmpStr
          mov   ah, 9
          int   21h
          ret
PrntNumb  ENDP
          .STACK   1024
END       FuncTest
```

Do wyświetlenia zachęty i pobrania danych od użytkownika funkcja **GetNumb** wykorzystuje dwie funkcje systemowe, przedstawione w Rozdziale 1. System MS-DOS umieścił odczytane znaki w zmiennej **TmpStr**.

Dalsza część podprogramu przekształca ciąg znaków w liczbę. Najpierw pobierany jest pojedynczy znak z łańcucha i transformowany na cyfrę dziesiętną. Następnie poprzednia wartość liczby mnożona jest przez 10 i dodawana jest aktualnie pobrana cyfra. Rysunek 5-1 ilustruje dokładnie ten proces.

Procedura **PrntNum**b ma odwrotne do procedury **GetNum**b działanie. Zamiast wielokrotnego mnożenia używa ona wielokrotnego dzielenia do znalezienia kolejnych cyfr. Powstałe przy tej operacji reszty są właśnie wyświetlanymi cyframi. Niestety, są one generowane w odwrotnym porządku, najpierw pojawia się najmniej znacząca, a jako ostatnia najbardziej znacząca cyfra wyniku.

Funkcja **PrntNum**b używa więc stosu do odwrócenia ich kolejności. Po wygenerowaniu następnej cyfry odkładana jest ona na stos. Po znalezieniu wszystkich liczb są one z niego pobierane. Ponieważ ostatnio położony element jest pierwszym, który będzie pobrany, powoduje to zamianę kolejności cyfr.



Rysunek 5-1. ***Konwersja znaków na liczbę całkowitą.***

Niektóre przykłady zamieszczone w tym rozdziale wykorzystują powyższe procedury do zademonstrowania działania różnych instrukcji. By uruchomić te przykłady należy tylko zastąpić podprogram **FuncTest** odpowiednim kodem.

Dodawanie

Instrukcja ADD dodaje dwa argumenty do siebie, a wynik umieszcza w pierwszym z nich. Jeśli wynik operacji nie mieści się w argumencie docelowym, to ustawiany jest znacznik przeniesienia. Rozkaz ADC dodaje do siebie dwa argumenty oraz wartość znacznika przeniesienia. Można go użyć po to, aby zsumować dane o rozmiarze większym od słowa. W poniższym przykładzie dodawane są liczby 32-bitowe:

```
; Pierwszy argument ma wartość 12345678h
    mov   ax, 5678h          ; Młodsze słowo pierwszego
                             ; argumentu
    mov   dx, 1234h          ; Starsze słowo pierwszego
                             ; argumentu
; Dodaj liczbę 87654321h do rejestrów AX i DX
    add   ax, 4321h          ; Dodaj młodsze słowa (wynik
                             ; AX=9999h); po tej
                             ; operacji CF=0
    adc   dx, 8765h          ; Dodaj starsze słowa oraz
                             ; znacznik przeniesienia
                             ; (wynik DX=9999h)
```

Odejmowanie

Istnieją dwa rozkazy odejmowania: SUB i SBB. Ich format jest taki sam jak instrukcji ADD. Instrukcja SUB odejmuje od pierwszego argumentu drugi, a wynik umieszcza w argumencie docelowym. Ustawienie znacznika przeniesienia oznacza, że konieczna była pożyczka. Tak się dzieje wtedy, gdy argument źródłowy jest większy niż argument docelowy.
Podczas obliczeń instrukcja SBB traktuje znacznik przeniesienia jako pożyczkę. Najpierw, podobnie jak w rozkazie SUB, odejmowane są wartości argumentów. Ponadto, gdy znacznik przeniesienia jest ustawiony, od wyniku odejmowana jest jeszcze jedynka. W poniższym przykładzie odejmowane są od siebie dwie 32-bitowe liczby.

```
mov   ax, 0              ; Młodsze słowo pierwszego
                         ; argumentu
mov   dx, 500h           ; Starsze słowo pierwszego
                         ; argumentu
sub   ax, 89h            ; Odjęcie młodszych słów
                         ; daje w rejestrze AX wartość
                         ; 0ff77h oraz powoduje
                         ; ustawienie znacznika
                         ; przeniesienia
sbb   dx, 488h           ; Odjęcie starszych słów daje
                         ; w wyniku 78h, a po odjęciu
                         ; znacznika przeniesienia
```

```
                                ; otrzymuje się w rejestrze
                                ; DX wartość 77h
```

Inkrementacja i dekrementacja

Dodawanie i odejmowanie może być użyte po to, aby policzyć elementy w zbiorze lub do sterowania ilością obiegów pętli. Do licznika dodawana jest wtedy jedynka, a od zmiennej sterującej pętli jest ona odejmowana. Wykorzystanie instrukcji ADD czy SUB powoduje zajęcie od dwóch do sześciu bajtów pamięci oraz trwa co najmniej cztery takty zegara. Procesor 80x86 posiada jednak efektywniejszy rozkaz INC, który służy do inkrementacji danych. Jego format jest następujący:

```
inc    reg/mem
```

Argument może być rejestrem lub miejscem w pamięci. Jeśli odnosi się on do rejestru, to zajmuje tylko jeden bajt, a jego wykonanie zajmuje trzy takty. Odjęcie jedynki może być natomiast zrealizowane instrukcją DEC.
Oba te rozkazy są najczęściej wykorzystywane do zliczania ilości elementów lub sterowania obiegami pętli. Poniższy przykład używa tych instrukcji oraz funkcji **GetNumb** i **PrntNumb** do obliczenia i wyświetlenia ilości bitów równych 1 w podanej liczbie.

```
TestFunc    PROC
            mov    ax, SEG DGROUP
            mov    ds, ax
            mov    es, ax
            call   GetNumb     ; Pobierz liczbę
            mov    cx, 16      ; Liczba bitów
            sub    bx, bx      ; Wyzeruj BX
BitLp:
            shl    ax, 1       ; Przesuń bity w słowie
                               ; wysuwając najstarszy
                               ; do znacznika CF
            jnc    SkCnt       ; Jeśli CF nie jest
                               ; ustawiony, nie zliczaj
            inc    bx
SkCnt:
            loop   BitLp       ; Przejrzyj wszystkie bity
            mov    ax, bx      ; Ilość jedynek w AX
            call   PrntNumb
            mov    ah, 4Ch
            int    21h
TestFunc    ENDP
```

Porównywanie liczb

Większość decyzji w programach podejmowana jest na podstawie porównania liczb. Funkcja może zliczać elementy tablicy, które to elementy są większe od zadanej wartości. Pętla będzie się wykonywała, aż do osiągnięcia pewnej wartości zmiennej. Procesor komend rozpoznaje polecenia porównując je z tymi, które ma zapamiętane w tablicy.

Instrukcja CMP w procesorach 80x86 dokonuje operacji odejmowania od siebie argumentów w celu ustawienia znaczników. Natomiast jej rezultat nie jest nigdzie wpisywany. Tak więc wartości obu argumentów pozostają niezmienione. Format instrukcji CMP jest taki sam, jak format rozkazu SUB. Pamiętanie o tym, że operacja porównania sprowadza się do operacji odejmowania, może ułatwić zrozumienie sposobu ustawiania znaczników. Jeśli liczby są równe, to odejmowanie da wynik zero i znacznik zera przyjmie wartość 1. Jeśli pierwsza liczba jest mniejsza, to podczas odejmowania powstaje pożyczka i ustawiane jest przeniesienie. Natomiast jeśli żaden z tych warunków nie jest spełniony, czyli gdy argument źródłowy jest większy niż docelowy, wówczas obie flagi zostają wyzerowane.
W poniższym przykładzie rejestr AL jest porównywany z liczbą 21. W sytuacji, gdy zachodzi równość, do AL wpisywane jest 0. Jeśli Al jest większy, wówczas przypisana mu zostanie wartość 1, a gdy mniejszy - -1.

```
        cmp     al, 21
        jb      less            ; Skocz, gdy AL jest
                                ; mniejszy niż 21
        ja      greater         ; Skocz, gdy AL jest
                                ; większy niż 21
        xor     al, al          ; w przeciwnym razie AL=21
        jmp     short done      ; Porównanie zakończone
less:
        mov     al, -1
        jmp     short done
greater:
        mov     al, 1
done:
```

Mnożenie i dzielenie

Operacje mnożenia i dzielenia wymagają specjalnej uwagi. Wynika to stąd, że rezultat mnożenia może mieć dwa razy większy rozmiar niż argumenty. Natomiast jeżeli dzielimy to wynik może być dwa razy mniejszy niż dzielna czy dzielnik.

MNOŻENIE

Istnieją dwie instrukcje mnożenia: MUL i IMUL. Ich format jest następujący:

```
mul     reg/mem                 ; Mnożenie liczb bez znaku
imul    reg/mem                 ; Mnożenie liczb ze znakiem
```

Reg/mem może być rejestrem lub miejscem pamięci. Należy zauważyć, że jest podawany tylko jeden argument, bowiem drugi przyjmowany jest domyślnie. Jeśli argument jest wartością 8-bitową, to drugim będzie AL, a wynik zostanie wpisany do rejestru AX. Natomiast, jeśli jest on daną 16-bitową, wówczas argumentem domyślnym stanie się AX, a rezultat zostanie przesłany do pary rejestrów **DX:AX**. Jeśli zaś na procesorze 80386 zostanie podany argument 32-bitowy, to przyjęty zostanie rejestr EAX, a wynik umieszczony w rejestrach **EDX:EAX**.

MNOŻENIE W PROCESORZE 80386

Poza możliwością mnożenia danych 32-bitowych, procesor 80386 ma kilka dodatkowych formatów instrukcji mnożenia:

```
imul   reg, stała
imul   reg1, reg/mem, stała
imul   reg/mem1, mem/reg2
```

W pierwszym z nich reg jest 16- lub 32-bitowym rejestrem, który jest również wykorzystywany do umieszczenia w nim rezultatu. Drugim argumentem jest dana natychmiastowa. Oba rejestry muszą mieć taki sam rozmiar. Jeśli rezultat nie mieści się w rejestrze, to ustawiane są znaczniki przeniesienia i nadmiaru, a tylko młodsze bity wyniku są zapamiętywane. Niestety nie ma sposobu na odzyskanie utraconych bitów.
Drugi format rozkazu powoduje pomnożenie drugiego argumentu przez stałą i umieszczenie wyniku w reg1. Podobne reguły są stosowane przy powstaniu nadmiaru.
Ostatnia instrukcja mnoży dowolny rejestr 16- lub 32-bitowy lub miejsce pamięci przez drugi argument, który ma podobną postać. Oba zaś muszą być tego samego rozmiaru i tylko jeden z nich może wskazywać na miejsce w pamięci. Jeśli wynik operacji nie mieści się w argumencie docelowym, to znaczniki przeniesienia i nadmiaru zostają ustawione, a najstarsze bity rezultatu zagubione.

DZIELENIE

Instrukcje dzielenia również używają argumentów domyślnych. Są to rozkazy DIV oraz IDIV. Argumentem obu z nich może być zarówno rejestr jak i miejsce pamięci. Może on być 8- lub 16-bitowy, a na procesorze 80386 również 32-bitowy. Po wykonaniu operacji młodsza część argumentu domyślnego zawiera wynik, a starsza - resztę z dzielenia. Jeśli argument jest 8-bitowy, to dzielna będzie w AX, a iloraz zostanie wpisany do rejestru AL, reszta zaś do AH. Natomiast, jeśli jest on daną 16-bitową, wówczas argumentem domyślnym stanie się 32-bitowa wartość umieszczona w DX i AX, a iloraz zostanie przesłany do rejestru AX, reszta zaś do DX. Dla procesora 80386, jeśli dzielnik jest 32-bitowy, zaś dzielna znajduje się w **EDX:EAX**, to wynik umieszczany jest w rejestrze EAX, a reszta w EDX.
Przed wykonaniem dzielenia należy zawsze upewnić się, czy argumenty zawierają poprawne wartości. Dzielenie przez zero lub przez wartość, powodującą powstanie nadmiaru, generuje przerwanie, które prowadzi do awaryjnego zakończenia programu.
Operacje mnożenia i dzielenia są często wykorzystywane przy obliczaniu wartości wyrażeń matematycznych. W poniższym przykładzie znajdziemy przybliżony obwód koła. Jego promień pobiera funkcja **GetNumb**. Ponieważ istnieje konieczność używania liczb całkowitych, to przyjmiemy wartość pi równą 22/7. Wynik wyświetla funkcja **PrntNumb**.

```
TestFunc   PROC
           mov    ax, SEG DGROUP
           mov    ds, ax
           mov    es, ax
           call   GetNumb        ; Pobierz promień
           mov    bx, 22         ; Jeśli nie korzystamy
                                 ; z procesora 80386, to
                                 ; musimy umieścić stałą
```

```
                                ; w rejestrze
        mul     bx              ; Pomnóż promień przez 22
                                ; Wynik znajduje się
                                ; w rejestrach DX:AX
        mov     bx, 7           ; Przygotuj się do
                                ; dzielenia
        cmp     dx, bx          ; Sprawdź poprawność
                                ; argumentów
        jge     error           ; Jeśli dx >= 7, zaniechaj
                                ; dzielenia
        div     bx              ; Wartość obwodu znajduję
                                ; się w AX; DX zawiera
                                ; część ułamkową
        push    dx
        call    PrntNumb        ; Wyświetl część całkowitą
                                ; wyniku
        mov     dl, ' '         ; Wyświetl spację
        mov     ah, 2           ; Numer funkcji systemowej,
                                ; która wyświetla znak
        int     21h
        pop     ax              ; Pobierz część ułamkową
        call    PrntNumb
        mov     dl, '/'         ; Wyświetl znak '/'
        mov     ah, 2
        int     21h
        mov     dl, '7'
        mov     ah, 2
        int     21h
error:
        mov     ah, 4Ch
        int     21h
```

Arytmetyka na liczbach ze znakiem

Przy operacji odejmowania należy rozważyć możliwość pojawienia się liczb ujemnych. Przy odejmowaniu większej wartości od mniejszej można odwrócić kolejność tych liczb, a następnie zmienić znak wyniku. Na przykład, by odjąć 52 od 45 odejmujemy 45 od 52 i otrzymujemy 7. Dopiero wówczas zmieniamy znak wyniku na -7. Na liczbach binarnych wygląda to tak:

```
 52     = 00110100
-45     = 00110100
  7     = 00000111
```

W normalnej kolejności otrzymalibyśmy:

```
 45      = 00101101
-52      = 00110100
 -7      = 11111001
```

Liczba binarna 11111001 jest zwana uzupełnieniem liczby 7 do dwóch. W ten właśnie sposób procesor 80x86 reprezentuje liczby ujemne. Po odjęciu od 11111001 jedynki, otrzymamy 11111000 czyli -8 itd. Stosowanie arytmetyki liczb ze znakiem oznacza traktowanie wszystkich liczb, które mają ustawiony najstarszy bit, jako ujemne. Tak więc na 8 bitach mogą być reprezentowane liczby z zakresu od -128 (10000000) do 127 (01111111), na 16 od -32,768 do 32,767, a na 32 od -2,147,483,648 do 2,147,483,647.

Zmiana znaku

Instrukcja NEG wyznacza uzupełnienie do dwóch podanego argumentu. Oznacza to konwersję liczby dodatniej na ujemną i odwrotnie. Format tego rozkazu jest następujący:
NEG reg/mem
Reg/mem może być dowolnym rejestrem lub miejscem pamięci. Dla większości liczb ta instrukcja ustawia znacznik przeniesienia. Jedynie gdy argument jest równy zero, to flaga ta nie jest ustawiana. Jeśli natomiast argument jest maksymalną możliwą liczbą ujemną (-128 przy ośmiu bitach i -32,768 dla liczb 16-bitowych), to jego wartość nie jest zmieniana, a ustawiane są tylko znaczniki przeniesienia i nadmiaru.
Instrukcja NEG jest użyteczna przy wyświetlaniu liczb ujemnych. Jeśli liczba jest ujemna, to najpierw może być wyświetlony znak minus, a następnie liczba może być zamieniona rozkazem NEG na liczbę dodatnią i wyświetlona. By funkcja **PrntNumb** obsługiwała również liczby ujemne, należy dodać na jej początku poniższy kod:

```
        or   ax, ax        ; Ustawienie flag
        jns   Positive     ; Jeśli liczba dodatnia
                           ; pomiń kilka linii
        push  ax           ; Zachowaj AX
        mov   dl, '-'      ; Wyświetl znak minus
        mov   ah, 2        ; Numer funkcji systemowej,
                           ; która wyświetla znak
        int   21h
        pop   ax           ; Odtworzenie rejestru AX
        neg   ax           ; Zmiana liczby ujemnej
                           ; na dodatnią
Positive:
```

Rozszerzanie liczb ze znakiem

Przekształcając 8-bitową liczbę bez znaku na daną 16-bitową, należy tylko wyzerować starszy bajt, a do młodszego przesłać tę liczbę. Dla liczb ze znakiem operacja jest bardziej skomplikowana. Starszy bajt danej ujemnej musi mieć bowiem wartość 0ffh. Na przykład, liczba -1, zakodowana na 8 bitach, wygląda następująco 11111111, a na 16 bitach - 1111111111111111. Ogólnie, by rozszerzyć liczbę ze znakiem należy skopiować jej bit znaku do każdego nowo dodanego bitu. Procesory 80x86 posiadają instrukcje, które

pozwalają zamienić jednobajtowe liczby ze znakiem na dwubajtowe, a dwubajtowe na czterobajtowe. Procesor 80386 może ponadto zamienić liczbę czterobajtową na ośmiobajtową. By zamienić jednobajtową liczbę ze znakiem na dwubajtową, należy najpierw umieścić ją w rejestrze AL. Po wykonaniu instrukcji CBW żądana wartość będzie znajdować się w rejestrze AX. Można kontynuować ten proces, używając rozkazu CWD. W parze rejestrów DX i AX otrzymamy wtedy czterobajtową liczbę ze znakiem.
Natomiast w procesorze 80386 można do tego celu zastosować instrukcję CWDE, która umieści wynik w rejestrze EAX. Jest to przydatne, jeśli zamierzamy rozszerzyć naszą liczbę do ośmiu bajtów. Po wykonaniu instrukcji CDQ wynik znajdzie się w rejestrach EDX i EAX.
Przypuśćmy, że po odjęciu dwóch 8-bitowych liczb należy ten rezultat wyświetlić. Można do tego celu użyć instrukcji CBW i funkcji **PrntNumb**:

```
mov   al, 45
sub   al, 50          ; Odjęcie dwóch liczb 8-bitowych
cbw                   ; Rozszerzenie wyniku na daną słowową
call  PrntNumb
```

Mnożenie i dzielenie liczb ze znakiem

Instrukcją MUL można mnożyć tylko liczby bez znaku. W mnożeniu liczb ze znakiem nie ma nic szczególnie trudnego. Należałoby tylko, używając rozkazu NEG, zamienić je na liczby dodatnie i pomnożyć. Prostszym sposobem jest jednak skorzystanie z instrukcji IMUL. Jeśli rezultat mnożenia jest ujemny, to rozkaz ten rozszerza go ze znakiem do rejestru DX przy danych 16-bitowych i do AH - przy 8-bitowych.
Do dzielenia liczb ze znakiem służy natomiast instrukcja IDIV.

Znaczniki

Poza umieszczeniem rezultatu operacji w argumencie docelowym, instrukcje arytmetyczne ustawiają znaczniki zgodnie z wynikiem. Prześledziliśmy już, w jaki sposób ustawiany jest bit przeniesienia przy dodawaniu i odejmowaniu. Istnieje jeszcze kilka innych znaczników, które informują nas o otrzymanym wyniku.
Znacznik zera jest ustawiany, gdy wynik dodawania, odejmowania, inkrementacji lub dekrementacji jest zerem. Jest on często używany w połączeniu z operacją dekrementowania do wykrycia końca pętli.
Znacznik znaku jest ustawiany, gdy wynik dodawania, odejmowania, inkrementacji lub dekrementacji ma najstarszy bit równy jeden. Znak rezultatu jest ważny w wielu operacjach matematycznych.
Znacznik nadmiaru jest ustawiany, gdy wynik operacji na liczbach ze znakiem znalazł się poza zakresem wyznaczonym przez rozmiar argumentów. Na przykład, jeśli rezultat odejmowania 8-bitowych liczb ze znakiem będzie mniejszy od -128, to nie da się przedstawić go na 8 bitach. Co więcej, wynikiem tej operacji będzie liczba dodatnia. Natomiast gdy wynik będzie większy niż 127 przy dodawaniu zostanie przekroczony zakres. Otrzymana suma będzie wtedy liczbą ujemną. Jednakże w obu tych wypadkach zostanie ustawiony bit nadmiaru, informując, że wynik operacji nie mieści się dostępnym zakresie:

```
 21      00010101
115      01110011
137      10001000      ( -118 i ustawiony znacznik nadmiaru )
```

Dwa pozostałe znaczniki są również ustawiane przez operacje dodawania, odejmowania, inkrementacji lub dekrementacji. Są to bity parzystości i przeniesienia pomocniczego. Są one ignorowane w starszych wersjach procesorów i rzadko wykorzystuje się je w programach. Znacznik parzystości jest ustawiony, gdy w wyniku operacji znajduje się nieparzysta liczba bitów równych jeden. Najczęściej jest on używany przy testowaniu poprawności danych. Jeśli jego wartość nie jest tym, czego oczekujemy, to daną uznajemy za błędną. Ten rodzaj sprawdzania jest czasami częścią programów komunikacyjnych.
Znacznik przeniesienia pomocniczego jest ustawiany, gdy wystąpiło przeniesienie na pozycji 3 bitu danej 8-bitowej lub na pozycji 3 lub 7 liczby 16-bitowej. Jest on wykorzystywany przy działaniu na liczbach w kodzie BCD:

```
47       01000111      ( kod BCD )
29       00101001      ( kod BCD )
76       01110000      ( 70 (BCD) i ustawiony znacznik
                         przeniesienia pomocniczego )
```

Arytmetyka na liczbach w kodzie BCD

Istnieje wiele sposobów reprezentowania liczb w komputerze. W poprzednich przykładach liczby były kodowane dwójkowo, każdemu bitowi odpowiada jedna cyfra dwójkowa. Inną popularną metodą zapisu liczb jest zapis dziesiętny kodowany dwójkowo (BCD). Na czterech bitach reprezentowana jest pojedyncza cyfra dziesiętna. Tak więc w jednym bajcie można zmieścić dwie cyfry.
Procesory 80x86 dysponują kilkoma instrukcjami, które działają na liczbach w rozpakowanym kodzie BCD. Oznacza to, że w każdym bajcie znajduje się tylko jedna cyfra. Jest to format dość wygodny, gdyż poprzez dodanie do każdej z nich wartości 30h, cyfry mogą być szybko zamienione na odpowiadające im znaki ASCII.
Wszystkie operacje arytmetyczne mogą być wykonane na liczbach zarówno w spakowanym, jak i rozpakowanym kodzie BCD. W niektórych sytuacjach potrzebne jest tylko wykonanie kilku niezbędnych modyfikacji wyniku. Poniższy przykład pokazuje, jak przebiega dodawanie liczb w kodzie BCD:

```
 26 (BCD)        = 00100110
+18 (BCD)        = 00011000
 3E              = 00111110
```

Ponieważ litera *E* nie jest cyfrą dziesiętną, musi być dokonana korekta wyniku. Instrukcja DAA wykonuje poprawkę dziesiętną po dodawaniu dwójkowym. Dotyczy ona akumulatora AL. W powyższym przykładzie, po jej użyciu, wartość 3E zostanie poprawiona na 44. Instrukcję tę można wykonać tylko po rozkazach ADD i ADC. Spójrzmy, co się dzieje w czasie dodawania liczb 26 i 18:

```
 26 (BCD) = 00100110
+18 (BCD) = 00011000
 44       = 00111110          (3F)

 01000100  Po wykonaniu instrukcji DAA
```

W tym przykładzie, po dodaniu czterech młodszych bitów pojawia się nadmiar, wymagana jest więc odpowiednia poprawka. Pojawienie się nadmiaru powoduje ustawienie znacznika przeniesienia pomocniczego. Posłużenie się instrukcją DAA da więc oczekiwany efekt.
Po operacji odejmowania muszą być wykonane trochę inne poprawki. Zamiast instrukcji DAA należy więc zastosować rozkaz DAS.
Istnieją również instrukcje, które operują na liczbach w rozpakowanym kodzie BCD. Są to rozkazy AAA (poprawka po dodawaniu) i AAS (poprawka po odejmowaniu). Działają one, podobnie jak ich odpowiedniki, w rozpakowanym BCD, z tym, że młodsza cyfra wyniku znajduje się w rejestrze AL, a starsza - w AH.
W wypadku mnożenia i dzielenia całość jest o wiele bardziej skomplikowana. Najlepszym rozwiązaniem jest przekształcenie liczby w kodzie BCD na liczbę binarną przed wykonaniem działania, a po jego zakończeniu wykonanie operacji odwrotnej. Instrukcja AAD (poprawka przed dzieleniem) przekształca liczbę w rozpakowanym kodzie BCD, która znajduje się w rejestrach AH i AL na liczbę binarną w AX. Natomiast rozkaz AAM (poprawka po mnożeniu) liczbę dwójkową w AL przekształca na liczbę w rozpakowanym kodzie BCD w AH i AL. Przed wykonaniem tej instrukcji należy się upewnić, czy wynik jest mniejszy od 100.
Liczby w kodzie BCD są często używane w operacjach wejścia/wyjścia ze względu na łatwość ich przekształcenia na znaki ASCII i odwrotnie. Poniższy przykład odczytuje dwie cyfry z łańcucha **InBuf**, do liczby tej dodaje 17, a wynik umieszcza w łańcuchu **OutBuf**:

```
mov   bx, OFFSET InBuf    ; Adres bufora InBuf
mov   ah, [bx]            ; Pobierz pierwszą cyfrę
inc   bx
mov   al, [bx]            ; Pobierz drugą cyfrę
and   ax, 0f0fh           ; Wyzeruj nieużywane bity
add   ax, 0107h           ; Dodaj liczbę 17 zapisaną
                          ; w rozpakowanym kodzie BCD
aaa                       ; Niezbędne poprawki
or    ax, 3030h           ; Przekształć na znaki ASCII
mov   bx, OFFSET InBuf    ; Adres bufora InBuf
mov   [bx], ah            ; Zapisz pierwszą cyfrę
inc   bx
mov   [bx], al            ; Zapisz drugą cyfrę
```

Prosty kalkulator

Instrukcje arytmetyczne znajdują główne zastosowanie przy wykonywaniu działań arytmetycznych. Wykorzystamy je w programie prostego czterodziałaniowego kalkulatora.

Kalkulator ten działa w Odwrotnej Notacji Polskiej. Po podaniu liczby jest ona odkładana na stos. Natomiast po podaniu komendy, z jego szczytu zostają pobrane dwie wartości, wykonana jest na nich operacja, a wynik zostaje wyświetlony na ekranie i przesłany na stos. Oto komendy rozpoznawane przez nasz kalkulator: "A" - dodawanie, "S" - odejmowanie, "M" - mnożenie, "D" - dzielenie, "P" - pobranie wartości ze szczytu stosu, "C" - całkowite opróżnienie stosu, "Q" - koniec działania.

```
            .MODEL SMALL
            .DATA
InpBuf      DB    10,0,10 dup(?)
OutBuf      DB    5 dup(?),0dh,0ah,"$"
Calcstk     DW    50 dup(?)
Stkptr      DW    0
Minus       DB    0
Prompt      DB    "INPUT> $"
Stkerrmsg   DB    "Stack empty error",0dh,0ah,"$"
Overerrmsg  DB    "Overflow",0dh,0ah,"$"
Cmderrmsg   DB    "Invalid input",0dh,0ah,"$"
            .CODE
Calculate   PROC
            mov   ax, SEG DGROUP     ; Pobierz adres
                                     ; segmentu danych
            mov   ds, ax             ; Prześlij go do DS
            mov   es, ax             ; i do ES
CmdLp:
            mov   dx, OFFSET Prompt  ; Adres łańcucha
                                     ; zachęty
            mov   ah, 9              ; Numer funkcji
                                     ; systemowej, która
                                     ; wyświetla ciąg
                                     ; znaków
            int   21h
            mov   ah, 0ah            ; Numer funkcji
                                     ; systemowej do
                                     ; odczytania danych
                                     ; z klawiatury
            mov   dx, OFFSET InpBuf
            int   21h
            mov   dl, 0Ah
            mov   ah, 2
            int   21h
            mov   dl, 0Dh
            mov   ah, 2
            int   21h
            cmp   InpBuf+2, 'A'      ; Czy to komenda
                                     ; dodawania
```

```
            jnz   NotAdd
            call  PopStack
            jc    CmdLp              ; Gdy wystąpił błąd
                                     ; nie wykonuj
                                     ; działania
            add   Calcstk[bx], ax    ; Zmień wartość
                                     ; na szczycie stosu
            call  PrinTop
            jmp   CmdLp
NotAdd:
            cmp   InpBuf+2, 'S'      ; Czy to komenda
                                     ; odejmowania
            jnz   NotSub
            call  PopStack
            jc    CmdLp
            sub   Calcstk[bx], ax
            call  PrinTop
            jmp   CmdLp
NotSub:
            cmp   InpBuf+2, 'M'      ; Czy to komenda
                                     ; mnożenia
            jnz   NotMul
            call  PopStack
            jc    CmdLp
            imul  Calcstk[bx]
            jnc   MulOK
Overflow:
            mov   dx, OFFSET Overerrmsg
            mov   ah, 9
            int   21h
            jmp   CmdLp
MulOK:
            mov   CalcStk[bx], ax    ; Prześlij rezultat
                                     ; mnożenia na szczyt
                                     ; stosu
            call  PrinTop
            jmp   CmdLp
NotMul:
            cmp   InpBuf+2, 'D'      ; Czy to komenda
                                     ; dzielenia
            jnz   NotDiv
            call  PopStack
            jc    CmdLp
            mov   cx, ax
            mov   ax, Calcstk[bx]
            cmp   ax, 0
```

```
            je    Overflow
            cwd                         ; Rozszerzenie (do
                                        ; rejestrów AX i DX)
                                        ; ze znakiem
                                        ; liczby, która
                                        ; znajduje się na
                                        ; szczycie stosu
            idiv  cx
            mov   Calcstk[bx], ax
            call  PrinTop
            jmp   CmdLp
NotDiv:
            cmp   InpBuf+2, 'P'         ; Czy to komenda
                                        ; pobrania wartości
                                        ; ze stosu
            jnz   NotPop
            call  PopStack
            jmp   CmdLp
NotPop:
            cmp   InpBuf+2, 'C'         ; Czy to komenda
                                        ; pobrania czyszczenia
                                        ; stosu
            jnz   NotClr
            mov   Stkptr, 0             ; Ustaw wskaźnik
                                        ; stosu na 0
            jmp   CmdLp
NotQuit:
            cmp   InpBuf+2, 'Q'
            jne   NotQuit
            mov   ah, 4Ch               ; Numer funkcji
                                        ; systemowej, która
                                        ; kończy program
            int   21h
NotQuit:
            mov   bx, offset Inpbuf+2
            mov   Minus, 0
            sub   ax, ax                ; w AX umieszczona
                                        ; zostanie liczba
            cmp   BYTE PTR Inpbuf[bx], '-' ; Czy to
                                           ; liczba
                                           ; ujemna
            jnz   NotMinus
            mov   Minus, 1
            inc   bx
NotMinus:
            cmp   BYTE PTR Inpbuf[bx], 0dh ; Czy to
```

```
                                        ; koniec
                                        ; linii
            je      Endnum
            cmp     BYTE PTR Inpbuf[bx], '0' ; Czy pobrane
                                        ; znaki to
                                        ; cyfry
            jb      Cmderr
            cmp     BYTE PTR Inpbuf[bx], '9'
            ja      Cmderr
            shl     ax, 1               ; Poprzednia
                                        ; wartość
                                        ; pomnożona
                                        ; przez 2
            mov     cx, ax
            shl     ax, 1
            shl     ax, 1               ; Poprzednia
                                        ; wartość
                                        ; pomnożona
                                        ; przez 8
            add     ax, cx              ; + poprzednia
                                        ; wartość razy 2 =
                                        ; poprzednia
                                        ; wartość razy 10
            mov     cl, BYTE PTR [bx]   ; Pobierz cyfrę
            and     cx, 0fh             ; Wyzeruj nie
                                        ; używane bity
            add     ax, cx              ; Dodaj ją do
                                        ; starej wartości
            inc     bx
            jmp     NotMinus            ; Następne cyfry
Endnum:
            cmp     Minus, 0            ; Sprawdź znak
                                        ; argumentu
            je      Positive
            neg     ax                  ; Zmień znak
Positive:
            mov     bx, Stkptr          ; Pobierz wskaźnik
                                        ; stosu
            mov     Calcstk[bx], ax     ; Zapisz na stosie
                                        ; liczbę
            inc     bx
            inc     bx
            mov     Stkptr, bx          ; Uaktualnij
                                        ; wskaźnik stosu
            jmp     CmdLp
Cmderr:
```

```
            mov   dx, OFFSET Cmderrmsg
            mov   ah, 9
            int   21h
            jmp   CmdLp
Calculate   ENDP
PrinTop     PROC
            mov   ax, Calcstk[bx]    ; Pobierz liczbę ze
                                     ; szczytu stosu
            or    ax, ax             ; Ustaw znaczniki
            jns   pos2
            mov   ah, 5              ; Numer funkcji,
                                     ; która wyświetla
                                     ; znak
            mov   dl, '-'            ; Wyświetl znak '-'
            int   21h
            neg   ax
pos2:
            mov   cl, 5              ; Maksymalna liczba
                                     ; cyfr
            mov   bx, 10             ; System dziesiętny
            mov   di, OFFSET outbuf+4
MakeDigit:
            cmp   ax, 0
            je    Endprn
            cwd
            div   bx
            or    dl, 30h            ; Przekształć na
                                     ; znak ASCII
            mov   byte ptr [di], dl
            dec   di
            loop  MakeDigit
Endprn:
            cmp   cl, 5              ; Czy zapisano choć
                                     ; jedną cyfrę wyniku
            jne   ep1
            mov   byte ptr [di], 0   ; Zapisz wartość 0
            dec   cl
ep1:
            mov   ah, 9              ; Wyświetl łańcuch
            mov   dx, OFFSET OutBuf
            xor   ch, ch
            add   dx, cx             ; Poprawka na nie
                                     ; zapisane cyfry
            int   21h
            ret
PrinTop     ENDP
```

```
PopStack    PROC                        ; Pobierz do AX
                                        ; wartość ze
                                        ; szczytu stosu.
                                        ; Uaktualnij BX, by
                                        ; wskazywał nowy
                                        ; szczyt stosu
            mov   bx, Stkptr            ; Pobierz wskaźnik
                                        ; stosu
            cmp   bx, 0                 ; Czy to koniec stosu
            je    Stkerr
            dec   bx                    ; Cofnij się o
                                        ; jedno słowo
            dec   bx
            mov   Stkptr, bx            ; Uaktualnij
                                        ; wskaźnik stosu
            mov   ax, Calcstk[bx]       ; Pobierz pierwszy
                                        ; argument
            jz    Stkerr                ; Gdy na stosie
                                        ; za mało
                                        ; argumentów, błąd
            dec   bx                    ; BX wskazuje na
                                        ; nowy szczyt stosu
            dec   bx
            clc                         ; Nie zaistniał
                                        ; żaden błąd
            ret
Stkerr:
            mov   dx, OFFSET Stkerrmsg
            mov   ah, 9
            int   21h
            stc                         ; Zaznacz
                                        ; wystąpienie błędu
            ret
PopStack    ENDP
            .STACK   1024
END
```

W pierwszej części programu adres grupy DGROUP jest ładowany do rejestrów DS i ES. Rejestr segmentowy dodatkowy jest również wypełniany po to, aby zapewnić poprawność wykonywania się operacji łańcuchowych. Instrukcje, które znajdują się za etykietą **CmdLp**, tworzą główną pętlę programu. Najpierw zostaje wypisana w niej zachęta, a następnie pobierany jest od użytkownika wiersz danych.

Linia wejściowa przechowywana jest w pamięci pod adresem **Inpbuf**+2. Pierwszy jej znak wyznacza rodzaj komendy. Jest on porównywany ze wszystkimi znakami poleceń. Jeśli jest to jeden ze znaków A, S, M lub D (dodawanie, odejmowanie, mnożenie lub dzielenie), wówczas ze stosu zostaje pobrany jeden argument, a drugim staje się kolejna wartość na

stosie. Operacja P powoduje pobranie jednej danej ze stosu, a polecenie C usuwa ze stosu wszystkie elementy. Komendą Q można program zakończyć.
Jeśli pierwszy znak nie został zinterpretowany jako polecenie, to program sprawdza, czy wiersz danych nie zawiera liczby. Odpowiedzialny za tę operację jest kod, który znajduje się za etykietą NotQuit. Jeśli jako pierwszy zostanie wykryty znak minus, to ustawiamy w stan jeden zmienną, która oznacza, że podana liczba jest ujemna. Algorytm, który przekształca łańcuch znaków na liczbę binarną, jest taki sam jak ten wykorzystany w funkcji **GetNumb**. Następnie uzyskana dana jest przesyłana na stos, a sterowanie wraca na początek pętli.
Do pobrania argumentu ze stosu używana jest funkcja **PopStack**. Stos znajduje się w pamięci pod adresem **CalcStk** i nie jest tym samym stosem, którego używają instrukcje PUSH i POP. Adres ostatnio umieszczonego elementu zawiera zmienna **Stkptr**. Funkcja przepisuje go do rejestru BX. Następnie sprawdza, czy nie jest on zerem, by upewnić się, że na stosie istnieją jeszcze nie pobrane dane. Wskaźnik ten zostaje zmniejszony tak, by wskazywał na element na szczycie stosu. Teraz zostaje pobrany pierwszy argument, a rejestr BX poddany jest znowu zmniejszaniu i wskazuje na argument docelowy. W to właśnie miejsce jest wpisywany rezultat operacji arytmetycznych.
Inną procedurą, używaną przez program kalkulatora, jest **PrinTop**. Jest ona bardzo podobna do procedury **PrntNumb**, z tym że powoduje ona wyświetlenie na ekranie wartości elementu ze szczytu stosu, a nie wartości pochodzącej z rejestru AX.

Instrukcje logiczne

Oprócz operacji arytmetycznych istnieją również operacje logiczne. Używane są one do wyznaczania wartości wyrażeń logicznych, do testowania i ustawiania dowolnych grup bitów, a nawet niektóre działania arytmetyczne mogą być, przy ich użyciu, wykonane efektywniej. Procesory 80x86 mają kilka instrukcji logicznych: and, or, xor oraz rozkazy, które pozwalają na testowanie poszczególnych bitów i ich grup.

Logiczna operacja AND

Rozkaz realizuje logiczne mnożenie odpowiadających sobie bitów dwóch argumentów. Jego format jest następujący:

```
and    mem/reg,reg/mem/stała
```

W rozkazie tym mem/reg jest rejestrem lub miejscem pamięci, a reg/mem/stała rejestrem, miejscem w pamięci lub daną natychmiastową. Oba argumenty muszą mieć takie same rozmiary. Również tylko jeden z nich może wskazywać na miejsce w pamięci. Jeśli jeden z nich jest daną w pamięci, to drugi musi być rejestrem.
Bit wyniku przyjmuje wartość jeden tylko wtedy, gdy odpowiadające sobie bity w obu argumentach są równe jeden. Popatrzmy, jak to wygląda na przykładzie:

```
      10011011
and   10110101
      10010001
```

Instrukcji AND można użyć do zamaskowania niewykorzystywanych bitów. Na przykład, duże i małe litery w kodzie ASCII różnią się tylko wartością piątego bitu. Przeprowadzenie operacji AND na masce, która ma wszystkie bity, oprócz piątego, ustawione na wartość jeden oraz na literze w kodzie ASCII, gwarantuje zamianę małej litery na dużą. Wykonuje to poniższy kod:

```
mov   al, 'a'          ; Zapisz do AL literę 'a' (61h)
and   al, 110111111    ; Wyzeruj piąty bit dając w AL
                       ; literę 'A' ( 41h )
```

Logiczna operacja OR

Format tego rozkazu jest taki sam jak instrukcji AND. Różny jest tylko sposób wyznaczania rezultatu. Bit wyniku przyjmuje wartość jeden wtedy, gdy co najmniej jeden z odpowiadających sobie bitów ma wartość 1. Operacja ta wygląda następująco:

```
      10011011
or    10110101
      10111111
```

Instrukcja OR może służyć do ustawiania poszczególnych bitów danych. Jeśli w przykładzie, dotyczącym zamiany małych liter na duże, zmienimy maskę i wykorzystamy instrukcję OR do ustawienia wartości piątego bitu na jeden, to duże litery zostaną przekształcone na małe:

```
mov   al, 'A'
or    al, 00100000b    ; Ustaw piąty bit na wartość 1
```

Logiczna operacja XOR

Format tego rozkazu jest taki sam jak instrukcji AND czy OR. Różny jest tylko sposób wyznaczania rezultatu. Bit wyniku przyjmuje wartość jeden tylko wtedy, gdy odpowiadające sobie bity w obu argumentach mają przeciwną wartość. Oto sposób działania tej instrukcji:

```
      10011011
xor   10110101
      00101110
```

Interesujące jest to, że wartość bitu, która jest poddawana działaniu instrukcji XOR z bitem równym jeden, zmienia się na przeciwną. Można to wykorzystać do przełączania stanów znaczników i flag.
Instrukcja XOR może być użyta do zakodowania danych. Przewagą tego sposobu jest to, że dwukrotne przeprowadzenie operacji XOR z tą samą maską przywraca pierwotną wartość danej. W poniższym przykładzie każdy bit łańcucha Secret poddawany jest działaniu rozkazu XOR z maską równą 55h.

```
Scrmble     PROC
            mov   bx, OFFSET Secret   ; Pobierz adres łańcucha
SLoop:
```

```
            cmp    [bx], 0            ; Czy to już koniec łań-
cucha
            je     Done
            xor    [bx], 55h          ; Zakoduj bajt
            inc    bx                 ; Następny znak
            jmp    SHORT SLoop
Done:
            ret
Scrmble     ENDP
```

Popatrzmy co się stanie, jeśli zmienna **Secret** będzie zawierała łańcuch "Coded message". Jego pierwszą literą jest C (43h). Po wykonaniu operacji XOR, stanie się ona znakiem control-V (16h). Następnie zostanie zwiększona o jeden zawartość rejestru BX. Gdy bajt wskazywany przez BX przyjmie wartość 0, która oznacza koniec łańcucha, to funkcja zakończy działanie i pozostawi dane już zakodowane.

Logiczna operacja NOT

Rozkaz NOT neguje wszystkie bity w podanym argumencie. Jego format jest następujący:

```
not    reg/mem
```

W rozkazie tym **mem/reg** jest rejestrem lub miejscem pamięci. Działanie instrukcji ilustruje przykład:

```
not   10011011
       01100100
```

Programy graficzne przechowują wyświetlane na ekranie kroje liter w tablicach punktów (ang. bitmaps). Każdy bit tej tablicy reprezentuje pojedynczy piksel ekranu. Normalnie ustawienie bitu na wartość 1 powoduje zaświecenie się odpowiadającego mu punktu na ekranie. Gdy nad literą znajdzie się kursor, to wszystkie bity w tablicy zostają zanegowane, co powoduję inwersję znaku na ekranie. Funkcja w przykładzie, przy użyciu instrukcji NOT, neguje bity ośmiobajtowej tablicy, która przechowuje krój litery:

```
        mov    bx, offset Letter ; Pobierz adres
                                 ; tablicy punktów
        mov    cx, 8             ; Ustal jej rozmiar
Floop:
        not    BYTE PTR [bx]     ; Zaneguj bity w bajcie
                                 ; Musi być podany
                                 ; rozmiar danej, na
                                 ; której przeprowadzana
                                 ; jest operacja
        inc    bx                ; Następny bajt
        loop   Floop
     Oto przykład tablicy punktów dla litery A:
; Przed inwersją
Letter   DB  00010000b
         DB  01101100b
```

```
        DB  11000110b
        DB  11111110b
        DB  11000110b
        DB  11000110b
        DB  00000000b
;Po inwersji
Letter  DB  11101111b
        DB  10010011b
        DB  00111001b
        DB  00000001b
        DB  00111001b
        DB  00111001b
        DB  11111111b
```

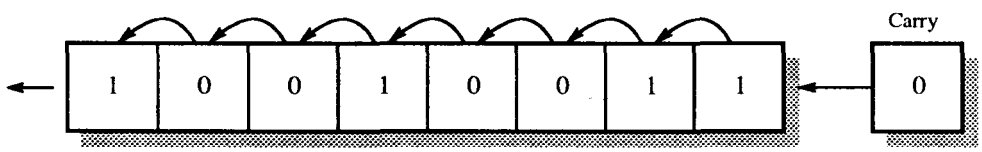
Testowanie bitów

Razem z instrukcjami logicznymi występuje instrukcja TEST - podobnie jak rozkaz CMP był związany z operacjami arytmetycznymi. Jej działanie jest takie samo jak instrukcji AND, z tym że wynik tej operacji służy tylko do ustawienia znaczników.
Rozkaz ten jest używany do testowania stanu pewnych bitów. Na przykład, by sprawdzić czy litera w kodzie ASCII jest duża czy mała, można wykorzystać instrukcję TEST z maską, która ma ustawiony piąty bit. Poniższy przykład ustawia znacznik zera, gdy litera w AL jest duża:

```
test    al, 00100000b          ; Sprawdź stan bitu numer 5
```

Przesunięcia bitów

Jest wiele sposobów kodowania danych w komputerze. Liczby mogą być binarne, dziesiętne kodowane dwójkowo (BCD) lub nawet łańcuchami znaków. Znaki mogą być w kodzie ASCII, BAUDOT itp. Każdy format wymaga odrębnego interpretowania znaczenia bitów w bajcie. Różnorodność ta sprawia, że komputer musi często dokonywać konwersji danych. W operacjach tych są przydatne instrukcje przesuwania. Poza tym mogą być one używane do mnożenia przez liczby, która są potęgami dwójki, do zliczania bitów, do wysuwania bitów traktowanych jako flagi.
Rozkazy przesunięć po prostu przesuwają zawartość bajtu, czy słowa, w lewo lub w prawo. Ilustruje to rysunek 5-2.



Rysunek 5-2. ***Przesuwanie bitów w bajcie.***

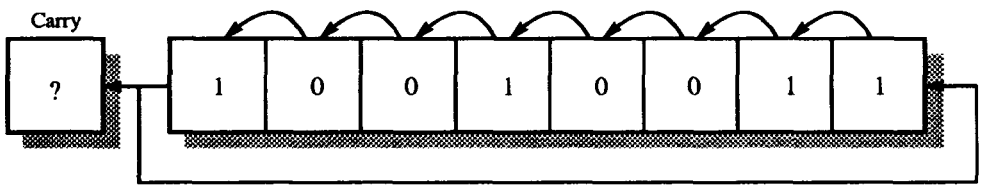
Jeden bit zostaje z bajtu wysunięty, pozostałe odpowiednio zostają w nim przemieszczone, a na drugim końcu pojawia się nowy bit. Różnice między instrukcjami przesuwania sprowadzają się do tego, co się dalej dzieje z bitem wysuniętym, i jakimi wartościami zostaje bajt uzupełniony.

Rotacje

Z rotacją danej mamy do czynienia wówczas, gdy bity, wysuwane z bajtu są wprowadzane do niego drugim końcem. Istnieją cztery tego rodzaju instrukcje: ROR, ROL, RCR i RCL. Rozkazy ROR oraz ROL dokonują rotacji danej, jednocześnie wprowadzając wysuwane bity do bitu przeniesienia. Instrukcje przesuwania mogą być użyte w trzech formatach:

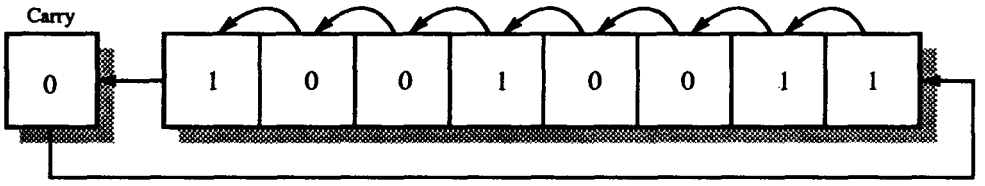
```
rol  mem/reg,1      ; Rotacja o jedną pozycję
rol  mem/reg, cl    ; Rotacja o ilość pozycji
                    ; określoną w rejestrze CL
rol  mem/reg, stała ; Rotacja o ilość pozycji
                    ; określoną przez daną
                    ; natychmiastową ( tylko
                    ; w procesorach 80286 i 80386 )
```

Mem/reg jest dowolnym 8- lub 16-bitowym rejestrem lub miejscem w pamięci. Rysunek 5-3 pokazuje efekt rotacji bajtu w lewo, o jedną pozycję.



Rysunek 5-3. ***Rotacja w lewo.***

W tym przykładzie, po wykonaniu instrukcji ROL, znacznik przeniesienia przyjmie wartość jeden. Oznacza to, że taką wartość miał wysunięty bit. Wynikiem tej operacji będzie liczba 00100111b. Podobne działanie ma rozkaz ROR, z tym że przesuwanie odbywa się w prawo i daje rezultat 11001001b oraz ustawiony znacznik przeniesienia.
Natomiast w instrukcjach RCL i RCR rotacja odbywa się poprzez znacznik przeniesienia. Różnica jest może subtelna, ale bardzo ważna. Ilustruje ją rysunek 5-4.
Teraz wynik jest równy 00100110b i znacznik przeniesienia zostaje ustawiony. Zamiast bowiem przesłania najstarszego bitu bezpośrednio do najmłodszego, najpierw jest do niego wpisywany bit przeniesienia, a dopiero potem najstarszy bit umieszczany jest w znaczniku CF. Proces ten można sobie wyobrażać jako zwykłą rotację danej 17- zamiast 16-bitowej lub 9-bitowej zamiast 8-bitowej. Rotacja poprzez znacznik przeniesienia może być użyteczna przy przesuwaniu kilku bitów z jednego bajtu do drugiego.
Następny przykład pokazuje, jak przesunąć w prawo, o jedną pozycję, 32-bitową daną z rejestrów AX i DX:



Rysunek 5-4. ***Rotacja w lewo poprzez wskaźnik przeniesienia.***

```
shr   dx, 1          ; Przesuń najpierw starsze słowo
rcr   ax, 1          ; Wstaw do młodszego bit, który
                     ; pochodzi ze starszego słowa
```

Przesuwanie bitów

Instrukcje przesuwania nie powodują powrotu wysuniętych bitów do danej macierzystej. Jest ich również cztery: SAL, SAR, SHL i SHR. Rozkazy SAL i SHL przesuwają daną w lewo. Najbardziej znaczący bit zostaje wysunięty do znacznika przeniesienia, a najmłodszy zostaje uzupełniony zerem. Właściwie instrukcje te są identyczne, a asembler generuje dla nich ten sam kod wynikowy.

Rozkaz SHR jest podobny do dwu poprzednich, z tym tylko że przesuwanie odbywa się w prawo. Natomiast instrukcja SAR działa trochę inaczej. Zamiast bowiem uzupełnienia zerem najstarszego bitu (tak, jak to robi rozkaz SHR), nie zmienia ona jego wartości. Wynika to stąd, że przesuwanie w prawo jest równoważne operacji dzielenia przez dwa. Najstarszy bit jest bitem znaku i zachowanie jego wartości zachowuje poprawność tego działania również na liczbach ze znakiem.

ZASTOSOWANIE INSTRUKCJI PRZESUWANIA

Instrukcje przesuwania stosowane są często w obliczeniach matematycznych. Pomyśl np. O tym, jak wykonujesz mnożenie przez dziesięć. Po prostu przesuwasz wszystkie cyfry o jedną pozycję w lewo, a na końcu dodajesz 0. Właśnie taką operację wykonują przecież rozkazy przesuwania, z tym tylko że stosowana ona jest przy podstawie 2. Dlatego też przesunięcie w lewo o jeden bit jest równoważne mnożeniu przez dwa, o dwa bity przez cztery itd. Na tej samej zasadzie operacja dzielenia może być wykonana jako przesuwanie w prawo.

Mnożenie przez przesuwanie znajduje zastosowanie przy przeglądaniu tablic. Gdy program wywołuje funkcję systemową, to podaje jej numer w rejestrze AH. MS-DOS używa tego numeru do znalezienia adresu tej funkcji w tablicy adresów. Ponieważ każdy adres funkcji zajmuje dwa bajty, to jej numer musi zostać najpierw pomnożony przez dwa. Poniższy program używa numeru funkcji podanego w rejestrze AH do znalezienia jej adresu zapisanego w tablicy FuncTab:

```
xor   bx, bx         ; Wyzeruj rejestr BX
mov   bl, ah         ; Zapisz numer funkcji do BX
shl   bx, 1          ; Pomnóż przez dwa ( rozmiar
```

```
                                ; elementu tablicy )
mov   bx, FuncTab[bx]           ; Pobierz adres funkcji
                                ; z tablicy
call  bx                        ; Wywołaj ją
```

MNOŻENIE PRZEZ DOWOLNĄ LICZBĘ

Mnożenie przez dowolną stałą wartość da się wykonać przy użyciu wielu przesunięć i operacji dodawania. Wynika to stąd, że każda liczba może być zapisana jako suma kilku potęg liczby dwa. Na przykład, liczbę 12 możemy otrzymać, dodając do siebie osiem (2^3) i cztery (2^2). Tak więc, by pomnożyć przez 12, można najpierw dokonać mnożenia przez 8 i 4, a później rezultaty tych działań zsumować. Sposób ten został wykorzystany w poniższym przykładzie do pomnożenia przez 12 rejestru AX:

```
shl   ax, 1                     ; ax = liczba * 2
shl   ax, 1                     ; ax = liczba * 4
mov   bx, ax                    ; bx = liczba * 4
shl   ax, 1                     ; ax = liczba * 8
add   ax, bx                    ; Dodanie obu rezultatów daje
                                ; wynik liczba * 12
```

Jeśli rejestr AX, przed wykonaniem tego kodu, zawiera wartość 30, to pierwsze przesunięcie zmienia ją na 60, a następne na 120. Instrukcją MOV wynik ten (AX * 4) zostaje przesłany do BX. Następne przesunięcie znowu podwaja zawartość rejestru AX, która wynosi teraz 240. Ostatni rozkaz dodaje do siebie dwie wartości, co daje rezultat 360.

Dla procesorów 8088 i 8086 wykonanie instrukcji MUL zajmuje od 82 do 89 taktów. Powyższy kod natomiast potrzebuje tylko 12 taktów na przeprowadzenie operacji. Jego możliwości są jednak ograniczone. Wynik operacji będzie poprawny tylko dla liczb mniejszych od 65536/12.

Normalizacja wskaźników

Ze względu na segmentową strukturę pamięci w procesorach 80x86, adres danej znajduje się w dwóch rejestrach. Ponieważ segmenty mogą się nakładać, to istnieje wiele różnych sposobów zaadresowania pojedynczego miejsca w pamięci. Na przykład, adres **000A:1234** jest równoważny adresowi **012D:0004**. Niejednoznaczność ta może powodować trudności przy porównywaniu wskaźników. Jednym ze sposobów obejścia tego problemu jest zastosowanie wskaźników znormalizowanych. Wskaźnik znormalizowany ma największy możliwy numer segmentu i najmniejszą możliwą wartość przemieszczenia. Najprościej można przeprowadzić normalizację wskazania, używając instrukcji przesunięć. Poniższy przykład dokonuje tej operacji na wskaźniku **ES:BX**.

```
; Na wejściu podaje się numer segmentu w rejestrze ES
; i przemieszczenie w BX
    mov   ax, bx            ; Prześlij przemieszczenie
    shr   bx, 1             ; Dzielenie przez 2
    shr   bx, 1             ; Dzielenie przez 4
    shr   bx, 1             ; Dzielenie przez 8
    shr   bx, 1             ; Dzielenie przez 16
    mov   cx, es            ; Pobierz numer segmentu
```

```
add     cx, bx          ; Dodaj przemieszczenie
                        ; podzielone przez 16
mov     es, cx
and     ax, 0fh         ; Wyzeruj część adresu, która
                        ; została przesłana do
                        ; rejestru segmentowego
mov     bx, ax
```

Tak więc, gdy wskaźnik zawarty w rejestrach ES:BX ma wartość 000A:1234, to podzielenie przesunięcia przez 16 da wartość 123h. Jest ona następnie dodawana do numeru segmentu, co ustala jego wartość na 012Dh. W ostatnim kroku maskowane są wszystkie bity oryginalnego przemieszczenia, oprócz czterech najmłodszych. Po znormalizowaniu rejestry ES:BX zawierają wskaźnik 012D:0004.

PRZESUWANIE WIELOKROTNE

W poprzednich przykładach dane były przesuwane tylko o jedną pozycję. Procesory 80x86 mogą jednakże przesuwać dane również o kilka pozycji. Dla układów 80286 i 80386 liczbę iteracji można podać bezpośrednio w rozkazie. Na przykład, by zamienić miejscami półbajty w rejestrze AL, można zastosować instrukcję:

```
rol     al, 4
```

Natomiast w procesorach 8088 i 8086 do wielokrotnego przesuwania musi być wykorzystany rejestr CL. By otrzymać taki sam wynik, jak w powyższym przykładzie, należy użyć kodu:

```
mov     cl, 4
rol     al, cl
```

Przed wybraniem jednego z tych formatów, należy poświęcić chwilę uwagi efektywności każdego z nich. Identyczny rezultat otrzymamy przecież jeśli umieścimy kolejno kilka rozkazów, które przesuwają o jedną pozycję. Powyższy przykład mógłby być zakodowany następująco:

```
rol     al, 1
rol     al, 1
rol     al, 1
rol     al, 1
```

Zajmie on więcej pamięci, ale w wielu wypadkach jego działanie będzie szybsze. Procesory 8088 i 8086 mają bowiem wolniejszy mikrokod dla operacji wielokrotnego przesuwania. Jest wiele czynników, które determinują kiedy oddzielne przesuwanie będzie szybsze. Ogólnie, jeśli przesuwamy o mniej niż sześć bitów, jest ono efektywniejsze.

PRZESUWANIE W PODWÓJNEJ PRECYZJI NA PROCESORACH 80386

Procesor 80386 ma dwie dodatkowe instrukcje przesuwające: SHLD i SHRD. Ich format jest następujący:

```
SHLD    reg/mem,reg/mem,licznik
```

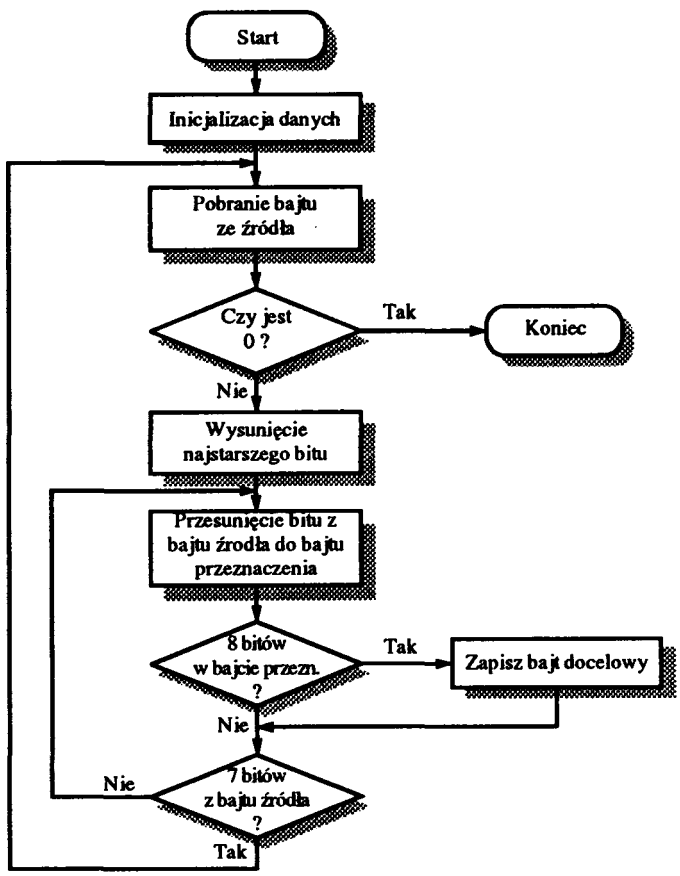
W rozkazie tym mem/reg jest rejestrem lub miejscem pamięci, a licznik może być 8-bitową stałą (0-255) lub rejestrem CL. Instrukcje te powodują przesunięcie bitów z drugiego

argumentu i wstawienie ich do pierwszego. Licznik określa liczbę powtórzeń tej operacji. Argumenty mogą być zarówno 16- jak i 32-bitowe. Oba natomiast muszą mieć takie same rozmiary i tylko jeden z nich może wskazywać na miejsce w pamięci.

Wykorzystanie instrukcji przesuwania do kompresji bitów

Instrukcje przesuwające są wygodne, gdy zmieniamy format danych. Taką pożyteczną zmianą może być np. usunięcie ósmego, nie wykorzystywanego w kodzie ASCII, bitu z każdego bajtu łańcucha znaków. Taki skompresowany łańcuch zajmie o 12.5% mniej miejsca w pamięci niż łańcuch zwykły.

W procesie kompresji łańcucha program najpierw pobiera z niego jeden bajt, a następnie wysuwa z niego poszczególne bity. Bit najstarszy zostaje odrzucony, a pozostałe są wsuwane do bajtu docelowego. Następnie zostaje pobrany kolejny znak. Gdy natomiast zapełni się bajt docelowy, to wpisywany jest on do już skompresowanego łańcucha. Program kończy się po odczytaniu z łańcucha źródłowego bajtu zerowego. Rysunek 5-5 pokazuje algorytm działania programu.



Rysunek 5-5. ***Algorytm kompresji bitów.***

```
            .MODEL SMALL
            .DATA
Source      DB    80 dup(?)          ; Łańcuch źródłowy
Dest        DB    70 dup(?)          ; Łańcuch
                                     ; skompresowany
            .CODE
Squeeze     PROC
            mov   ax, SEG DGROUP
            mov   ds, ax
            mov   es, ax
            cld                      ; Zwiększanie adresów
            mov   si, OFFSET Source  ; Adres łańcucha
                                     ; źródłowego
            mov   di, OFFSET Dest    ; Adres łańcucha
                                     ; docelowego
            mov   dh, 8              ; Liczba bitów
                                     ; do skopiowania
NxtChar:
            mov   dl, 7              ; Liczba bitów
                                     ; w bajcie docelowym
            lodsb                    ; Pobierz znak
            or    al, al             ; Ustaw znaczniki
            jz    Done               ; Koniec, gdy zero
            shl   al, 1              ; Pozbycie się
                                     ; najstarszego bitu
NxtBit:
            shl   al, 1              ; Wysuń bit do
                                     ; znacznika przeniesienia
            rcl   ah, 1              ; Umieść ten bit
                                     ; w bajcie docelowym
            dec   dh                 ; Zliczaj zapisane bity
            jnz   ChkSrc             ; Skocz, gdy potrzebne
                                     ; jeszcze są bity
            xchg  ah, al             ; w przeciwnym razie
                                     ; zapisz AH w łańcuchu
                                     ; docelowym
            stosb
            xchg  ah, al
            mov   dh, 8              ; Ustaw początkową
                                     ; wartość licznika
ChkSrc:
            dec   dl                 ; Zliczaj bity w
                                     ; bajcie źródłowym
            jz    NxtChar            ; Następny znak
            jmp   short NxtBit       ; Kontynuuj
                                     ; przesuwanie
```

```
Done:
        shl     ah, 1               ; Wsuń zerowe bity
                                    ; w ostatni bajt
        dec     dh
        jnz     Done
        xchg    ah, al
        stosb                       ; Zapisz ostatni bajt
        xchg    ah, al
        stosb                       ; Zapisz 0
        mov     ah, 4Ch
        int     21h
Squeeze ENDP
        .STACK    1024
        END         .
```

Podsumowanie

Procesory 80x86 dysponują wieloma rozkazami do operowania na danych: instrukcje arytmetyczne, logiczne oraz instrukcje przesuwające bity. Zastosowanie instrukcji arytmetycznych jest dobrze znane: obliczanie wartości wyrażeń matematycznych.

Trochę mniej znane jest zastosowanie rozkazów logicznych. Najbardziej znanym jest maskowanie poszczególnych części bajtów i słów oraz manipulowanie na znacznikach. W tym ostatnim wypadku należy pamiętać , że aby znacznik wyzerować należy poddać go operacji AND z bitem równym 0, by go ustawić operacji OR - z bitem równym 1, a zmienić jego stan na przeciwny można, używając instrukcji XOR z wartością bitu maski równą 1.

Użycie instrukcji logicznych nie zmienia położenia bitów w bajcie. Jeśli potrzebna informacja znajduje się np. W jego starszych czterech bitach, to zamaskowanie ich nadal pozostawia je w tym samym miejscu. Natomiast, używając rozkazów przesuwania, można je już przemieścić na właściwą pozycję.

Instrukcje przesuwania można również zastosować do mnożenia i dzielenia liczb przez stałe. Operacja ta jest najbardziej efektywna, jeśli stałe te są potęgami dwójki: 2, 4, 8, 16 itd. Używając rozkazów przesuwania zamiast instrukcji MUL czy DIV można poprawić efektywność twoich programów.

ASCII Adjust after Addition

88/86	286	386
●	●	●

AAA

ZASTOSOWANIE

Instrukcja AAA używana jest do korekty wyniku po dodaniu dwóch liczb w rozpakowanym kodzie BCD.

SKŁADNIA

AAA

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Stan nieokreślony
Zero	Stan nieokreślony
Parzystość	Stan nieokreślony
Przeniesienie	Ustawiany, gdy cyfra w AL była większa niż 9
Przeniesienie pomocnicze	Ustawiany, gdy znacznik przeniesienia jest ustawiony

Znaczniki przeniesień są ustawione, gdy cyfra w AL jest większa niż 9. W przeciwnym wypadku zostają one wyzerowane.
Stan znaczników nadmiaru, znaku, zera i parzystości jest nieokreślony.

KOMENTARZ

Rozpakowany kod BCD jest użyteczny ze względu na łatwość jego zamiany na kod ASCII. Po wykonaniu operacji dodawania może się zdarzyć, że rezultat będzie zawierał cyfrę nie dziesiętną. Instrukcja ta dokonuje wtedy korekty wyniku.

PATRZ

AAS	*Poprawka po odejmowaniu*
DAA, DAS	*Poprawki w spakowanym kodzie BCD*
AAD, AAM	*Konwersja między systemem binarnym i BCD*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	00110111	aaa	8	3	4

PRZYKŁAD

Dodanie zawartości rejestru BX do zmiennej BcdNum. Wszystkie liczby są w rozpakowanym kodzie BCD.

```
          .DATA
BcdNum    DW    45h
          .CODE
          mov   bx, 27h
```

AAD

```
mov   ax, BcdNum    ; Pobierz wartość zmiennej
                    ; do AX
add   ax, bx        ; AX = 6Ch
aaa                 ; Po korekcie AX = 72h
mov   BcdNum, ax    ; Zapisz rezultat
```

ASCII Adjust for Division

88/86	286	386
●	●	●

AAD

ZASTOSOWANIE

Instrukcja AAD przeprowadza konwersję liczby w rozpakowanym kodzie BCD, która to liczba znajduje się w rejestrach AH i AL, na liczbę binarną w AX.

SKŁADNIA

AAD

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Ustawiany zgodnie ze znakiem rezultatu
Zero	Ustawiany, jeśli wynik jest równy zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Stan nieokreślony
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Rozpakowany kod BCD jest bardzo nieefektywnym sposobem przechowywania danych. Dwubajtowa liczba w tym kodzie ma zakres od 0 do 99, podczas gdy taka sama liczba binarna - od 0 do 65535. Z tego właśnie względu oraz biorąc pod uwagę to, że działania arytmetyczne w tym kodzie wymagają dodatkowych operacji, w większości programów dokonuje się konwersji danych do systemu dwójkowego. Wtedy może się okazać przydatna instrukcja AAD.

PATRZ

AAM	*Konwersja liczby binarnej na liczbę w rozpakowanym kodzie BCD*
AAA, AAS	*Poprawki w rozpakowanym kodzie BCD*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	11010101 00001010	aad	60	14	19

PRZYKŁAD

Procedura przekształca cyfry w kodzie ASCII na liczbę binarną i umieszcza ją w zmiennej **BinNum**.

```
          .DATA
BcdNum    DB    "43"          ; Przekształcana liczba
BinNum    DW    ?             ; Miejsce na wynik
          .CODE
           mov    ax, BcdNum   ; Pobierz liczbę
           and    ax, 0F0Fh    ; Przekształć na rozpakowany
                               ; kod BCD (0403h)
           aad                  ; Przekształć na kod dwójkowy
           mov    BinNum, ax   ; Zapisz rezultat
```

ASCII Adjust for Multiplication — 88/86 ● 286 ● 386 ●

AAM

ZASTOSOWANIE

Konwersja liczby binarnej na daną w rozpakowanym kodzie BCD.

SKŁADNIA

AAM

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynik jest równy zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Stan nieokreślony
Przeniesienie pomocnicze	Stan nieokreslony

KOMENTARZ

Za pomocą tej instrukcji można w najprostszy sposób przygotować do wyświetlenia liczby mniejsze niż 100.

PATRZ

AAD	*Konwersja liczby w rozpakowanym kodzie BCD na liczbę binarną*
AAA, AAS	*Poprawki w rozpakowanym kodzie BCD*

AAS

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	11000101 00001010	aam	83	16	17

PRZYKŁAD

Wyświetlenie liczby, która znajduje się w zmiennej BinNum. Przyjmujemy, że jest ona mniejsza niż 100.

```
          .DATA
BinNum    DW    37
          .CODE
          mov   ax, BinNum    ; Pobierz liczbę
          aam                 ; Dokonaj konwersji
                              ; ( AX = 0307h )
          mov   bx, ax        ; Zachowaj tą wartość
          mov   dl, bh        ; Pobierz starszą
                              ; cyfrę (03h)
          add   dl, '0'       ; Przekształć ją na
                              ; znak ASCII
          mov   ah, 2         ; Numer funkcji, która
                              ; wyświetla pojedynczy znak
          int   21h
          mov   dl, bl        ; Pobierz młodszą cyfrę (07h)
          add   dl, '0'
          int   21h           ; AH = 2
```

ASCII Adjust for Substraction

88/86	286	386
●	●	●

AAS

ZASTOSOWANIE

Instrukcja AAS używana jest do korekty wyniku, po odjęciu dwóch liczb w rozpakowanym kodzie BCD.

SKŁADNIA

AAS

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Stan nieokreślony
Zero	Stan nieokreślony

Parzystość	Stan nieokreślony
Przeniesienie	Ustawiany, gdy cyfra w AL była większa niż 9
Przeniesienie pomocnicze	Ustawiany, gdy znacznik przeniesienia jest ustawiony

KOMENTARZ

Rozpakowany kod BCD jest użyteczny ze względu na łatwość jego zamiany na kod ASCII. Po wykonaniu operacji odejmowania może się zdarzyć, że rezultat będzie zawierał cyfrę nie dziesiętną. Instrukcja ta dokonuje wtedy korekty wyniku.

PATRZ

AAA	*Poprawka po dodawaniu*
DAA, DAS	*Poprawki w spakowanym kodzie BCD*
AAD, AAM	*Konwersja między systemem binarnym i BCD*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	00111111	aas	8	3	4

PRZYKŁAD

W poniższym programie odejmujemy zawartość rejestru BX od zmiennej BcdNum. Wszystkie liczby są w rozpakowanym kodzie BCD.

```
mov   ax, BcdNum      ; Pobierz wartość zmiennej do AX
sub   ax, bx
aas                   ; Korekta wyniku
mov   BcdNum, ax      ; Zapisz rezultat
```

Jeśli w rejestrze BX znajduje się wartość 0502h, a w zmiennej BcdNum wartość 0107h, to odejmowanie da wynik 03FBh. Instrukcja AAS dokonuje korekty wyniku, a rezultatem działania jest 0305h - liczba poprawna w rozpakowanym kodzie BCD.

Add with Carry

88/86	286	386
●	●	●

ADC

ZASTOSOWANIE

Dodanie dwóch liczb i znacznika przeniesienia. Instrukcja jest najczęściej używana przy dodawaniu wielobajtowych liczb. Pozwala ona bowiem na uwzględnienie przeniesień, powstałych na młodszych bajtach.

ADC

SKŁADNIA

`ADC reg, reg`	*Dodanie zawartości dwóch rejestrów*
`ADC reg, mem`	*Dodanie do zawartości rejestru danej z pamięci*
`ADC mem, reg`	*Dodanie do danej w pamięci zawartości rejestru*
`ADC reg, stała`	*Dodanie danej natychmiastowej do rejestru*
`ADC mem, stała`	*Dodanie danej natychmiastowej do danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli powstał nadmiar przy dodawaniu liczb ze znakiem
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy najmłodszy bit argumentu był ustawiany
Przeniesienie pomocnicze	Ustawiany, gdy pojawiło się przeniesienie z pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Rozkaz ten jest używany po instrukcjach, które mogą zmieniać wartość znacznika CF. Służy on najczęściej do dodania dwóch bardziej znaczących słów, po dodaniu słów mniej znaczących.

PATRZ

ADD	*Dodawanie bez przeniesienia*
STC, CLC	*Ustawienie i wyzerowanie znacznika przeniesienia*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0001000w 11,reg,r/m	adc ax,bx	3	2	2
mem/reg	0001000w mod,reg,r/m przesunięcie	adc num,ax	24/16+EA	7	7
reg/mem	0001001w mod,reg,r/m przesunięcie	adc ax,num	13/9+EA	7	6
reg/stała	100000sw mod,010,r/m dana	adc bx,4	4	3	2
mem/stała	100000sw mod,010,r/m przemieszczenie dana	adc num,5	23/17+EA	7	7

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
acc, stała	0001010w dana	adc al,4	4	3	2

PRZYKŁAD

Dodanie 64-bitowej liczby, która znajduje się w zmiennej BigNum, do liczby 32-bitowej w SmallNum.

```
          .DATA
SmallNum  DD   12345678h
BigNum    DQ   4122323112312344h
          .CODE
          mov   ax, WORD PTR SmallNum   ; Pobierz młodsze
                                        ; słowo zmiennej
                                        ; SmallNum (5678h)
          add   WORD PTR BigNum, ax     ; Dodaj do
                                        ; BigNum ( 79BCh )
          mov   ax, WORD PTR SmallNum+2; Pobierz następne
                                        ; słowo ( 1234h )
          adc   WORD PTR BigNum+2, ax   ; Dodaj do BigNum,
                                        ; uwzględniając
                                        ; przeniesienie
                                        ; (2465h)
          adc   WORD PTR BigNum+4, 0    ; Dodaj ostatnie
                                        ; przeniesienie
          adc   WORD PTR BigNum+6, 0    ; BigNum =
                                        ; 41223231246579BCh
```

Add Data

88/86	286	386
●	●	●

ADD

ZASTOSOWANIE

Dodanie do siebie dwóch liczb.

SKŁADNIA

`ADD reg, reg`	*Dodanie zawartości dwóch rejestrów*
`ADD reg, mem`	*Dodanie do zawartości rejestru danej z pamięci*
`ADD mem, reg`	*Dodanie do danej w pamięci zawartości rejestru*
`ADD reg, stała`	*Dodanie danej natychmiastowej do rejestru*
`ADD mem, stała`	*Dodanie danej natychmiastowej do danej w pamięci*

ADD

ZNACZNIKI

Nadmiar	Ustawiany, jeśli powstał nadmiar przy dodawaniu liczb ze znakiem
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy najmłodszy bit argumentu był ustawiany
Przeniesienie pomocnicze	Ustawiany, gdy pojawiło się przeniesienie z pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Instrukcja ta przeprowadza operację dodawania dwóch liczb. Jeśli rezultat nie mieści się w podanym argumencie, to znacznik przeniesienia jest ustawiany. W opisie rozkazu ADC przedstawiony jest jeden ze sposobów wykorzystania tego faktu.

PATRZ

ADC *Dodawanie z przeniesieniem*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0000000w 11,reg,r/m	add ax,bx	3	2	2
mem/reg	0000000w mod,reg,r/m przemieszczenie	add num,ax	24/16+EA	7	7
reg/mem	0000001w mod,reg,r/m przemieszczenie	add ax,num	13/9+EA	7	6
reg/stała	100000sw mod,000,r/m dana	add bx,4	4	3	2
mem/stała	100000sw mod,000,r/m przemieszczenie dana	add num,5	23/17+EA	7	7
acc, stała	0001010w dana	add al,4	4	3	2

PRZYKŁAD

Patrz przykład do instrukcji ADC.

Logical AND

88/86	286	386
●	●	●

AND

ZASTOSOWANIE

Mnożenie logiczne argumentów

SKŁADNIA

`AND reg, reg`	*Mnożenie logiczne zawartości dwóch rejestrów*
`AND reg, mem`	*Mnożenie logiczne zawartości rejestru i danej z pamięci*
`AND mem, reg`	*Mnożenie logiczne danej w pamięci i zawartości rejestru*
`AND reg, stała`	*Mnożenie logiczne zawartości rejestru i danej natychmiastowej*
`AND mem, stała`	*Mnożenie logiczne zawartości pamięci i danej natychmiastowej*

ZNACZNIKI

Nadmiar	Zerowany
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Zerowany
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Instrukcja ta jest często stosowana do wyzerowania niepotrzebnych bitów. Przeprowadzenie operacji AND z maską nie zmieni tylko wartości tych bitów danej, które w masce mają wartość jeden. Pozostałe zostaną wyzerowane.

PATRZ

OR, XOR, NOT	*Pozostałe instrukcje logiczne*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0010000w 11,reg,r/m	and ax,bx	3	2	2
mem/reg	0010000w mod,reg,r/m przemieszczenie	and num,ax	24/16+EA	7	7
reg/mem	0010001w mod,reg,r/m przemieszczenie	and ax,num	13/9+EA	7	6
reg/stała	100000sw mod,100,r/m dana	and bx,4	4	3	2

BSF, BSR

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
mem/stała	100000sw mod,100,r/m przemieszczenie dana	and num,5	23/17+EA	7	7
acc, stała	0010010w dana	and al,4	4	3	2

PRZYKŁAD

Funkcja pobiera stan klawiatury z obszaru danych BIOS-u. Maskuje bity, które odpowiadają klawiszom *Insert*, *Caps Lock*, *Num Lock* oraz *Scroll Lock*. Jeśli klawisz *Caps Lock* był aktywny, wówczas ustawia ona stan lewego klawisza Shift na wartość 1.

```
GetShift    PROC
            mov     ah, 2       ; Odczyt stanu klawiszy
                                ; sterujących
            int     16h         ; Wywołanie procedur BIOS-u,
                                ; które obsługują klawiaturę
            test    al, 40h     ; Sprawdź stan klawisza
                                ; Caps Lock
            jz      GS1         ; Skocz, jeśli nie aktywny
            or      al, 2       ; Ustaw bit, który odpowiada
                                ; lewemu klawiszowi Shift
GS1:
            and     al, 0Fh     ; Wyzeruj starszy półbajt
            ret
GetShift    ENDP
```

Bit Scan Forward, Bit Scan Reverse — 88/86 286 386 ●

BSF, BSR

ZASTOSOWANIE

Przeglądanie bitów argumentu źródłowego w celu znalezienia pozycji pierwszego ustawionego bitu. Pozycja ta jest następnie wpisywana do argumentu docelowego.

SKŁADNIA

BSF	*Przeszukiwanie od najmniej znaczącego bitu*
BSR	*Przeszukiwanie od najbardziej znaczącego bitu*

ZNACZNIKI

Znacznik zera jest ustawiany, gdy zostanie znaleziony ustawiony bit. W przeciwnym razie jest on zerowany.

KOMENTARZ

Liczba zwracana przez obie te instrukcje jest numerem bitu w słowie. Najmniej znaczący bit ma numer 0.

PATRZ

BT, BTC, BTR, BTS *Operacje testowania bitów*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
BSF					
reg, reg	00001111 10111100 mod,reg,r/m	bsf ecx,eax	-	-	10+3n
reg, mem	00001111 10111100 mod,reg,r/m przemieszczenie	bsf ecx, mask	-	-	10+3n
BSR					
reg, reg	00001111 10111101 mod,reg,r/m	bsr cx,ax	-	-	10+3n
reg, mem	00001111 10111101 mod,reg,r/m przemieszczenie	bsr ecx, mask	-	-	10+3n

PRZYKŁAD

Reprezentacja liczby rzeczywistej w komputerze składa się z dwóch części. Pierwsza z nich jest mantysą, a drugą stanowi wykładnik. Mówimy, że liczba rzeczywista jest znormalizowana, gdy pierwszy bit mantysy ma wartość 1. Skoro tak to nie musi być on zapamiętywany. Poniższy przykład korzysta z instrukcji BSR, by ustalić, o ile pozycji powinna zostać przesunięta mantysa.

```
            .MODEL SMALL
            .386                    ; Używane są instrukcje
                                    ; procesora 80386
            .DATA
FloatNumb   DW   100                ; Mantysa
```

```
FloatExp    DB   0                  ; Wykładnik
            .CODE
Normalize   PROC
            bsr   ax, FloatNumb     ; Ustal pozycję
                                    ; najstarszego bitu
                                    ; równego 1
            sub   FloatExp, al      ; Zmodyfikuj wartość
                                    ; wykładnika
            mov   cl, 16            ; Liczba bitów w słowie
            sub   cl, al            ; Oblicz liczbę pozycji,
                                    ; o którą należy
                                    ; przesunąć mantysę
            shl   FloatNumb, cl     ; Przesunięcie mantysy
            ret
Normalize   ENDP
```

Bit Test, Bit Test and Complement
Bit Test and Reset, Bit Test and Set

88/86	286	386
		●

BT, BTC, BTR, BTS

ZASTOSOWANIE

Przepisanie do znacznika przeniesienia bitu, wskazanego przez argument źródłowy, z argumentu docelowego. Ponadto instrukcja BTC powoduje jego negację, BTR jego wyzerowanie, a BTS przypisuje mu wartość jeden.

SKŁADNIA

```
BT   reg/mem, reg/stała
BTC  reg/mem, reg/stała
BTR  reg/mem, reg/stała
BTS  reg/mem, reg/stała
```

KOMENTARZ

Wyżej wymienione instrukcje są użyteczne do sprawdzania stanu podanego bitu. Po ich wykonaniu, za pomocą rozkazu JC lub JNC, można podjąć działanie zależne od stanu tego bitu.

PATRZ

BSF, BSR	*Szukanie bitu*
JC, JNC	*Rozkazy skoku zależne od stanu znacznika przeniesienia*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
BT					
reg, reg	00001111 10100011 mod,reg,r/m	bt ecx,eax	-	-	3
mem, reg	00001111 10100011 mod,reg,r/m przemieszczenie	bt Bit, ax	-	-	12
reg, stała	00001111 10100011 mod,100,r/m dana	bt ax,5	-	-	3
mem, stała	00001111 10100011 mod,100,r/m przemieszczenie dana	bt BIT,10	-	-	8
BTC, BTR, BTS					
reg, reg	00001111 10bbb011 mod,reg,r/m	btc ecx,eax	-	-	6
mem, reg	00001111 10bbb011 mod,reg,r/m przemieszczenie	btr Bit, ax	-	-	13
reg, stała	00001111 10bbb011 mod,100,r/m dana	btc ax,5	-	-	6
mem, stała	00001111 10bbb011 mod,100,r/m przemieszczenie dana	bts BIT,6	-	-	8
bbb=111 dla BTC, 110 dla BTR, 101 dla BTS					

BT, BTC, BTR, BTS

PRZYKŁAD

W edytorach tekstu każde naciśnięcie klawisza powoduje wywołanie wielu funkcji. Klawisze alfanumeryczne wymagają uaktualnienia zawartości pliku i ekranu, klawisze kursora aktualizują zawartość ekranu itp. Poniższy przykład pokazuje, w jaki sposób można wykorzystać słowo znaczników do wyznaczania, którą funkcję należy uruchomić:

```
            .MODEL SMALL
EXTRN       GetKey:PROC                 ; Funkcje te
                                        ; zdefiniowane są w
                                        ; innych modułach
EXTRN       UpdateFile:PROC
EXTRN       UpdateScrn:PROC
EXTRN       AutoSave:PROC
GETKEYFLG   EQU   1
MAXFLAG     EQU   4
            .DATA
FuncFlags   DW    0
FuncLst     DW    GetKey, UpdateFile, UpdateScrn, AutoSave
NextFlag    DW    0
            .CODE
MainLoop    PROC
            mov   ah, 1                 ; Test, czy został
                                        ; wciśnięty klawisz
            int   16h
            jnz   NoKey
            or    FuncFlags, GETKEYFLG; Ustaw znacznik
                                        ; pobrania klawisza
            mov   NextFlag, 0           ; Początkowy
                                        ; znacznik
NoKey:
            mov   bx, NextFlag
            bt    FuncFlags, bx         ; Testowanie
                                        ; znaczników
            jnc   NoFunc                ; Nie wywołuj żadnej
                                        ; funkcji
            shl   bx, 1                 ; Mnożenie przez
                                        ; rozmiar słowa
            add   bx, OFFSET FuncLst    ; Dodaj adres
                                        ; początku tablicy
            call  WORD PTR [bx]         ; Wywołaj funkcję
NoFunc:
            inc   NextFlag              ; Następny bit
            cmp   NextFlag, MAXFLAG     ; Czy to koniec
            jl    NoKey                 ; Istnieją jeszcze
                                        ; znaczniki do zbadania
            mov   NextFlag, 0           ; Stan początkowy
```

```
            jmp   MainLoop            ; Czekaj na klawisz
MainLoop    ENDP
```

Convert Byte to Word

88/86	286	386
●	●	●

CBW

ZASTOSOWANIE

Zamiana bajtu w AL na słowo w AX

SKŁADNIA

CBW

ZNACZNIKI

Bez zmian

KOMENTARZ

Przy wykonywaniu operacji arytmetycznych na liczbach ze znakiem, które mają różne rozmiary, przy ich zamianie należy uwzględnić bit znaku. Dokonuje tego właśnie instrukcja CBW.

PATRZ

CDQ, CWD, CWDE *Inne instrukcje rozszerzania danych*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011000*	cbw	2	2	3

*Dla procesora 80386 rozkazy CBW i CWDE mają ten sam kod. Natomiast w trybie 16-bitowym instrukcja CWDE jest poprzedzana przedrostkiem rozmiaru (66h). Zaś w trybie 32-bitowym instrukcja CBW jest poprzedzana przedrostkiem rozmiaru.

PRZYKŁAD

Rozkaz CBW może być wykorzystany do wyznaczenia wartości bezwzględnej danej liczby. W przykładzie do instrukcji CWD, zamiast CWD użyj CBW i wykorzystaj rejestry AL i AH zamiast AX i DX.

CDQ

Convert Double to Quad 88/86 286 386 ●

CDQ

ZASTOSOWANIE

Zamiana podwójnego słowa na słowo poczwórne

SKŁADNIA

CDQ

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja ta jest często używana do przekształcenia danej ze znakiem na liczbę 64-bitową. Liczba ta wykorzystywana jest wtedy jako dzielna, w operacji 32-bitowego dzielenia ze znakiem.

PATRZ

CBW, CWD, CWDE *Inne instrukcje rozszerzania danych*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011001*	cdq	-	-	2

*Dla procesora 80386 rozkazy CDQ i CWD mają ten sam kod. Natomiast w trybie 16-bitowym instrukcja CDQ jest poprzedzana przedrostkiem rozmiaru (66h). Zaś w trybie 32-bitowym instrukcja CWD jest poprzedzana przedrostkiem rozmiaru.

PRZYKŁAD

Przykład 32-bitowego dzielenia liczby **NumOne** przez liczbę **NumTwo**. Wynik umieszczany jest w pierwszej z nich.

```
          .MODEL SMALL
          .DATA
NumOne    DD    12345h
NumTwo    DD    45h
          .CODE
          mov   eax, NumOne    ; Załaduj rozszerzony
                               ; rejestr AX
          cdq                  ; Rozszerz ją do rejestru
                               ; EDX ( EDX=0 i EAX=12345h )
          idiv  NumTwo         ; Podziel EDX:EAX przez
                               ; NumTwo (45h)
          mov   NumOne, eax    ; Zapisz rezultat (438h)
```

CoMPare two operands

88/86	286	386
●	●	●

CMP

ZASTOSOWANIE

Porównanie dwóch argumentów.

SKŁADNIA

`CMP reg, reg`	*Porównanie zawartości dwóch rejestrów*
`CMP reg, mem`	*Porównanie zawartości rejestru i danej z pamięci*
`CMP mem, reg`	*Porównanie danej w pamięci i zawartości rejestru*
`CMP reg, stała`	*Porównanie zawartości rejestru i danej natychmiastowej*
`CMP mem, stała`	*Porównanie zawartości pamięci i danej natychmiastowej*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli rezultat nie mieści się w zakresie wyznaczonym przez argumenty
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy argument docelowy jest mniejszy niż argument źródłowy
Przeniesienie pomocnicze	Ustawiany, gdy na 3 lub 7 (argumenty 16-bitowe) pozycji pojawiła się pożyczka

KOMENTARZ

Operację porównywania należy traktować jako odejmowanie, którego rezultat nie jest nigdzie zapisywany. Ułatwia to zrozumienie sposobu ustawienia znaczników po tej operacji. Za rozkazem porównania często występuje instrukcja skoku Jwarunek, która wykorzystująca rezultat porównania.

PATRZ

SUB	*Instrukcja odejmowania*
Jwarunek	*Instrukcje skoków warunkowych*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0011100w 11,reg,r/m	cmp ax,bx	3	2	2
mem/reg	0011100w mod,reg,r/m przemieszczenie	cmp limit,bl	13/9+EA	6	6

CMP

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/mem	0011101w mod,reg,r/m przemieszczenie	cmp ax,limit	13/9+EA	7	5
reg/stała	100000sw mod,111,r/m dana	cmp dx,5	4	3	3
mem/stała	100000sw mod,111,r/m przemieszczenie dana	cmp value,6	14/10+EA	6	5
acc, stała	0011110w dana	cmp ax,55	4	3	2

PRZYKŁAD

Procedura pobiera ze zmiennej **NumList** ciąg liczb bez znaku. Jeśli liczba jest większa od 14, to rejestr CX jest zwiększany o jeden. Działanie funkcji kończy się, jeśli pobrana dana jest równa 14.

```
        mov     bx, OFFSET NumList ; Pobierz adres
                                   ; ciągu liczb
TSLoop:
        cmp     WORD PTR [bx], 14  ; Porównanie z
                                   ; liczbą 14
        je      Done               ; Koniec, gdy równe
        cmc                        ; Ustawienie znacznika
                                   ; przeniesienia dla
                                   ; liczb większych
                                   ; niż 14
        adc     cx, 0              ; Dodanie wartości
                                   ; znacznika do rejestru
        jmp     TSLoop
Done:
```

Compare String

88/86	286	386
●	●	●

CMPS, CMPSB, CMPSW

ZASTOSOWANIE

Porównanie bajtu lub słowa wskazywanego przez **DS:SI** z daną, która znajduje się pod adresem **ES:DI**. Następnie, jeśli znacznik kierunku jest wyzerowany, to rejestry indeksowe są zwiększane. W przeciwnym wypadku następuje zmniejszenie rejestrów indeksowych.

SKŁADNIA

```
CMPS [seg:]argument, argument
CMPSB
CMPSW
```

ZNACZNIKI

Nadmiar	Ustawiany, jeśli rezultat nie mieści się w zakresie wyznaczonym przez argumenty
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy argument docelowy jest mniejszy niż argument źródłowy
Przeniesienie pomocnicze	Ustawiany, gdy na 3 lub 7 (argumenty 16-bitowe) pozycji pojawiła się pożyczka

Rezultat odejmowania nie jest zapisywany w argumencie docelowym.

KOMENTARZ

Argumenty podane instrukcji CMPS są używane jedynie do wyznaczenia rozmiaru porównywanych danych oraz do zmienienia segmentu, przypisanego do argumentu źródłowego. Rozkaz ten, w celu porównania bloku danych, może zostać poprzedzony przedrostkiem REPE lub REPNE.

PATRZ

LODS, LODSB, LODSW	*Pobieranie danych z łańcucha*
MOVS, MOVSB, MOVSW	*Kopiowanie łańcuchów*
SCAS, SCASB, SCASW	*Przeszukiwanie łańcucha*
STOS, STOSB, STOSW	*Zapisywanie danych do łańcucha*
REP, REPNZ, REPZ	*Przedrostki repetycyjne*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	1010011w	cmpsb	22	8	3

CWD

PRZYKŁAD

Poniższa funkcja dokonuje operacji porównania dwóch ciągów znaków.

```
        .CODE
; Ciągi znaków wskazywane są przez DS:SI i ES:DI i oba
; zakończone są bajtem równym zero. Jeśli oba łańcuchy
; są identyczne, to funkcja zeruje znacznik przeniesienia.
StrCmp  PROC
        cld
        sub   al, al
        sub   cx, cx        ; Wyzeruj licznik
        repne scansb        ; Znajdź bajt zerowy
        add   si, cx        ; Przywróć początkową
                            ; wartość SI
        neg   cx            ; w CX liczba znaków łańcucha
        repe  cmpsb         ; Porównanie obu łańcuchów
        clc                 ; Wyzerowanie znacznika
                            ; przeniesienia
        jcxz  Match
        stc                 ; Ustaw znacznik przeniesienia
Match:
        ret
StrCmp  ENDP
```

Convert Word to Double

88/86	286	386
●	●	●

CWD

ZASTOSOWANIE

Zamiana słowa w AX na podwójne słowo w AX i DX

SKŁADNIA

CWD

ZNACZNIKI

Bez zmian

KOMENTARZ

Instrukcja ta jest często używana do przekształcenia danej ze znakiem na liczbę 32-bitową. Liczba ta wykorzystywana jest wtedy jako dzielna w operacji 16-bitowego dzielenia ze znakiem.

PATRZ

CBW, CDQ, CWDE *Inne instrukcje rozszerzania danych*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011001*	cwd	5	2	2

*Dla procesora 80386 rozkazy CWD i CDQ mają ten sam kod. Natomiast w trybie 16-bitowym instrukcja CDQ jest poprzedzana przedrostkiem rozmiaru (66h). Zaś w trybie 32-bitowym instrukcja CWD jest poprzedzana przedrostkiem rozmiaru.

PRZYKŁAD

Instrukcja CWD może być użyta do wyznaczenia wartości bezwzględnej danej liczby. Rozkaz ten umieszcza bowiem w rejestrze DX wartość -1 (0FFFFh) - jeśli liczba jest ujemna, a wartość 0 - jeśli dodatnia. Operacja XOR danej liczby z zawartością rejestru DX powoduje więc negację wszystkich jej bitów, ale tylko wtedy, gdy jest ona ujemna. Powoduje to zmianę znaku tej liczby.
Jedyną rzeczą, jaką należy jeszcze zrobić, jest dodanie jedynki do liczby ujemnej. Wynika to stąd, że liczby dodatnie zaczynają się od 0, a ujemne od -1. Ponieważ dodanie jedynki jest równoważne odjęciu wartości -1, która znajduje się przecież wtedy w rejestrze DX, to użycie instrukcji SUB kończy wykonanie operacji.
Następująca procedura, używając powyższego triku, wyznacza wartość bezwzględną liczby, która znajduje się w AX:

```
cwd          ; Rozszerzenie liczby ze znakiem do rejestru DX
xor  ax, dx  ; Negacja rejestru AX, jeśli liczba jest ujemna
sub  ax, dx  ; Dodanie jedynki, jeśli liczba była ujemna
```

Convert Word to Double Extended — 88/86 286 386 ●

CWDE

ZASTOSOWANIE

Zamiana słowa w AX na podwójne słowo w EAX.

SKŁADNIA

CWDE

ZNACZNIKI

Bez zmian.

KOMENTARZ

Działanie instrukcji CWDE jest takie samo jak rozkazu CWD, z tym że wynik zostaje umieszczony w rejestrze EAX. Jest to wygodne przy przygotowywaniu danych do operacji 32-bitowych na procesorze 80386.

DAA

PATRZ

CBW, CDQ, CWD *Inne instrukcje rozszerzania danych*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	10011001*	cwde	-	-	3

*Dla procesora 80386 rozkazy CWDE i CBW mają ten sam kod. Natomiast w trybie 16-bitowym instrukcja CWDE jest poprzedzana przedrostkiem rozmiaru (66h). Zaś w trybie 32-bitowym instrukcja CBW jest poprzedzana przedrostkiem rozmiaru.

PRZYKŁAD

Poniższy fragment kodu pobiera 16-bitową wartość zmiennej **SmallNum**, a następnie przekształca ją na daną 32-bitową w rejestrze EAX.

```
            .DATA
SmallNum    DW    8451h
            .CODE
            mov   ax, SmallNum   ; Pobierz daną do
                                 ; AX (8451h)
            cwde                 ; Rozszerz i prześlij
                                 ; do EAX (0FFFF8451h)
```

Decimal Adjust after Addition

88/86	286	386
●	●	●

DAA

ZASTOSOWANIE

Poprawka dziesiętna po dodawaniu dwójkowym w spakowanym kodzie BCD.

SKŁADNIA

DAA

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy rezultat jest większy niż 99h
Przeniesienie pomocnicze	Ustawiany, gdy rezultat jest większy niż 99h

KOMENTARZ

Instrukcja ta, korzystając z pomocniczego znacznika przeniesienia, dokonuje korekty liczby w kodzie BCD. Powinna ona być wykonana bezpośrednio za instrukcją dodawania lub też sam powinieneś nadać odpowiednią wartość temu znacznikowi.
Ponieważ na podstawie flag można określić, czy już pojawiło się przepełnienie, można dodać do siebie wiele liczb.

PATRZ

DAS	*Poprawka po odejmowaniu*
AAA, AAS	*Poprawki w rozpakowanym kodzie BCD*
AAD, AAM	*Konwersja między systemem binarnym i BCD*
ADD, ADC	*Instrukcje dodawania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	00100111	daa	4	3	4

PRZYKŁAD

Dodanie do siebie dwóch liczb w spakowanym kodzie BCD. Wynik zostaje umieszczony w zmiennej **NumOne**.

```
        .DATA
NumOne  DB  47h
NumTwo  DB  27h
        .CODE
        mov   al, NumOne    ; Pobierz pierwszą
                            ; liczbę (47h)
        add   al, NumTwo    ; Dodaj je (6Eh)
        daa                 ; Korekta wyniku (74h)
        mov   NumOne, al
```

Decimal Adjust after Substraction

88/86	286	386
●	●	●

DAS

ZASTOSOWANIE

Poprawka dziesiętna po odejmowaniu dwójkowym w spakowanym kodzie BCD.

SKŁADNIA

DAS

DAS

ZNACZNIKI

Nadmiar	Stan nieokreślony
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy rezultat jest większy niż 99h
Przeniesienie pomocnicze	Ustawiany, gdy rezultat jest większy niż 99h

KOMENTARZ

Instrukcja ta umożliwia wykonanie odejmowania liczb w spakowanym kodzie BCD bez konieczności ich zamiany na system dwójkowy. Ponieważ podczas tej operacji wykorzystywany jest pomocniczy znacznik przeniesienia, to instrukcja BCD powinna być wykonana bezpośrednio po instrukcji odejmowania.

PATRZ

DAA	*Poprawka po dodawaniu*
AAA, AAS	*Poprawki w rozpakowanym kodzie BCD*
SUB, SBB	*Instrukcje odejmowania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	00101111	das	4	3	4

PRZYKŁAD

Odjęcie od siebie dwóch liczb w spakowanym kodzie BCD. Wynik zostaje umieszczony w zmiennej **NumOne**.

```
          .DATA
NumOne    DB    24h
NumTwo    DB    7h
          .CODE
          mov   al, NumOne    ; Pobierz pierwszą
                              ; liczbę (24h)
          sub   al, NumTwo    ; Odejmij je (1Dh)
          das                 ; Korekta wyniku (17h)
          mov   NumOne, al    ; Zapisz rezultat
```

Decrement

88/86	286	386
●	●	●

DEC

ZASTOSOWANIE

Odjęcie jedynki od bajtu lub słowa

SKŁADNIA

`DEC reg`	*Dekrementacja rejestru*
`DEC mem`	*Dekrementacja danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli rezultat traktowany jako liczba ze znakie jest poza zakresem, wyznaczonym przez rozmiar argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Bez zmian
Przeniesienie pomocnicze	Ustawiany, gdy pojawiło się przeniesienie z pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Instrukcja ta powoduje odjęcie jedynki od swojego argumentu. Jej użycie oszczędza pamięć i skraca czas wykonania programu.

Tylko w jednym wypadku należy użyć rozkazu SUB. Jeśli interesuje nas znacznik przeniesienia po wykonaniu odjęcia, to powinniśmy wykorzystać rozkaz SUB, gdyż instrukcja dekrementacji nie zmienia stanu tego znacznika.

PATRZ

INC	*Dodanie jedynki*
SUB	*Instrukcja odejmowania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11111110 11001,r/m	dec al	3	2	2
mem	1111111w mod,001,r/m przemieszczenie	dec counter	23/15+EA	7	6
reg16/32*	01001reg	dec ax	3	2	2

*Wersja 32-bitowa rozkazu dotyczy tylko procesora 80386

DIV

PRZYKŁAD

Procedura odczytuje port wejścia/wyjścia o numerze **3F8h** i zmniejsza zawartość rejestru BX po każdorazowym pobraniu znaku 0Dh. Gdy rejestr BX przyjmie wartość zero, to procedura kończy swoje działanie.

```
SDLoop:
        in      al, 3F8h        ; Pobierz bajt z portu
                                ; danych COM1
        cmp     al, 0Dh         ; Czy to znak 0Dh
        jne     SDLoop          ; Jeśli nie, czekaj w pętli
        dec     bx              ; Zmniejsz BX
        jnz     SDLoop          ; Wykonuj pętlę, dopóki BX
                                ; jest różny od zera
```

Divide

88/86	286	386
●	●	●

DIV

ZASTOSOWANIE

Dzielenie liczb bez znaku

SKŁADNIA

`DIV reg`	*Dzielenie przez daną z rejestru*
`DIV mem`	*Dzielenie przez daną z pamięci*

ZNACZNIKI

Znaczniki nadmiaru, znaku, zera, obu przeniesień oraz parzystości przyjmują wartości nieokreślone.

KOMENTARZ

Instrukcja DIV przyjmuje, że dzielna znajduje się w akumulatorze. Przy dzieleniu przez wartość 8-bitową jest nim rejestr AX. Natomiast przy dzieleniu przez daną 16-bitową dzielna znajduje się w rejestrach DX i AX. Dodatkowo w procesorach 80386 dzielnik może być daną 32-bitową. Wtedy argumentem domyślnym stają się rejestry EDX i EAX.

Przy dzieleniu przez potęgę liczby dwa (2, 4, 8, itd.) należy również rozpatrzyć możliwość użycia do tego celu instrukcji przesuwających. Są one o wiele bardziej efektywne, szczególnie na procesorach 8088 i 8086.

Dzielenie przez zero lub liczbę, która powoduje, że wynik dzielenia nie mieści się w argumencie docelowym, powoduje wygenerowanie przerwania. Najczęściej powoduje ono awaryjne zakończenie programu. Dobrą praktyką jest sprawdzanie poprawności danych jeszcze przed wykonaniem tej instrukcji. Najprostszym testem jest porównanie dzielnika ze starszym bajtem lub słowem dzielnej. Jeśli jest on mniejszy, wówczas oznacza to, że jest albo równy zeru, albo spowoduje powstanie nadmiaru.

PATRZ

IDIV	*Dzielenie liczb ze znakiem*
IMUL, MUL	*Instrukcje mnożenia*
SHR, SAR	*Instrukcje przesuwania w prawo*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 mod,110,r/m	div bl	80-90	14	14
reg16	11110111 mod,110,r/m	div bx	144-162	22	22
reg32	11110111 mod,110,r/m	div ebx	-	-	38
mem8	11110110 mod,110,r/m przemieszczenie	div bval	86-96+EA	17	17
mem16	11110111 mod,110,r/m przemieszczenie	div wval	150-176+EA	25	25
mem32	11110111 mod,110,r/m przemieszczenie	div dval	-	-	41

PRZYKŁAD

Procedura dzieli zmienną **BigNum** przez liczbę zawartą w BX, a rezultat działania umieszcza w zmiennej **SmallNum**.

```
          .DATA
BigNum     DD   214561h
SmallNum   DW   ?
           .CODE
           mov   bx, 50h
           mov   ax, BigNum    ; Pobierz młodsze
                               ; słowo (4561h)
           mov   dx, BigNum+2  ; Pobierz starsze
                               ; słowo (21h)
           cmp   dx, bx        ; Czy dane poprawne
           jge   err           ; Jeśli DX >= dzielnik, błąd
           div   bx            ; Dzielenie
                               ; ( AX = 6A77h DX = 31h )
           mov   SmallNum, ax  ; Zapisz wynik
```

```
err:
```

Integer Divide	88/86	286	386
	●	●	●

IDIV

ZASTOSOWANIE

Dzielenie liczb ze znakiem

SKŁADNIA

`IDIV reg`	*Dzielenie przez daną z rejestru*
`IDIV mem`	*Dzielenie przez daną z pamięci*

ZNACZNIKI

Znaczniki nadmiaru, znaku, zera, obu przeniesień oraz parzystości przyjmują wartości nieokreślone.

KOMENTARZ

Instrukcja IDIV wykonuje dzielenie również na liczbach ujemnych. Ponieważ rozmiar dzielnej musi być zawsze dwa razy większy niż ilorazu, to należy pamiętać o jej prawidłowym rozszerzeniu przed wykonaniem operacji.

Przy dzieleniu przez potęgę liczby dwa (2, 4, 8, itd.) należy również rozpatrzyć możliwość użycia do tego celu instrukcji SAR. Jest to efektywniejszy sposób, szczególnie na procesorach 8088 i 8086.

Dzielenie przez zero lub liczbę, która powoduje, że wynik dzielenia nie mieści się w argumencie docelowym, powoduje wygenerowanie przerwania. Najczęściej powoduje ono awaryjne zakończenie programu. Dobrą praktyką jest sprawdzanie poprawności danych jeszcze przed wykonaniem dzielenia.

PATRZ

DIV	*Dzielenie liczb bez znaku*
IMUL, MUL	*Instrukcje mnożenia*
SHR, SAR	*Instrukcje przesuwania w prawo*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 mod,111,r/m	idiv cl	101-112	17	19

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg16	11110111 mod,111,r/m	idiv cx	165-184	25	27
reg32	11110111 mod,111,r/m	idiv ecx	-	-	43
mem8	11110110 mod,111,r/m przemieszczenie	idiv bval	107-118+EA	20	22
mem16	11110111 mod,111,r/m przemieszczenie	idiv wval	171-194+EA	28	30
mem32	11110111 mod,111,r/m przemieszczenie	idiv dval	-	-	46

PRZYKŁAD

Patrz przykład do instrukcji CBW

Integer Multiply

88/86	286	386
●	●	●

IMUL

ZASTOSOWANIE

Mnożenie liczb ze znakiem

SKŁADNIA

`IMUL reg`	*Mnożenie przez rejestr*
`IMUL mem`	*Mnożenie przez zawartość pamięci*

Ponadto w procesorze 80386:

`IMUL reg, stała`	*Mnożenie rejestru przez stałą*
`IMUL reg1, reg2, stała`	*Mnożenie rejestru reg2 przez stałą i przesłanie rezultatu do reg1*
`IMUL reg, mem, stała`	*Mnożenie zawartości pamięci przez stałą i przesłanie rezultatu do reg1*
`IMUL reg1, reg2`	*Mnożenie dwóch rejestrów*
`IMUL reg, mem`	*Mnożenie rejestru i zawartości pamięci*

IMUL

ZNACZNIKI

Nadmiar	Ustawiany, gdy wynik jest większy niż argumenty
Znak	Stan nieokreślony
Zero	Stan nieokreślony
Parzystość	Stan nieokreślony
Przeniesienie	Ustawiany, gdy wynik jest większy niż argumenty
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Instrukcja mnożenia używa, przy pobraniu danych i przy umieszczaniu wyniku, argumentów niejawnych. Dla 8-bitowych czynników jeden z nich pobierany jest z rejestru AL, a iloczyn przesyłany jest do AX. Przy mnożeniu danych 16-bitowych jeden z argumentów znajduje się w rejestrze AX, a rezultat operacji jest umieszczany w rejestrach DX i AX. Ponadto dla procesora 80386 istnieje jeszcze mnożenie liczb 32-bitowych. Wtedy jedna z nich znajduje się w rejestrze EAX, a iloczyn zapisywany jest do pary EDX, EAX.
W procesorze 80386 istnieje kilka dodatkowych trybów adresowania. Pozwalają one na pomnożenie rejestru czy zawartości pamięci przez siebie lub przez stałą. W tych wypadkach należy pamiętać, że iloczyn jest zapisywany w miejscu, które ma taki sam rozmiar jak argumenty. Oznacza to, że gdy iloczyn będzie zbyt duży jego starsze bity będą obcięte. Wtedy też zostaną ustawione znaczniki przeniesienia i nadmiaru.

PATRZ

MUL	*Mnożenie liczb bez znaku*
DIV, IDIV	*Instrukcje dzielenia*
SHL, SAL	*Instrukcje przesuwania w lewo*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 mod,101,r/m	imul dl	80-98	13	9-14
reg16	11110111 mod,101,r/m	imul bx	128-154	12	9-22
reg32	11110111 mod,101,r/m	imul ecx	-	-	9-38
mem8	11110110 mod,101,r/m przemieszczenie	imul bval	86-104+EA	16	12-17
mem16	11110111 mod,101,r/m przemieszczenie	imul wval	134-164+EA	24	12-25
mem32	11110111 mod,101,r/m przemieszczenie	imul dval	-	-	12-41

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, stała	011010s1 mod,reg,r/m stała	imul cx,4	-	21	9-38
reg, reg, stała	001010s1 mod,reg,r/m stała	imul cx,ax,5	-	21	9-38
reg, mem, stała	001010s1 mod,reg,r/m przemieszczenie stała	imul cx,wval,7	-	24	12-41
reg, reg	00001111 10101111 mod,reg,r/m	imul cx,ax	-	-	9-38
reg, mem lub mem, reg	00001111 10101111 mod,reg,r/m przemieszczenie	imul ax,wval	-	-	12-41

PRZYKŁAD

Procedura wykonuje mnożenie liczb **NumOne** przez **NumTwo**. Jeśli rezultat jest zbyt duży, to wynikiem operacji jest zero. Wynik jest wpisywany do zmiennej **NumOne**.

```
            .DATA
NumOne      DW    1234h
NumTwo      DW    21h
            .CODE
            mov   ax, NumOne
            imul  NumTwo        ; AX = 58B4h, DX = 2h Bit
                                ; przeniesienia ustawiony
            jnc   SaveIt        ; Jeśli nie jest ustawiony
                                ; znacznik przeniesienia,
                                ; wówczas  rezultat zmieści
                                ; się w argumencie
            xor   ax, ax
SaveIt:
            mov   NumOne, ax
```

INC

Increment

88/86	286	386
●	●	●

INC

ZASTOSOWANIE

Dodanie jedynki do bajtu lub słowa

SKŁADNIA

`INC reg`	*Inkrementacja rejestru*
`INC mem`	*Inkrementacja danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli rezultat, traktowany jako liczba ze znakiem jest poza zakresem wyznaczonym przez rozmiar argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Bez zmian
Przeniesienie pomocnicze	Ustawiany, gdy pojawiło się przeniesienie z pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Instrukcja INC powoduje dodanie jedynki do swojego argumentu. Jej użycie oszczędza pamięć i skraca czas wykonania programu.

PATRZ

DEC	*Odjęcie jedynki*
ADD	*Instrukcja dodawania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11111110 11000,r/m	inc cl	3	2	2
mem	1111111w mod,000,r/m przemieszczenie	inc counter	23/15+EA	7	6
reg16	01000reg	inc ax	3	2	2
reg32	01000reg	inc edx	-	-	2

PRZYKŁAD

Poniższa procedura zlicza słowa w liście. Listę kończy słowo o wartości 0.

```
        sub     cx, cx          ; Wyzeruj licznik
```

```
            mov     bx, OFFSET List
WCLoop:
            cmp     [bx], 0             ; Porównaj z zerem
            je      Done
            inc     cx                  ; Zwiększ licznik
            inc     bx                  ; Zwiększ rejestr BX,
                                        ; by wskazywał na
            inc     bx                  ; następne słowo
            jmp     SHORT WCLoop
Done:
            ret
```

Multiply

88/86	286	386
●	●	●

MUL

ZASTOSOWANIE

Mnożenie liczb bez znaku

SKŁADNIA

`MUL reg`	*Mnożenie przez rejestr*
`MUL mem`	*Mnożenie przez zawartość pamięci*

ZNACZNIKI

Znaczniki nadmiaru i przeniesienia są ustawione wtedy, gdy rozmiar wyniku jest większy niż rozmiar argumentów.

Stan znaczników znaku, zera, przeniesienia pomocniczego i parzystości jest nieokreślony.

KOMENTARZ

Instrukcja mnożenia używa, przy pobraniu danych i przy umieszczaniu wyniku, argumentów niejawnych. Dla 8-bitowych czynników jeden z nich pobierany jest z rejestru AL, a iloczyn przesyłany jest do AX. Przy mnożeniu danych 16-bitowych jeden z argumentów znajduje się w rejestrze AX, a rezultat operacji jest umieszczany w rejestrach DX i AX. Ponadto dla procesora 80386 istnieje jeszcze mnożenie liczb 32-bitowych. Wtedy jedna z nich znajduje się w rejestrze EAX, a iloczyn zapisywany jest do pary EDX, EAX. Jeśli wartość w rejestrze EDX jest różna od zera, to zostaną ustawione znaczniki przeniesienia i nadmiaru.

PATRZ

IMUL	*Mnożenie liczb ze znakiem*
SHL, SAL	*Instrukcje przesuwania w lewo*

NEG

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 mod,100,r/m	mul cl	70-77	13	9-14
reg16	11110111 mod,100,r/m	mul dx	118-133	21	9-22
reg32	11110111 mod,100,r/m	mul ebx	-	-	9-38
mem8	11110110 mod,100,r/m przemieszczenie	mul bval	76-83+EA	16	12-17
mem16	11110111 mod,100,r/m przemieszczenie	mul wval	124-139+EA	24	12-25
mem32	11110111 mod,100,r/m przemieszczenie	mul dval	-	-	12-41

PRZYKŁAD

By znaleźć rozmiar tablicy w bajtach, należy pomnożyć liczbę jej elementów przez rozmiar pojedynczego elementu. Dzięki użyciu instrukcji MUL operacja ta jest bardzo prosta.

```
mov   ax, RecCnt          ; Liczba elementów
mov   bx, RecSize         ; Rozmiar elementu
mul   bx
mov   ArraySize, ax       ; Zapisz młodsze słowo wyniku
mov   ArraySize+2, dx     ; Zapisz starsze słowo
```

Negate

88/86	286	386
●	●	●

NEG

ZASTOSOWANIE

Zmiana znaku liczby

SKŁADNIA

NEG	reg	*Zmiana znaku danej w rejestrze*
NEG	mem	*Zmiana znaku danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, gdy w wyniku znajdują się wszystkie bity równe jeden
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, jeśli wynikiem operacji nie jest zero

KOMENTARZ

Wykonanie rozkazu jest równoznaczne z obliczeniem dla argumentu uzupełnienia do dwóch. Operacja ta polega na zanegowaniu wszystkich bitów argumentu, a następnie dodaniu od tak otrzymanego wyniku jedynki.

PATRZ

NOT *Logiczna instrukcja NOT*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 11,011,r/m	neg al	3	2	2
reg16	11110111 11,011,r/m	neg cx	3	2	2
reg32	11110111 11,011,r/m	neg eax	-	-	2
mem8	11110110 mod,011,r/m przemieszczenie	neg bval	24/16+EA	7	6
mem16	11110111 mod,011,r/m przemieszczenie	neg wval	24/16+EA	7	6
mem32	11110111 mod,011,r/m przemieszczenie	neg dval	-	-	6

PRZYKŁAD

Funkcja wyznacza wartość bezwzględną zmiennej **SignNum**.

```
        mov     ax, SignNum
        neg     ax
        js      Done                ; Sprawdź znak wyniku
        mov     SignNum, ax         ; Zapisz wynik
Done:
        ret
```

Jeśli wartość zmiennej **SignNum** wynosi -45, to instrukcja NEG zmienia ją na 45. Rozkaz JS jest rozkazem skoku warunkowego. Skok jest wykonywany, gdy znacznik znaku jest

ustawiony. W przeciwnym wypadku liczba jest dodatnia, a więc powinna być zapisana do zmiennej **SignNum**.
Bardziej efektywny sposób obliczania wartości bezwzględnej jest przedstawiony w przykładzie do instrukcji CWD.

Logical NOT

88/86	286	386
●	●	●

NOT

ZASTOSOWANIE

Negacja każdego bitu liczby

SKŁADNIA

`NOT reg`	*Negacja danej w rejestrze*
`NOT mem`	*Negacja danej w pamięci*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja NOT jest bardzo często stosowana w wyrażeniach logicznych. Używana jest również do negacji masek, wykorzystywanych przez rozkazy AND oraz OR.

PATRZ

AND, OR, XOR	*Instrukcje logiczne*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg8	11110110 11,010,r/m	not al	3	2	2
reg16	11110111 11,010,r/m	not ax	3	2	2
reg32	11110111 11,010,r/m	not eax	-	-	2
mem8	11110110 mod,010,r/m przemieszczenie	not bval	24/16+EA	7	6
mem16	11110111 mod,010,r/m przemieszczenie	not wval	24/16+EA	7	6

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
mem32	11110111 mod,010,r/m przemieszczenie	not dval	-	-	6

PRZYKŁAD

W przykładzie maskowany jest bit rejestru AL. Bit ten w masce **Mask** ma wartość 1.

```
        .DATA
Mask    DB    00010000b
        .CODE
         mov    bl, Mask        ; Pobierz maskę
         not    bl              ; Negacja maski (11101111b)
         and    al, bl          ; wyzerowanie bitu
```

Inclusive OR

88/86	286	386
●	●	●

OR

ZASTOSOWANIE

Suma logiczna argumentów

SKŁADNIA

OR	reg, reg	*Suma logiczna zawartości dwóch rejestrów*
OR	reg, mem	*Suma logiczna zawartości rejestru i danej z pamięci*
OR	mem, reg	*Suma logiczna danej w pamięci i zawartości rejestru*
OR	reg, stała	*Suma logiczna zawartości rejestru i danej natychmiastowej*
OR	mem, stała	*Suma logiczna zawartości pamięci i danej natychmiastowej*

ZNACZNIKI

Nadmiar	Zerowany
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Zerowany
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Instrukcja OR jest najczęściej stosowana do ustawiania bitów w bajcie

PATRZ

AND, XOR, NOT — *Pozostałe instrukcje logiczne*

RCL

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0000100w 11,reg,r/m	or ax,bx	3	2	2
mem/reg	0000100w mod,reg,r/m przemieszczenie	or flags,cx	24/16+EA	7	7
reg/mem	0000101w mod,reg,r/m przemieszczenie	or bx,mask	24/16+EA	7	7
reg/stała	100000sw mod,001,r/m dana	or bl,0fh	4	3	2
mem/stała	100000sw mod,001,r/m przemieszczenie dana	or flags,3	25/17+EA	7	7
acc, stała	0000110w dana	or al,4	4	3	2

PRZYKŁAD

Patrz przykład do instrukcji AND.

Rotate through Cary Left

88/86	286	386
●	●	●

RCL

ZASTOSOWANIE

Przesunięcie cykliczne w lewo bajtu lub słowa z przeniesieniem.

SKŁADNIA

`RCL reg/mem, cl` *Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w lewo*

`RCL reg/mem, stała` *Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w lewo. Liczba ta może być równa:*
1 dla procesorów 8088, 8086
0-31 dla procesorów 80286 i 80386

ZNACZNIKI

Znacznik nadmiaru jest ustawiany, gdy pojedyncza rotacja powoduje zmianę znaku. Znacznik przeniesienia przyjmuje wartość jaką miał najstarszy bit danej przed wykonaniem operacji rotacji.

KOMENTARZ

Instrukcja RCL powoduje przesunięcie w lewo wszystkich bitów argumentu. Najstarszy bit wpisywany jest do znacznika przeniesienia, a poprzednią wartością tego znacznika uzupełniany jest bit najmłodszy. Jeśli drugim argumentem jest jedynka, to dana jest przesuwana o jedną pozycję. Drugim argumentem może być również rejestr **CL** i to właśnie on określa wtedy liczbę bitów do przesunięcia. W procesorach 80286 i 80386 drugi argument może być ponadto 8-bitową liczbą (0-31), która określa żądaną liczbę rotacji.

PATRZ

ROL, ROR, RCR	*Inne instrukcje przesunięć cyklicznych*
SHL, SHR, SAL, SAR	*Instrukcje przesuwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101000w 11010,r/m	rcl bx,1	2	2	3
mem, 1	1101000w mod,010,r/m	rcl value,1	23/15+EA	7	7
reg, cl	1101001w 11010,r/m	rcl bx,cl	8+4n	5+n	9
mem, cl	1101001w mod,010,r/m	rcl value,cl	28/20+EA+4n	8n	10
reg, stała	1100000w 11010,r/m	rcl bx,4	-	5+n	10
mem, stała	1100000w mod,010,r/m	rcl value, 10	-	8+n	9

PRZYKŁAD

Poniższy przykład pokazuje, jak użyć instrukcji RCL do przesunięcia 32-bitowej danej:

```
.
.
.
shl ax, 1      ; Przesuń młodsze słowo, najstarszy bit
               ; skopiuj do znacznika przeniesienia
rcl dx, 1      ; Przesuń starsze słowo
.
.
.
```

RCR

Rotate through Cary Right

88/86	286	386
●	●	●

RCR

ZASTOSOWANIE

Przesunięcie cykliczne w prawo bajtu lub słowa z przeniesieniem.

SKŁADNIA

RCR	reg/mem, cl	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w prawo*
RCR	reg/mem, stała	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w prawo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ZNACZNIKI

Znacznik nadmiaru jest ustawiany, gdy pojedyncza rotacja powoduje zmianę znaku. Znacznik przeniesienia przyjmuje wartość jaką miał najstarszy bit danej przed wykonaniem operacji rotacji.

KOMENTARZ

Instrukcja RCR powoduje przesunięcie w prawo wszystkich bitów argumentu. Najmłodszy bit wpisywany jest do znacznika przeniesienia, a poprzednią wartością tego znacznika uzupełniany jest bit najstarszy. Jeśli drugim argumentem jest jedynka, to dana jest przesuwana o jedną pozycję. Drugim argumentem może być również rejestr **CL** i to właśnie on określa wtedy liczbę bitów do przesunięcia. W procesorach 80286 i 80386 drugi argument może być ponadto 8-bitową liczbą (0-31), która określa żądaną liczbę rotacji.

PATRZ

ROL, ROR, RCL	*Inne instrukcje przesunięć cyklicznych*
SHL, SHR, SAL, SAR	*Instrukcje przesuwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101000w 11011,r/m	rcr bx,1	2	2	3
mem, 1	1101000w mod,011,r/m przemieszczenie	rcr value,1	23/15+EA	7	7
reg, cl	1101001w 11011,r/m	rcr bx,cl	8+4n	5+n	3

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
mem, cl	1101001w mod,011,r/m przemieszczenie	rcr value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11010,r/m stała	rcr bx,4	-	5+n	3
mem, stała	1100000w mod,011,r/m stała	rcr value, 10	-	8+n	3

PRZYKŁAD

Poniższy przykład pokazuje, jak użyć instrukcji RCR do przesunięcia 32-bitowej danej:

```
.
.
.
shr   ax, 1          ; Przesuń młodsze słowo,
                     ; najmłodszy bit skopiuj do
                     ; znacznika przeniesienia
rcr   dx, 1          ; Przesuń starsze słowo
.
.
.
```

Rotate Left

88/86	286	386
●	●	●

ROL

ZASTOSOWANIE

Przesunięcie cykliczne w lewo bajtu lub słowa.

SKŁADNIA

`ROL reg/mem, cl`	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w lewo*
`ROL reg/mem, stała`	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w lewo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ROL

ZNACZNIKI

Znacznik nadmiaru jest ustawiany, gdy pojedyncza rotacja powoduje zmianę znaku. Znacznik przeniesienia przyjmuje wartość jaką miał najstarszy bit danej przed wykonaniem operacji rotacji.

KOMENTARZ

Instrukcja ROL powoduje przesunięcie w lewo wszystkich bitów argumentu. Najstarszy bit wpisywany jest do znacznika przeniesienia oraz jego wartością uzupełniany jest bit najmłodszy. Jeśli drugim argumentem jest jedynka, to dana jest przesuwana o jedną pozycję. Drugim argumentem może być również rejestr **CL** i to właśnie on określa wtedy liczbę bitów do przesunięcia. W procesorach 80286 i 80386 drugi argument może być ponadto 8-bitową liczbą (0-31), która określa żądaną liczbę rotacji.

PATRZ

RCL, ROR, RCR	*Inne instrukcje przesunięć cyklicznych*
SHL, SHR, SAL, SAR	*Instrukcje przesuwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101000w 11000,r/m	rol bx,1	2	2	3
mem, 1	1101000w mod,000,r/m	rol value,1	23/15+EA	7	7
reg, cl	1101001w 11000,r/m	rol bx,cl	8+4n	5+n	3
mem, cl	1101001w mod,000,r/m	rol value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11000,r/m	rol bx,4	-	5+n	3
mem, stała	1100000w mod,000,r/m	rol value, 10	-	8+n	3

PRZYKŁAD

Funkcja zamienia liczbę w kodzie BCD na dwa znaki ASCII. Liczba jest podawana w rejestrze AL, a wynik zostaje umieszczony w AX.

```
.
.
.
mov   al, 34h        ; w rejestrze AL umieść
                     ; liczbę
call  BCD2ASCII      ; Zamień ją
```

```
            .
            .
            .
BCD2ASCII   PROC
            mov   ah, al
            rol   ah, 1          ; Zamień kolejność
                                 ; półbajtów
            rol   ah, 1
            rol   ah, 1
            rol   ah, 1          ; AX = 4334h
            and   ax, 0f0fh      ; Wyzeruj starsze bity
                                 ; w obu bajtach AX = 0304h
            or    ax, 3030h      ; Przekształć na kod
                                 ; ASCII AX = 3334h
            ret
BCD2ASCII   ENDP
```

Rotate Right

88/86	286	386
●	●	●

ROR

ZASTOSOWANIE

Przesunięcie cykliczne w prawo bajtu lub słowa.

SKŁADNIA

`ROR reg/mem, cl`	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w prawo*
`ROR reg/mem, stała`	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w prawo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ZNACZNIKI

Znacznik nadmiaru jest ustawiany, gdy pojedyncza rotacja powoduje zmianę znaku. Znacznik przeniesienia przyjmuje wartość jaką miał najstarszy bit danej przed wykonaniem operacji rotacji.

KOMENTARZ

Instrukcja ROR powoduje przesunięcie w prawo wszystkich bitów argumentu. Najmłodszy bit wpisywany jest do znacznika przeniesienia oraz jego wartością uzupełniany jest bit najstarszy. Jeśli drugim argumentem jest jedynka, to dana jest przesuwana o jedną pozycję. Drugim argumentem może być również rejestr **CL** i to właśnie on określa wtedy liczbę

bitów do przesunięcia. W procesorach 80286 i 80386 drugi argument może być ponadto 8-bitową liczbą (0-31), która określa żądaną liczbę rotacji.

PATRZ

RCL, ROL, RCR	*Inne instrukcje przesunięć cyklicznych*
SHL, SHR, SAL, SAR	*Instrukcje przesuwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101000w 11001,r/m	ror bx,1	2	2	3
mem	1101000w mod,001,r/m przemieszczenie	ror value,1	23/15+EA	7	7
reg, cl	1101001w 11001,r/m	ror bx,cl	8+4n	5+n	3
mem, cl	1101001w mod,001,r/m przemieszczenie	ror value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11001,r/m	ror bx,4	-	5+n	3
mem, stała	1100000w mod,001,r/m	ror value, 10	-	8+n	3

PRZYKŁAD

Funkcja zamienia liczbę w kodzie BCD na dwa znaki ASCII. Liczba jest podawana w rejestrze AL, a wynik zostaje umieszczony w AX.

```
            .
            .
            .
            mov   al, 34h     ; w rejestrze AL umieść
                              ; liczbę
            call  BCD2ASCII   ; Zamień ją
            .
            .
            .
BCD2ASCII   PROC
            mov   ah, al
            ror   ah, 1       ; Zamień kolejność
                              ; półbajtów
            ror   ah, 1
            ror   ah, 1
```

```
            ror     ah, 1       ; AX = 4334h
            and     ax, 0f0fh   ; Wyzeruj starsze bity
                                ; w obu bajtach AX = 0304h
            or      ax, 3030h   ; Przekształć na kod ASCII
                                ; AX = 3334h
            ret
BCD2ASCII   ENDP
```

Shift Arithmetic Left

88/86	286	386
●	●	●

SAL

ZASTOSOWANIE

Przesunięcie w lewo bajtu lub słowa. Operacja ta jest równoważna mnożeniu przez dwa.

SKŁADNIA

`SAL reg/mem, cl`	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w lewo*
`SAL reg/mem, stała`	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w lewo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli zmienił się znak argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy najstarszy bit argumentu był jedynką
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Zastosowanie instrukcji SAL do mnożenia przez potęgę liczby dwa jest bardziej efektywne niż użycie rozkazu MUL. Instrukcja SAL jest identyczna z instrukcją SHL.

PATRZ

RCL, RCR, ROR, RCR	*Inne instrukcje przesunięć cyklicznych*
SHR, SAL, SAR	*Instrukcje przesuwania*
MUL	

SAL

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101100w 11100,r/m	sal bx,1	2	2	3
mem, 1	1101100w mod,100,r/m przemieszczenie	sal value,1	23/15+EA	7	7
reg, cl	1101001w 11100,r/m	sal bx,cl	8+4n	5+n	3
mem, cl	1101001w mod,100,r/m przemieszczenie	sal value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11100,r/m dana	sal bx,4	-	5+n	3
mem, stała	1100000w mod,100,r/m dana	sal value, 10	-	8+n	3

PRZYKŁAD

Poniższy podprogram mnoży przez 4 32-bitową liczbę ze znakiem, która to liczba znajduje się w rejestrach AX i DX. Można do tego celu użyć instrukcji MUL, ale dla procesorów 8088 i 8086 poniższy sposób jest o wiele szybszy.

```
mov   dx, 41h      ; Starsze słowo równa się 41h
mov   ax, 1234h    ; Młodsze słowo równa się 1234h
sal   ax, 1        ; Pomnóż przez dwa (2468h)
rcl   dx, 1        ; Rotacja z wykorzystaniem
                   ; znacznika przeniesienia
                   ; ustawionego przez poprzednią
                   ; operację (0082h)
sal   ax, 1        ; Powtórz powyższe działania
                   ; jeszcze raz by pomnożyć przez
                   ; cztery
rcl   dx, 1        ; dx = 0104h ax = 48D0h
```

Shift Arithmetic Right

88/86	286	386
●	●	●

SAR

ZASTOSOWANIE

Przesunięcie w prawo bajtu lub słowa. Operacja ta jest równoważna dzieleniu przez dwa. Ponieważ nie zmienia ona bitu znaku, to może być użyta do dzielenia liczb ze znakiem.

SKŁADNIA

`SAR reg/mem, cl`	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w prawo*
`SAR reg/mem, stała`	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w prawo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli zmienił się znak argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy najmłodszy bit argumentu był jedynką
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Zastosowanie instrukcji SAR do dzielenia przez potęgę liczby dwa jest bardziej efektywne niż użycie rozkazu IDIV. Powinna ona być wykorzystywana przy działaniu na liczbach ze znakiem.

PATRZ

RCL, RCR, ROR, RCR	*Inne instrukcje przesunięć cyklicznych*
SHL, SAL, SHR	*Instrukcje przesuwania*
IDIV	

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101100w 11111,r/m	sar bx,1	2	2	3
mem, 1	1101100w mod,111,r/m przemieszczenie	sar value,1	23/15+EA	7	7
reg, cl	1101001w 11111,r/m	sar bx,cl	8+4n	5+n	3

SHL

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
mem, cl	1101001w mod,111,r/m przemieszczenie	sar value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11111,r/m dana	sar bx,4	-	5+n	3
mem, stała	1100000w mod,111,r/m dana	sar value, 10	-	8+n	3

PRZYKŁAD

Poniższy podprogram dzieli przez 4 32-bitową liczbę ze znakiem, która to liczba znajduje się w rejestrach AX i DX. Można do tego celu użyć instrukcji IDIV, ale dla procesorów 8088 i 8086 poniższy sposób jest o wiele szybszy.

```
mov   dx, 41h      ; Starsze słowo równa się 41h
mov   ax, 1234h    ; Młodsze słowo równa się 1234h
sar   dx, 1        ; Podziel przez dwa (2h)
rcr   ax, 1        ; Rotacja z wykorzystaniem
                   ; znacznika przeniesienia,
                   ; ustawionego przez poprzednią
                   ; operację (811Ah)
sar   dx, 1        ; Powtórz powyższe działania
                   ; jeszcze raz by podzielić
                   ; przez cztery
rcr   ax, 1        ; dx = 1 ax = 408Dh
```

Shift Left

88/86	286	386
●	●	●

SHL

Patrz opis instrukcji SAL.

Shift Left Double Precision 88/86 286 386 ●

SHLD

ZASTOSOWANIE

Przesunięcie podwójnej precyzji w lewo.

SKŁADNIA

`SHLD reg, reg, stała`	*Przesunięcie zawartości dwóch rejestrów o podaną liczbę bitów w lewo*
`SHLD mem, reg, stała`	*Przesunięcie zawartości rejestru do danej w pamięci*
`SHLD reg, reg, cl`	*Przesunięcie zawartości dwóch rejestrów o podaną w cl liczbę bitów*
`SHLD mem, reg, cl`	*Przesunięcie zawartości rejestru do danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli zmienił się znak argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany na wartość ostatnio wysuniętego bitu
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Instrukcja ta może być pomocna przy operowaniu na dużych liczbach. Ponieważ bity wysuwane z jednego są umieszczane w drugim rejestrze, to jest ona wygodna do sprawdzania wartości poszczególnych części słowa.

PATRZ

RCL, RCR, ROR, RCR	*Inne instrukcje przesunięć cyklicznych*
SHL, SAL, SHR, SAR	*Instrukcje przesuwania*
SHRD	*Instrukcja przesuwania w prawo w podwójnej precyzji*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg, stała	00001111 10100100 11,reg,r/m dana	shld ax,bx,4	-	-	3

SHR

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
mem, reg, stała	00001111 10100100 mod,reg,r/m przemieszczenie dana	shld value,cx,3	-	-	7
reg, reg, cl	00001111 10100101 11,reg,r/m	shld ax,bx,cl	-	-	3
mem, reg, cl	00001111 10100101 mod,reg,r/m przemieszczenie	shld value,ax,cl	-	-	7

PRZYKŁAD

Patrz przykład do instrukcji SHRD. Użycie instrukcji SHLD zamiast rozkazu SHRD spowoduje wyświetlenie liczb w normalnej kolejności.

Shift Right

88/86	286	386
●	●	●

SHR

ZASTOSOWANIE

Przesunięcie w prawo bajtu lub słowa. Operacja ta jest równoważna dzieleniu przez dwa.

SKŁADNIA

`SHR reg/mem, cl`	*Przesunięcie zawartości rejestru lub danej w pamięci o cl bitów w prawo*
`SHR reg/mem, stała`	*Przesunięcie zawartości rejestru lub danej w pamięci o pewną liczbę bitów w prawo. Liczba ta może być równa:* *1 dla procesorów 8088, 8086* *0-31 dla procesorów 80286 i 80386*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli zmienił się znak argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy najmłodszy bit argumentu był jedynką
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Zastosowanie instrukcji SHR do dzielenia przez potęgę liczby dwa jest bardziej efektywne niż użycie rozkazu DIV.

PATRZ

RCL, RCR, ROR, RCR	*Instrukcje przesunięć cyklicznych*
SHL, SAL, SAR	*Instrukcje przesuwania*
IDIV	

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, 1	1101100w 11101,r/m	shr bx,1	2	2	3
mem, 1	1101100w mod,101,r/m przemieszczenie	shr value,1	23/15+EA	7	7
reg, cl	1101001w 11101,r/m	shr bx,cl	8+4n	5+n	3
mem, cl	1101001w mod,101,r/m przemieszczenie	shr value,cl	28/20+EA+4n	8n	7
reg, stała	1100000w 11101,r/m dana	shr bx,4	-	5+n	3
mem, stała	1100000w mod,101,r/m dana	shr value, 10	-	8+n	3

PRZYKŁAD

Poniższy podprogram dzieli przez 4 32-bitową liczbę, która znajduje się w rejestrach AX i DX. Można wprawdzie w tym celu użyć instrukcji IDIV, ale dla procesorów 8088 i 8086 poniższy sposób jest o wiele szybszy.

```
mov   dx, 41h      ; Starsze słowo równa się 41h
mov   ax, 1234h    ; Młodsze słowo równa się 1234h
shr   dx, 1        ; Podziel przez dwa (2h)
rcr   ax, 1        ; Rotacja z wykorzystaniem
                   ; znacznika przeniesienia
                   ; ustawionego przez poprzednią
                   ; operację (811Ah)
shr   dx, 1        ; Powtórz powyższe działania
                   ; jeszcze raz by podzielić
```

```
                        ; przez cztery
rcr   ax, 1             ; dx = 1 ax = 408Dh
```

Shift Right Double Precision

88/86	286	386
●	●	●

SHRD

ZASTOSOWANIE

Przesunięcie podwójnej precyzji w prawo.

SKŁADNIA

`SHRD reg, reg, stała`	*Przesunięcie zawartości dwóch rejestrów o podaną liczbę bitów w prawo*
`SHRD mem, reg, stała`	*Przesunięcie zawartości rejestru do danej w pamięci*
`SHRD reg, reg, cl`	*Przesunięcie zawartości dwóch rejestrów o podaną w cl liczbę bitów*
`SHRD mem, reg, cl`	*Przesunięcie zawartości rejestru do danej w pamięci*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli zmienił się znak argumentu
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany na wartość ostatnio wysuniętego bitu
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Instrukcja ta może być pomocna przy operowaniu na dużych liczbach. Ponieważ bity wysuwane z jednego są umieszczane w drugim rejestrze, to instrukcja SHRD jest wygodna do sprawdzania wartości poszczególnych części słowa.

PATRZ

RCL, RCR, ROR, RCR	*Instrukcje przesunięć cyklicznych*
SHL, SAL, SHR, SAR	*Instrukcje przesuwania*
SHLD	*Instrukcja przesuwania w lewo w podwójnej precyzji*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg, stała	00001111 10101100 11,reg,r/m dana	shrd ax,bx,4	-	-	3
mem, reg, stała	00001111 10101100 mod,reg,r/m przemieszczenie dana	shrd value,cx,3	-	-	7
reg, reg, cl	00001111 10101101 11,reg,r/m	shrd ax,bx,cl	-	-	3
mem, reg, cl	00001111 10101101 mod,reg,r/m przemieszczenie	shrd value,ax,cl	-	-	7

PRZYKŁAD

Poniższy przykład wyświetla w odwrotnej kolejności cyfry spakowanej liczby w kodzie BCD z rejestru BX. Instrukcja SHRD służy do pobierania kolejnych cyfr tej liczby.

```
        mov   cl, 4      ; Liczba wyświetlanych cyfr
Ploop:
        xor   dx, dx     ; Wyzeruj DX
        shrd  dx, bx, 4  ; Pobierz do dl cyfrę
        or    dl, 30h    ; Przekształć ją na znak ASCII
        mov   ah, 5      ; Numer funkcji wyświetlającej
                         ; znak
        int   21h
        loop  Ploop
```

Substract with Borrow

88/86	286	386
●	●	●

SBB

ZASTOSOWANIE

Odjęcie dwóch liczb z pożyczką.

SBB

SKŁADNIA

`SBB reg, reg`	*Odjęcie zawartości dwóch rejestrów*
`SBB reg, mem`	*Odjęcie od zawartości rejestru danej z pamięci*
`SBB mem, reg`	*Odjęcie od danej w pamięci zawartości rejestru*
`SBB reg, stała`	*Odjęcie od zawartości rejestru danej natychmiastowej*
`SBB mem, stała`	*Odjęcie od danej w pamięci danej natychmiastowej*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli powstał nadmiar przy odejmowaniu liczb ze znakiem
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy argument docelowy był mniejszy niż źródłowy
Przeniesienie pomocnicze	Ustawiany, gdy pojawiła się pożyczka na pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Rozkaz ten jest używany po instrukcjach odejmowania. Służy on najczęściej do odejmowania dwóch bardziej znaczących słów "długiej" liczby.

PATRZ

ADD, ADC	*Instrukcje dodawania*
SUB	*Instrukcja odejmowania*
NEG	*Uzupełnienie do dwóch danej liczby*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0001100w 11,reg,r/m	sbb bx,cx	3	2	2
mem/reg	0001100w mod,reg,r/m przesunięcie	sbb value,ax	24/16+EA	7	6
reg/mem	0001101w mod,reg,r/m przesunięcie	sbb cx,value	13/9+EA	7	7
reg/stała	100000sw mod,reg,r/m dana	sbb dx,5	4	3	2
mem/stała	100000sw mod,reg,r/m przemieszczenie dana	sbb value,44	23/17+EA	7	7

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
acc, stała	0001110w dana	sbb al,21	4	3	2

PRZYKŁAD

Rozmiar bloku pamięci może być wyznaczony przez odjęcie adresu jego początku od adresu końca. Jeśli oba adresy znajdują się w tym samym segmencie, to wystarczy użycie tylko instrukcji SUB. Natomiast w przeciwnym wypadku należy najpierw wyznaczyć fizyczne 20-bitowe adresy i dopiero teraz dokonać operacji odejmowania. Do poprawnego odjęcia czterech najstarszych bitów potrzebna będzie instrukcja SBB.
Funkcja odejmuje dalekie wskazanie zawarte w **ES:DI** od adresu **DS:SI**.

```
mov   ax, ds          ; Numer segmentu adresu
                      ; początku bloku
sub   bx, bx          ; Wyzeruj BX
shl   ax, 1           ; Aby wyznaczyć adres fizyczny
                      ; należy pomnożyć przez 16
rcl   bx, 1
shl   ax, 1
rcl   bx, 1
shl   ax, 1
rcl   bx, 1
shl   ax, 1
rcl   bx, 1
add   si, ax          ; Zmodyfikuj przesunięcie
adc   bx, 0           ; Dodaj do starszego słowa
mov   ax, es          ; Numer segmentu adresu końca bloku
sub   bx, bx          ; Wyzeruj CX
shl   ax, 1           ; Aby wyznaczyć adres fizyczny
                      ; należy pomnożyć przez 16
rcl   cx, 1
shl   ax, 1
rcl   cx, 1
shl   ax, 1
rcl   cx, 1
shl   ax, 1
rcl   cx, 1
add   di, ax          ; Zmodyfikuj przesunięcie
add   cx, 0           ; Dodaj do starszego słowa
sub   di, si          ; Odjęcie adresu początku od
                      ; adresu końca bloku
sbb   cx, dx          ; Odjęcie czterech najstarszych
                      ; bitów z uwzględnieniem pożyczki
```

By wyznaczyć ilość bajtów pamięci pomiędzy adresami 0452:0D03 i 2510:0003, należy, przed wywołaniem powyższego podprogramu, umieścić je w rejestrach DS:SI i ES:DI.

SCAS, SCASB, SCASW

W pierwszej części funkcji modyfikowany jest adres początku bloku. Po wykonaniu instrukcji SHL i RCL, rejestr SI ma wartość 4520h, a BX wartość 0. Operacja dodawania powoduje wyznaczenie fizycznego adresu początku bloku pamięci. Rejestr SI jest równy 5223h, a BX nadal 0.
Po przeprowadzeniu takiej samej operacji na adresie końcowym otrzymujemy jego adres fizyczny. Rejestr DI zawiera wartość 5103h (młodsze słowo), a CX 0002h (starsze słowo). Odjęcie młodszych słów daje wynik 0FEE0h i powoduje ustawienie znacznika przeniesienia. Rozkaz SBB odejmuje starsze słowa, a wynik (0001h) umieszcza w rejestrze CX. Otrzymaliśmy więc rozmiar bloku pamięci równy 1FEE0h czyli 130,784 bajtów.

Scan a String

88/86	286	386
●	●	●

SCAS, SCASB, SCASW

ZASTOSOWANIE

Porównanie bajtu lub słowa wskazywanego przez ES:DI z daną, która znajduje się w akumulatorze AL lub AX.

SKŁADNIA

```
SCAS    [seg:]argument
SCASB
SCASW
```

ZNACZNIKI

Nadmiar	Ustawiany, jeśli rezultat odejmowania nie mieści się w zakresie wyznaczonym przez argumenty
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy argument docelowy jest mniejszy niż argumentź źródłowy
Przeniesienie pomocnicze	Ustawiany, gdy na 3 lub 7 (argumenty 16-bitowe) pozycji pojawiła się pożyczka

KOMENTARZ

Argument podany instrukcji SCAS używany jest jedynie do wyznaczenia rozmiaru porównywanych danych oraz do opcjonalnej zmiany segmentu przypisanego do niego.
Rozkaz ten może zostać poprzedzony przedrostkiem REPNE w celu odszukania konkretnego bajtu lub słowa w bloku danych.

PATRZ

LODS, LODSB, LODSW	*Pobieranie danych z łańcucha*
MOVS, MOVSB, MOVSW	*Kopiowanie łańcuchów*

CMPS, CMPSB, CMPSW	*Porównywanie łańcuchów*
STOS, STOSB, STOSW	*Zapisywanie danych do łańcucha*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślne	1010111w	scasb	15/19	7	7

PRZYKŁAD

Patrz przykład do instrukcji CMPS.

Substract

88/86	286	386
●	●	●

SUB

ZASTOSOWANIE

Odjęcie dwóch liczb.

SKŁADNIA

`SUB reg, reg`	*Odjęcie zawartości dwóch rejestrów*
`SUB reg, mem`	*Odjęcie od zawartości rejestru danej z pamięci*
`SUB mem, reg`	*Odjęcie od danej w pamięci zawartości rejestru*
`SUB reg, stała`	*Odjęcie od zawartości rejestru danej natychmiastowej*
`SUB mem, stała`	*Odjęcie od danej w pamięci danej natychmiastowej*

ZNACZNIKI

Nadmiar	Ustawiany, jeśli powstał nadmiar przy odejmowaniu liczb ze znakiem
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Ustawiany, gdy argument docelowy był mniejszy niż źródłowy
Przeniesienie pomocnicze	Ustawiany, gdy pojawiła się pożyczka na pozycji 3, lub 7 dla liczb 16-bitowych

KOMENTARZ

Rozkaz ten jest używany do odejmowania danych o rozmiarze bajtu lub słowa. Służy on również do odejmowania mniej znaczących słów długiej liczby.

PATRZ

ADD, ADC	*Instrukcje dodawania*
SBB	*Instrukcja odejmowania z pożyczką*

SUB

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0010101w 11,reg,r/m	sub ax,bx	3	2	2
mem/reg	0010101w mod,reg,r/m przesunięcie	sub value,ax	24/16+EA	7	6
reg/mem	0010100w mod,reg,r/m przesunięcie	sub ax,value	13/9+EA	7	7
reg/stała	100000sw mod,reg,r/m dana	sub bx,5	4	3	2
mem/stała	100000sw mod,reg,r/m przemieszczenie dana	sub value,10	25/17+EA	7	7
acc, stała	0010110w dana	sub al,17	4	3	2

PRZYKŁAD

Do pomnożenia liczby przez 15 przy użyciu instrukcji SHL konieczne są trzy rozkazy SHL i trzy rozkazy ADD:

```
mov   bx, ax        ; BX = Liczba * 1
shl   bx, 1         ; BX = Liczba * 2
add   ax, bx        ; AX = Liczba * 3
shl   bx, 1         ; BX = Liczba * 4
add   ax, bx        ; AX = Liczba * 7
shl   bx, 1         ; BX = Liczba * 8
add   ax, bx        ; AX = Liczba * 15
```

Natomiast, korzystając z instrukcji SUB, można zmniejszyć liczbę koniecznych rozkazów:

```
mov   bx, ax        ; BX = Liczba * 1
shl   ax, 1         ; AX = Liczba * 2
shl   ax, 1         ; AX = Liczba * 4
shl   ax, 1         ; AX = Liczba * 8
shl   ax, 1         ; AX = Liczba * 16
sub   ax, bx        ; AX = Liczba * 15
```

Jeśli AX = 11, wynikiem czwartej instrukcji przesunięcia będzie liczba 176. Odjęcie od niej 11 daje 165, czyli właśnie 15 * 11.

Test bits

88/86	286	386
●	●	●

TEST

ZASTOSOWANIE

Sprawdzenie stanu pewnych bitów argumentu.

SKŁADNIA

`TEST reg, reg`	*Testowanie logiczne zawartości dwóch rejestrów*
`TEST reg, mem`	*Testowanie logiczne zawartości rejestru z daną w pamięci*
`TEST mem, reg`	*Testowanie logiczne danej w pamięci z zawartością rejestru*
`TEST reg, stała`	*Testowanie logiczne zawartości rejestru z daną natychmiastową TEST*
`mem, stała`	*Testowanie logiczne zawartości pamięci z daną natychmiastową*

ZNACZNIKI

Nadmiar	Zerowany
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Zerowany
Przeniesienie pomocnicze	Stan nieokreślony

Otrzymany wynik nie jest nigdzie wpisywany, natomiast według niego ustawiane są znaczniki.

KOMENTARZ

Instrukcja TEST wykonuje logiczną operację AND na swoich argumentach. Wynik nie jest nigdzie zapamiętywany, ale na jego podstawie są ustawiane znaczniki. Rozkaz jest najczęściej używany do testowania wartości określonych bitów.

PATRZ

AND	*Logiczna operacja AND*
OR, XOR, NOT	*Pozostałe instrukcje logiczne*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	1000011w 11,reg,r/m	test bx,ax	3	2	2
mem/reg*	1000011w mod,reg,r/m przemieszczenie	test value,ax	13/9+EA	6	5

TEST

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/stała	1111011w mod,000,r/m dana	test dx,4	5	3	2
mem/stała	1111011w mod,000,r/m przemieszczenie dana	test value,5	11+EA	6	5
acc, stała	1010100w dana	test ax,16	4	3	2

* Nie istnieje oddzielne kodowanie trybu reg,mem. Turbo Assembler automatycznie zamienia tryb reg,mem na mem,reg, co oczywiście daje taki sam rezultat.

PRZYKŁAD

Poniższa funkcja przeprowadza testowanie podanej grupy bitów. Testowana dana znajduje się w rejestrze AX. Rejestr CH zawiera numer pozycji najmniej znaczącego bitu, a rejestr CL liczbę testowanych bitów.

```
mov   bx, 1      ; Rejestr BX będzie zawierał maskę
shl   bx, cl     ; Przesuń maskę o ilość testowanych
                 ; bitów
dec   bx         ; Teraz prawa część maski zawiera
                 ; same jedynki
mov   cl, ch     ; Ustal pozycję testowanych bitów
shl   bx, cl     ; Przesuń maskę na tę pozycję
test  ax, bx     ; Testuj daną
ret              ; Jeśli w polu nie ma żadnego
                 ; ustawionego bitu, powrót
                 ; z ustawionym znacznikiem zera
```

Rozważmy przypadek, gdy rejestr AX = 0110100111110000b, CH = 5 i CL = 6. Pierwsze dwie instrukcje powodują umieszczenie jedynki w rejestrze BX na szóstej pozycji (BX = 0000000001000000b). Po dekrementacji rejestr BX przyjmuje wartość 0000000000111111b. Jest to już prawie gotowa maska. Następna instrukcja przesuwania umieszcza grupę jedynek na właściwym miejscu (0000011111100000b). Teraz rozkaz TEST dokonuje operacji AND na podanej w AX danej oraz tak wyznaczonej masce i otrzymujemy rezultat 0000000111100000b. Jest on ignorowany, ale znacznik zera nie zostanie ustawiony.

Exclusive OR

88/86	286	386
●	●	●

XOR

ZASTOSOWANIE

Różnica symetryczna argumentów.

SKŁADNIA

`XOR reg, reg`	*Różnica symetryczna zawartości dwóch rejestrów*
`XOR reg, mem`	*Różnica symetryczna zawartości rejestru i danej z pamięci*
`XOR mem, reg`	*Różnica symetryczna danej w pamięci i zawartości rejestru*
`XOR reg, stała`	*Różnica symetryczna zawartości rejestru i danej natychmiastowej*
`XOR mem, stała`	*Różnica symetryczna zawartości pamięci i danej natychmiastowej*

ZNACZNIKI

Nadmiar	Zerowany
Znak	Ustawiany zgodnie ze znakiem wyniku
Zero	Ustawiany, jeśli wynikiem operacji jest zero
Parzystość	Ustawiany, gdy w wyniku znajduje się parzysta liczba jedynek
Przeniesienie	Zerowany
Przeniesienie pomocnicze	Stan nieokreślony

KOMENTARZ

Bit wyniku przyjmuje wartość 1, gdy odpowiadające sobie bity w argumentach mają przeciwną wartość. Cecha ta jest wykorzystywana do negowania wartości pewnych bitów. Przeprowadzenie operacji XOR z bitem równym 1, powoduje zmianę stanu odpowiadającego mu bitu w argumencie docelowym.

Jeśli dana jest poddawana tej operacji sama ze sobą, to wynikiem operacji będzie wartość 0. Jest to szybki sposób wyzerowania zawartości rejestru:

```
XOR    ax, ax    ; Ten rozkaz, który zeruje rejestr AX
                 ; zajmuje jeden bajt pamięci i jest
                 ; wykonywany w 3 taktach. Natomiast
                 ; instrukcja przesłania zajęłaby dwa
                 ; bajty, a jej wykonanie 4 takty
```

PATRZ

AND, OR, NOT, TEST	*Pozostałe instrukcje logiczne*

XOR

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg/reg	0011001w 11,reg,r/m	xor ax,ax	3	2	2
mem/reg	0011001w mod,reg,r/m przemieszczenie	xor value,ax	24/16+EA	7	6
reg/mem	0011000w mod,reg,r/m przemieszczenie	xor ax,value	13/9+EA	7	7
reg/stała	100000sw mod,reg,r/m dana	xor bx,10	4	3	2
mem/stała	100000sw mod,001,r/m przemieszczenie dana	xor value,17	25/17+EA	7	7
acc, stała	0011010w dana	xor al,4	4	3	2

PRZYKŁAD

Do indeksowania danych używa się kluczy mieszających. Klucz ten jest przekształcany na liczbę, która jest indeksem w tablicy wskaźników do rekordów. Jednym ze sposobów utworzenia klucza mieszającego jest przeprowadzenie operacji XOR tego klucza z każdym znakiem łańcucha. Zawartość klucza, po pobraniu każdego znaku, jest przesuwana w lewo. Poniższa funkcja tworzy klucz mieszający na podstawie łańcucha, który jest zakończony bajtem zerowym. Jego adres znajduje się w rejestrze BX. Rezultat jest wpisywany do rejestru CX.

```
        xor     cx, cx              ; Wyzeruj rejestr CX
HLoop:
        cmp     byte ptr [bx], 0    ; Czy to koniec łańcucha
        je      done
        xor     cl, byte ptr [bx]   ; Operacja XOR
                                    ; z kolejnym znakiem
                                    ; łańcucha
        shl     cx, 1               ; Przesuń klucz
                                    ; o jeden bit w lewo
        inc     bx                  ; Następny bajt
        jmp     short HLoop
done:
```

Rozdział 6 *Procedury, pętle i skoki.*

Żaden program nie wykonuje się od początku do końca liniowo. W typowym programie wywoływane są przecież procedury, wykonywane są pętle, za pośrednictwem skoków sterowanie przekazywane jest z jednej części programu do innej. Linia, po której porusza się program może być prawie prosta, ale w ekstremalnych wypadkach może tworzyć istny labirynt.
Większość nowoczesnych języków wysokiego poziomu narzuca pewną strukturę programu, lecz język asemblera nie. Nie ma właściwie żadnych ograniczeń na złożoność przepływu sterowania w programie. Zależy ona tylko od doświadczenia i umiejętności programisty. Rozdział ten opisuje rozkazy, które pozwalają na tworzenie tego "bałaganu", lecz zawiera również wiele wskazówek i rad, które pomagają uniknąć chaosu.

Skoki

Instrukcje są zwykle wykonywane w takiej kolejności, w jakiej znajdują się w pamięci. Rozkaz skoku pozwala na przeniesienie sterowania w inne miejsce programu. W procesorach 80x86 adres następnej instrukcji do wykonania przechowuje specjalny rejestr zwany wskaźnikiem rozkazów (IP). Polecenie skoku po prostu powoduje zapisanie do niego nowej wartości.

Instrukcja JMP

Instrukcja JMP powoduje załadowanie rejestru IP nową wartością. Ma ona dwa podstawowe formaty. W pierwszym z nich podaje się po prostu etykietę, pod którą będzie przekazane sterowanie:

```
jmp    etykieta
```

Asembler może ją przetłumaczyć na trzy różne sposoby. Różnice między nimi wynikają z odległości, na jaką skok ma być wykonany. Wybór ten można pozostawić do rozwiązania kompilatorowi lub też samemu poinformować go, który rodzaj skoku ma być użyty.
Gdy adres skoku znajduje się w przedziale od -128 do +127 bajtów od instrukcji JMP, to można użyć skoku krótkiego. Na przykład, by wykonać skok do przodu o 5 bajtów, musi być wygenerowany kod:

```
EDh    05h
```

Drugi bajt jest traktowany jako liczba ze znakiem, która jest dodawana do rejestru IP, a wykonywanie programu wznawiane jest od tak powstałego adresu. By wymusić krótki format skoku, należy użyć słowa kluczowego SHORT:

```
jmp    SHORT etykieta
```

Jeśli się jednak okaże, że etykieta znajduje się zbyt daleko, to podczas kompilacji pojawi się błąd. Należy wtedy usunąć z programu słowo SHORT.

Następnym rodzaj skoków stanowią skoki bliskie (NEAR). Jego działanie jest dokładnie takie samo, jak rozkazu poprzedniego, inny jest tylko zakres. Zwiększa się on do -32,768 a 32,767 bajtów. Skok w tył o 150 bajtów jest przez asembler kodowany następująco:

```
E9h 6Ah FFh
```

Ostatnie dwa bajty (0FF6Ah czyli -150) są daną dodawaną do wskaźnika rozkazów.
Ostatni typ skoku pozwala na przekazanie sterowania w dowolne miejsce pamięci. Jest to skok daleki (FAR). Rozkaz ten wymaga czterech bajtów, które zawierają nowy numer segmentu i przemieszczenie w tym segmencie. W procesorze 80386, w segmentach 32-bitowych, skok ten wymaga podania 6 bajtów. Pierwsze cztery stanowią przemieszczenie w segmencie, którego numer zawarty jest w dwóch ostatnich bajtach. Skok pod adres **0A1E:0100** zostanie zakodowany następująco:

```
EAh 00h 01h 1Eh 0Ah
```

Ostatnie cztery bajty zawierają przemieszczenie oraz numer segmentu skoku.
Na procesorach 8088 i 8086 jedyną różnicą w efektywności pomiędzy tymi instrukcjami jest ilość zajętej pamięci. Natomiast na procesorach 80286 i 80386 wykonanie rozkazu dalekiego skoku wymaga dodatkowych 5 taktów zegara.
Turbo Assembler pozwala również na definiowanie typów etykiet. Mogą być one bliskie (NEAR) lub dalekie (FAR). Odwołanie do etykiety, która ma określony typ, definiuje jednocześnie typ skoku. (Więcej informacji o definiowaniu etykiet zawiera rozdział 9.)

Skoki pośrednie

Powszechną techniką, używaną przez interpretery poleceń, jest wykorzystywanie tablic adresów funkcji. Jedyną rzeczą, którą analizator musi zrobić jest wyznaczenie w tablicy numeru potrzebnej funkcji, a następnie jej wywołanie.
Aby można było wywołać podprogram poprzez tablicę adresów, musi istnieć sposób na wykonanie skoku, którego adres znajduje się w pamięci lub w rejestrze. Są to tzw. skoki pośrednie. Tabela 6-1 zawiera opis trzech możliwych typów skoków pośrednich.

Tabela 6-1. Skoki pośrednie.

Adresowanie	Przykład	Opis
Rejestr	jmp cx	Skok pod adres znajdujący się w rejestrze. W procesorze 80386 można użyć rejestrów 32-bitowych przy skoku w segmencie 32-bitowym.
mem near	jmp WORD PTR	Skok bliski do adresu zawartego w podanym słowie w pamięci. Dozwolone są wszystkie tryby adresowania. Dyrektywa WORD PTR jest dla asemblera poleceniem przetłumaczenia rozkazu jako skok bliski, gdyż nakazuje ona użycie tylko dwóch bajtów.

Adresowanie	Przykład	Opis
mem far	jmp DWORD PTR	Skok daleki do adresu zawartego w podanych dwóch słowach w pamięci. Dyrektywa DWORD PTR jest dla asemblera poleceniem przetłumaczenia rozkazu jako skok daleki, gdyż nakazuje mu ona użycie czterech bajtów.

Skoki warunkowe

W programie zachodzi często potrzeba podjęcia decyzji na podstawie otrzymanych wyników. W procesorach 80x86 służą do tego celu rozkazy skoków warunkowych. To, czy skok zostanie wykonany, zależy od stanu poszczególnych znaczników. Wszystkie 18 warunków jest przedstawionych w tabeli 6-2.

Tabela 6-2. Skoki warunkowe.

Testy po operacjach arytmetycznych na liczbach bez znaku

Nazwa	Opis
JB/JNAE	Jeśli mniejsze/nie większe i nie równe (CF=1)
JAE/JNB	Jeśli większe lub równe/nie mniejsze (CF=0)
JBE/JNA	Jeśli mniejsze lub równe/nie większe (CF=1 ZF=1)
JA/JNBE	Jeśli mniejsze/nie większe i nie równe (CF=0 ZF=0)

Testy po operacjach arytmetycznych na liczbach ze znakiem

Nazwa	Opis
JL/JNGE	Jeśli mniejsze/nie większe i nie równe (SF<>OF)
JGE/JNL	Jeśli większe lub równe/nie mniejsze (SF=OF)
JLE/JNG	Jeśli mniejsze lub równe/ nie większe (ZF=1 lub SF<>OF)
JG/JNLE	Jeśli większe/nie mniejsze i nie równe (ZF=0 lub SF=OF)

Testy równości

Nazwa	Opis
JE/JZ	Jeśli równe/zero (ZF=1)
JNE/JNZ	Jeśli nie równe/nie zero (ZF=0)

Testy znaku

Nazwa	Opis
JS	Jeśli ujemne (SF=1)
JNS	Jeśli dodatnie (SF=0)

Testy przeniesienia

Nazwa	Opis
JC	Jeśli wystąpiło przeniesienie
JNC	Jeśli nie wystąpiło przeniesienie

Testy nadmiaru

Nazwa	Opis
JO	Jeśli wystąpił nadmiar
JNO	Jeśli nie wystąpił nadmiar

Testy parzystości

Nazwa	Opis
JP/JPE	Jeśli parzyste (PF=1)
JNP/JPO	Jeśli nie parzyste (PF=0)

By skorzystać z rozkazu skoku warunkowego, należy najpierw odpowiednio ustawić wartości znaczników. Najczęściej używa się do tego celu instrukcji CMP:

```
cmp     ax, bx          ; Porównaj rejestr AX z rejestrem
                        ; BX (Znaczniki są ustawiane na
                        ; podstawie wyniku odejmowania
                        ; argumentów
je      match           ; Skok będzie wykonany,
                        ; gdy AX = BX (ZF = 1)
```

Innym, często używanym przed instrukcjami skoków warunkowych rozkazem jest TEST (sprawdzenie stanu pewnych grup bitów). Dodatkowo znaczniki są jeszcze ustawiane przez rozkazy arytmetyczne i logiczne. Instrukcje skoków warunkowych mogą być użyte, na przykład do sprawdzenia, czy operacja arytmetyczna spowodowała powstanie nadmiaru:

```
add     ax, cx          ; Dodanie dwóch liczb
jc      unsigned        ; Ustawiony znacznik przeniesienia
                        ; oznacza wystąpienie nadmiaru
                        ; przy operacji na liczbach
                        ; bez znaku
jo      signed          ; Ustawiony znacznik nadmiaru
                        ; oznacza wystąpienie nadmiaru przy
                        ; operacji na liczbach ze znakiem
```

Instrukcja OR ustawia, na podstawie danej zawartej w rejestrze, wartości znaczników zera, znaku i parzystości. Przeprowadzenie operacji OR na danej samej ze sobą ustawi znacznik znaku, gdy jej najstarszy bit równy jest jeden; znacznik zera - gdy dana równa jest zeru i znacznik parzystości - gdy jej reprezentacja dwójkowa zawiera parzystą liczbę jedynek. Następny przykład pokazuje sposób użycia instrukcji OR, do ustawiania znaczników przy odczytywaniu ciągu danych:

```
          .DATA
DataList  DB    "Oto lista bajtów", 0
          .CODE
          mov   bx, OFFSET DataList ; Pobierz adres
                                    ; danych
Dloop:
          mov   al, [bx]            ; Pobierz bajt
          or    al, al              ; Ustaw odpowiednio
                                    ; znaczniki
          jz    Done                ; Koniec po napotkaniu
                                    ; zerowego bajtu
          .                         ; Operacje na danej
          .
          .
          inc   bx                  ; Następny bajt
          jmp   Dloop
Done:
```

Podprogram umieszcza adres danych w rejestrze BX. Następnie pobiera do AL jeden bajt. Ponieważ instrukcja MOV nie wpływa na stan znaczników, to do ich ustawienia należy użyć rozkazu OR. Jeśli odczytany został bajt równy zero, wówczas rozkaz OR ustawi znacznik zera, a instrukcja JZ przekaże sterowanie do etykiety Done. W przeciwnym wypadku na danej zostaną przeprowadzone pewne operacje i pętla zostanie powtórzona.

Instrukcje wysokiego poziomu w języku asemblera

Języki wysokiego poziomu zapewniają kilka instrukcji, które upraszczają przepływ sterowania w programie. Na przykład, pętla while jest wykonywana aż do spełnienia określonego warunku, a decyzje są podejmowane za pomocą konstrukcji if,then,else. Procesory 80x86 nie mają bezpośrednio zaimplementowanych tego typu operacji. Musi to zrobić programista za pomocą instrukcji skoków.

Pętla WHILE

W pętli while najpierw jest obliczana wartość wyrażenia logicznego. Jeśli jest ono prawdziwe, to wykonywane jest ciało pętli. Skonstruowanie pętli, która sprawdza prosty warunek nie jest trudne. Poniższa pętla wykonuje się tak długo, jak długo rejestr AL jest różny od zera.

```
Loop1:
        or    al, al      ; Ustaw znaczniki
        jz    Done
                          ; w ciele pętli wartość AL musi
                          ; być ustawiona na 0, inaczej pętla
                          ; nigdy się nie zakończy

        jmp   Loop1
Done:
```

Jednak nie wszystkie pętle są tak proste. Wyrażenia logiczne mogą się przecież składać z wielu składników połączonych spójnikami AND lub OR. Poniższa pętla wykonuje się, gdy rejestr AL jest różny od 0 i rejestr CX jest mniejszy od 10 lub gdy zawartości rejestrów BX i DX są równe.

```
Loop2:
        or    al, al    ; Ustaw znaczniki
        jz    Done      ; Koniec, gdy AL = 0
        cmp   cx, 10    ; w przeciwnym razie sprawdź
                        ; czy CX < 10
        jb    Cxok
        cmp   bx, dx    ; lub bx = dx
        jne   Done
Cxok:
        .
        .
        .
        jmp   Loop2
Done:
```

Przy obliczeniu wartości tego wyrażenia uwzględniamy trzy możliwe wyniki. Gdy pierwszy warunek nie będzie spełniony, to, niezależnie od wartości pozostałych (wyrażenia są połączone spójnikiem AND), pętlę należy zakończyć. Gdy natomiast drugi warunek okaże się prawdziwy, wówczas, niezależnie od wartości ostatniego (wyrażenia są połączone spójnikiem OR), wykonywanie pętli należy kontynuować. Jeśli żaden z powyższych rezultatów nie został otrzymany, to pozostaje obliczenie wartości ostatniego warunku. Dopiero jego wynik decyduje o prawdziwości całego wyrażenia.
Rzecz jeszcze bardziej się komplikuje, gdy pętla jest długa i etykieta końca pętli leży poza zasięgiem krótkich skoków warunkowych. Jednym ze sposobów na pokonanie tej trudności jest zastosowanie skoku z przeciwnym warunkiem do ominięcia instrukcji **JMP**, która przekazuje sterowanie na koniec pętli.

```
Loop2:
        or    al, al
        jnz   Skip1       ; Tak naprawdę chcemy skoczyć, gdy
                          ; wynik jest równy zero,
        jmp   Done        ; ale etykieta Done jest zbyt
                          ; daleko
Skip1:
        .
        .
        .
        jmp   Loop2
Done:
```

Wszystko jest w porządku, gdy sprawdzany jest tylko jeden warunek. Natomiast, gdy kilka warunków wymaga skoków długich, sposób ten staje się złożony i niezbyt efektywny. Można wtedy przenieść sprawdzanie wszystkich warunków na koniec pętli. Ponieważ na

jej końcu umieszczony jest już rozkaz skoku, to skoki warunkowe mogą przekazywać sterowanie bezpośrednio do niego.
Poprzedni przykład ze złożonym warunkiem, zakodowany tym sposobem, będzie wyglądał następująco:

```
        jmp     Lptest          ; Skok do sprawdzania
                                ; warunków
Loop3:
        .
        .
        .
Lptest:
        or      al, al
        jz      Done            ; Koniec, gdy AL = 0
        cmp     cx, 10
        jb      Cxok            ; Gdy CX < 10, kontynuuj pętlę
        cmp     bx, dx
        jne     Done            ; Koniec gdy bx <> dx
Cxok:
        jmp     Loop3
Done:
```

W podobny sposób można zrealizować pętle **do,until**, pętlę **for** czy instrukcję **break**.

IF,THEN,ELSE

Do podejmowania decyzji w językach wysokiego poziomu służy instrukcja **if,then,else**. Jeśli wyrażenie logiczne jest prawdziwe, to wykonywany jest blok, który występuje po słowie **then**; w przeciwnym wypadku - blok za słowem **else**. Oto prosty przykład, w którym sprawdzamy, czy rejestr AL jest równy 0:

```
        or      al, al
        jnz     CElse           ; Jeśli nie jest zerem wykonaj
                                ; instrukcje po else
        .                       ; w przeciwnym razie instrukcje
                                ; po then
        .
        .
        jmp     CEndif          ; Omiń instrukcje, które dotyczą
                                ; else
CElse:
        .
        .
        .
CEndif:
```

Podobnie jak w pętli **while** wyrażenie w instrukcji **if,then,else** może być złożone. W poniższym przykładzie wykorzystano takie samo wyrażenie jak w poprzednich przykładach:

```
        or   al, al
        jnz  CElse          ; Jeśli al <> 0 wykonaj
                            ; instrukcje po else
        cmp  cx, 10
        jb   CThen          ; Gdy CX < 10, wykonaj
                            ; instrukcje po then
        cmp   bx, dx
        jne   CElse
CThen:
         .
         .
         .
         jmp   CEndif
CElse:
         .
         .
         .
CEndif:
```

ITEROWANIE

Czasami jest znana liczba obiegów pętli. Może ona, na przykład, odczytywać daną liczbę rekordów z pliku lub pobierać dane z bufora o znanej długości. Pętle te, przy sprawdzaniu warunku zakończenia, używają specjalnego licznika. Za każdym obiegiem jego wartość zostaje zmniejszona o jeden, i gdy jest różna od zera pętla jest kontynuowana.
Instrukcja LOOP przy wyznaczaniu liczby obiegów korzysta z rejestru CX. Jej format jest następujący:

```
loop   etykieta
```

Rozkaz LOOP zmniejsza o jeden rejestr CX. Następnie, gdy jest on różny od zera, sterowanie jest przekazywane pod adres wyznaczony przez etykietę. Gdy natomiast rejestr CX jest równy zeru, to wykonywana jest kolejna instrukcja za rozkazem LOOP. Instrukcja ta jest więc równoważna poniższemu fragmentowi kodu:

```
        dec   cx            ; Dekrementacja rejestru CX
        jnz   etykieta      ; Skok, gdy CX <> 0
```

W poniższym przykładzie pętla będzie wykonana 12 razy:

```
        mov   cx, 12        ; Liczba iteracji
                            ; w rejestrze CX
Sloop:
        .                   ; Ciało pętli
        .
        .
        loop  Sloop         ; Dekrementacja CX; skok, gdy
                            ; jest on różny od 0
        .                   ; Dalsza część programu,
```

```
                                ; gdy CX = 0
        .
        .
```

Istnieją odmiany instrukcji LOOP, które badają oprócz rejestru CX, także stan znacznika zera. Są to rozkazy: LOOPZ i LOOPNZ. Skok w instrukcji LOOPZ wykona się, gdy rejestr CX ma wartość różną od zera, a znacznik zera jest ustawiony; a w instrukcji LOOPNZ, gdy rejestr ma wartość różną od zera, a znacznik zera jest wyzerowany.
Rozkazy te mogą zostać użyte do przeglądania zawartości bufora, aż do spełnienia pewnego warunku lub aż do napotkania końca bufora. W tym celu należy najpierw załadować długość bufora do rejestru CX. Następnie, ustawiając lub zerując znacznik zera (zależnie od zastosowanej instrukcji LOOP), można wykonywanie pętli zakończyć.
W poniższym przykładzie, używając instrukcji LOOPNZ, szukamy w buforze o długości 128 bajtów liczby 0.

```
        mov     cx, 128             ; Długość bufora
        mov     bx, OFFSET Buffer-1
Bloop:
        inc     bx
        cmp     BYTE PTR [bx], 0    ; Szukaj 0
        loopnz Bloop
```

Pętla może zostać zakończona z dwóch powodów: został osiągnięty koniec bufora lub został znaleziony zerowy bajt. Istnieją dwa sposoby na ich rozróżnienie. Jednym z nich jest użycie instrukcji **JZ** lub **JNZ** do zbadania stanu znacznika zera. Drugi natomiast opiera się na wykorzystaniu rozkazu **JCXZ**. Jest to instrukcja krótkiego skoku, który jest wykonywany, gdy rejestr CX zawiera wartość 0. Jest ona więc podobna do instrukcji LOOP, z tym że nie wykonuje zmniejszania zawartości rejestru CX.
Dla procesora 80386 istnieje ponadto instrukcja **JECXZ**, która testuje rejestr ECX.

Podprogramy

Jednym z powodów stosowania podprogramów jest oszczędność pamięci. Ten sam fragment kodu może być wielokrotnie wywoływany z różnych części programu. Również dzięki stosowaniu procedur program staje się bardziej przejrzysty i zrozumiały.
Niektóre procedury mogą być także wykorzystywane w wielu różnych programach. Pozwala to na skrócenie czasu ich powstawania. Zamiast pisania i testowania nowego kodu dla każdego programu, procedura, napisana dla jednego z nich, może być wykorzystana w wielu innych.

Wywołanie i powrót z podprogramu

Do wywołania podprogramu służy instrukcja CALL o formacie:

```
call    podprogram
```

W rozkazie tym podprogram jest etykietą wywoływanej procedury. Podobnie jak w instrukcji JMP etykiety mogą być typu NEAR lub FAR. Różnica między tymi rozkazami polega na tym, że instrukcja CALL, przed wykonaniem skoku, odkłada na stos adres następnego

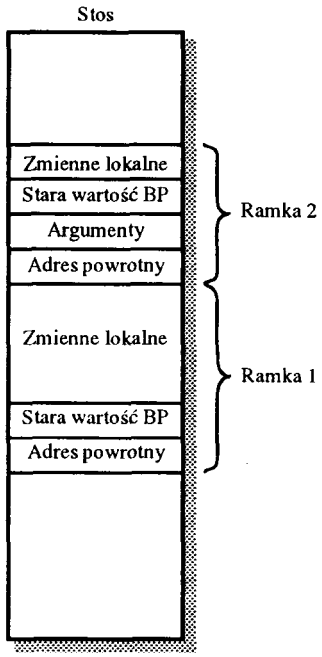
rozkazu. Gdy typem etykiety jest FAR, to na stos przesyłany jest zarówno jej numer segmentu, jak i przemieszczenie. Dla etykiet typu NEAR odkładane jest tylko przemieszczenie.
Po zakończeniu podprogramu odłożony adres jest wykorzystywany do wykonania operacji powrotu do miejsca wywołania procedury. Czynność tę wykonuje instrukcja RET. Musi ona wiedzieć, czy na stosie zapamiętany jest wskaźnik daleki czy bliski. Istnieją dwa sposoby na poinformowanie asemblera o tym, który rozkaz powrotu ma być użyty. Pierwszym jest bezpośrednie użycie odpowiedniej instrukcji RETN lub RETF. Pierwsza powoduje zdjęcie ze stosu adresu bliskiego, a druga dalekiego. Innym sposobem jest użycie dyrektywy PROC, która pozwala określić typ procedury (Rozdział 10 zawiera dokładny opis tej dyrektywy). Poniższy fragment kodu zawiera przykładowe wywołanie i powrót z podprogramu.

```
            .
            .
            .
            call    AFunction
            .
            .
            .
AFunction   PROC    FAR         ; Jest to funkcja daleka
            .
            .
            .
            ret                 ; Powrót do miejsca
                                ; wywołania
AFunction   ENDP                ; Dyrektywa oznaczająca
                                ; koniec funkcji
```

Ramki stosu

Parametry mogą sterować zachowaniem się podprogramu. Podprogram może zaś, zwracając pewne wartości, informować o wynikach swojej działalności. Podczas działania może on również potrzebować miejsca na przechowanie swoich zmiennych tymczasowych.
Program wywołujący może przekazać parametry do podprogramu w rejestrach. Także w rejestrach może być zwrócony wynik. Miejsce na dane tymczasowe może być natomiast przydzielone jeszcze przed wywołaniem procedury.
Takie rozwiązanie ma jednak kilka niedogodności. Co zrobić, gdy np. podprogram wymaga podania większej ilości parametrów niż liczba dostępnych rejestrów? Jak rozwiązać problem rekurencji? Wymaga ona przecież, dla każdego wywołania, oddzielnego pakietu zmiennych lokalnych. Natomiast wspólny obszar byłby zamazywany przez każdorazowe wywołanie.
Dobrym sposobem na rozwiązanie tego problemu jest wykorzystanie stosu. Gdy potrzebne jest odłożenie nań wielu danych, to po prostu definiujemy większy stos. Natomiast zmienne lokalne możemy na nim tworzyć za każdym wywołaniem podprogramu. Zapobiega to oddziaływaniu na siebie poszczególnych procedur. Po zakończeniu podprogramu stos jest czyszczony i dzięki temu zwalnia się miejsce dla kolejnych wywołań.

Parametry podprogramu, adres powrotu oraz zmienne lokalne tworzą ramkę stosu. Rysunek 6-1 przedstawia stos z umieszczonymi na nim kilkoma ramkami.



Rysunek 6-1. ***Ramki stosu.***

Następny przykład przedstawia sposób przekazania do podprogramu parametrów przez stos. Następnie podprogram ten rezerwuje sobie miejsce na zmienne lokalne i tak ustawia wartość rejestru BP, by wskazywał on na obszar parametrów i danych tymczasowych. Po zakończeniu zwalnia on zarezerwowane miejsce i przekazuje sterowanie do programu wywołującego. Na koniec program przykładowy, dokonując odpowiedniej modyfikacji wskaźnika stosu, usuwa ze stosu przekazane parametry.

```
        push  bx          ; Drugi parametr
        mov   ax, 10
        push  ax          ; Pierwszy parametr
        call  MyFunc      ; Wywołanie funkcji
        add   sp, 4       ; Uaktualnienie stosu
MyFunc  PROC  NEAR
        push  bp          ; Zachowaj starą wartość BP
        mov   bp, sp      ; Parametry znajdują się
                          ; o 4 bajty dalej (2 bajty
                          ; na adres powrotu i 2 na
                          ; zapamiętaną wartość BP)
        sub   sp, 10      ; Zarezerwuj 10 bajtów na
```

```
                                ; zmienne lokalne
        mov   di, [bp+6]        ; Pobierz pierwszy parametr
        mov   cx, [bp+4]        ; Pobierz drugi parametr
        mov   [bp-10], cx       ; Zapisz oba w obszarze
                                ; danych lokalnych
        mov   [bp-8], di
        .
        .
        .
        mov   sp, bp            ; Usunięcie danych lokalnych
                                ; ze stosu
        pop   bp                ; Przywrócenie starej
                                ; wartości BP
        ret                     ; Powrót z podprogramu
MyFunc  ENDP
```

Czasami wygodnie byłoby, gdyby to wywoływany podprogram usuwał wszystkie przekazane mu dane ze stosu. Problem tkwi w tym, że przed parametrami znajduje się jeszcze adres powrotu. Na szczęście procesor 80x86 pozwala na jego obejście, udostępniając specjalną instrukcję RET. Jeśli bowiem wystąpi za nią pewna liczba, to jest ona, po pobraniu adresu powrotu, dodawana do wskaźnika stosu. Jej działanie ilustruje przykład:

```
        .
        .
        .
        mov   ax, 50
        push  ax                ; Pierwszy parametr
        mov   ax, 21
        push  ax                ; Drugi parametr
        call  Func2             ; Wywołanie funkcji
        .
        .
        .
Func2   PROC  NEAR
        push  bp                ; Ustaw odpowiednio
                                ; ramkę stosu
        mov   bp, sp
        sub   sp, 4
        mov   ax, [bp+4]        ; Pobierz drugi parametr (21)
        add   ax, [bp+6]        ; Pierwszy parametr (50)
        .
        .
        .
        mov   sp, bp
        pop   bp
        ret   4                 ; Powrót z dodaniem
                                ; wartości 4 do SP
```

Zauważ, że tym razem kolejność przesyłanych parametrów została odwrócona. Jako pierwszy został odłożony na stos pierwszy parametr funkcji. Sposób ten oraz to, że podprogram usuwa przekazane mu parametry, stosowany jest w języku Pascal (Pascal calling convention). Natomiast metoda zastosowana w pierwszym przykładzie wykorzystana została w języku C (C calling convention).
Procesory 80286 oraz 80386 mają dwie dodatkowe instrukcje, które operują na ramkach stosu. Rozkaz ENTER odkłada na stos zawartość rejestru BP oraz przesyła do niego wartość wskaźnika stosu. Następnie rezerwuje on na stosie miejsce na dane lokalne. Powoduje to więc utworzenie takiej samej ramki stosu jak w poprzednich przykładach. Format tej instrukcji jest następujący:

```
ENTER   dane lokalne, poziom zagnieżdżenia
```

Pierwszy argument oznacza rozmiar obszaru przeznaczonego na dane lokalne. Drugi zaś definiuje poziom zagnieżdżenia podprogramu i pozwala na odwołanie się do zmiennych lokalnych podprogramów wywołujących. Ustawienie tego poziomu na 0 powoduje wygenerowanie takiej samej ramki stosu jak w poprzednich przykładach.
Instrukcja LEAVE działa odwrotnie. Ustawia ona wartość wskaźnika stosu na wartość BP, a następnie pobiera ze stosu starą wartość BP. Jest ona bezargumentowa.

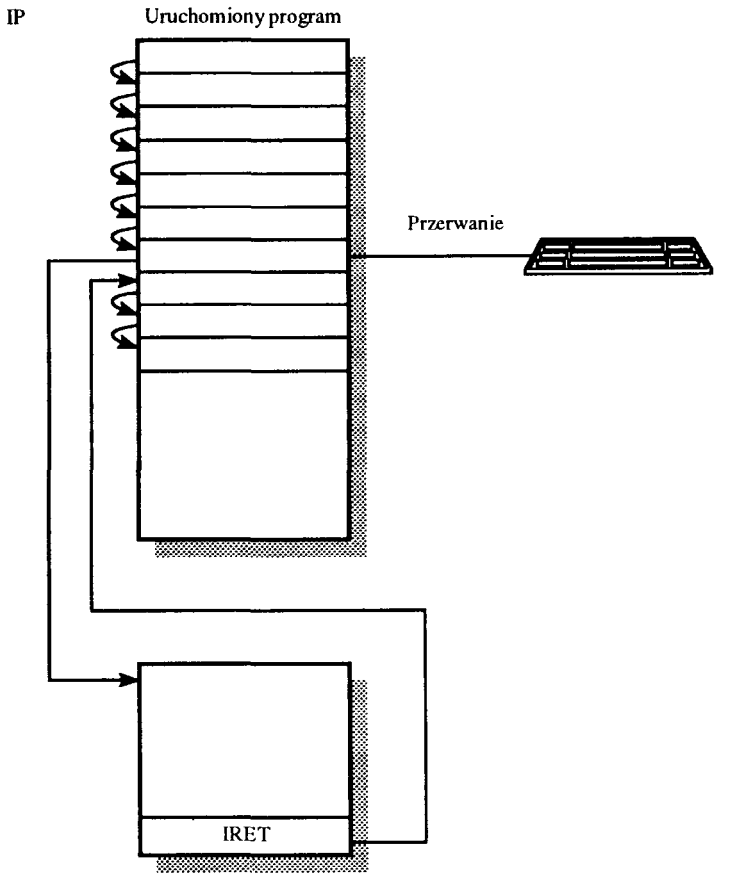
Obsługa przerwań

Pewne zdarzenia zewnętrzne wymagają natychmiastowej obsługi przez procesor. O ich zaistnieniu urządzenia powiadamiają jednostkę centralną wysyłając odpowiedni sygnał. Razem z nim wysyłany jest również numer przerwania. Procesor kończy wtedy właśnie wykonywany rozkaz i, na podstawie tego numeru, ustala adres procedury obsługi danego przerwania (interrupt handler). Następnie przesyła na stos rejestry znaczników, CS oraz IP i wykonuje skok do procedury obsługi. Po jej zakończeniu, za pomocą instrukcji IRET, odtwarzane są znaczniki, a sterowanie wraca do przerwanego programu. Proces ten przedstawia rysunek 6-2.

PRZERWANIA PROGRAMOWE

Program może symulować pojawienie się przerwania za pomocą instrukcji INT. Jej argumentem jest liczba, określająca, które z przerwań należy obsłużyć. Ten typ przerwań jest używany przez programy, które muszą być dostępne z innych programów. Numer przerwania może bowiem pozostać niezmieniony, mimo że procedura obsługująca go może znajdować się pod różnymi adresami. Dobrym przykładem w ten sposób wywoływanego programu jest MS-DOS.
MS-DOS udostępnia wiele funkcji, które mogą być wykorzystane w programach użytkowych. By z nich skorzystać należy w rejestrze AH umieścić numer żądanej funkcji, a następnie wywołać przerwanie **21h**. Na przykład, by wyświetlić na ekranie pojedynczy znak, należy jego kod przesłać do rejestru DL i wywołać funkcję o numerze 2:

```
mov   dl, 'A'        ; Umieść znak w rejestrze DL
mov   ah, 2          ; Numer funkcji w AL
int   21h            ; Wywołaj MS-DOS
```



Rysunek 6-2. *Przerwanie sprzętowe.*

PRZERWANIA SPECJALNE

Przerwanie 3 jest używane w szczególnych wypadkach. Programy uruchomieniowe używają go do ustawiania punktów kontrolnych. Kod instrukcji **INT 3** (0CCh) jest tylko jednobajtowy, może więc zastępować dowolną instrukcję uruchamianego programu. Wykonanie rozkazu **INT 3** powoduje wywołanie programu uruchomieniowego, który odtwarza zamienioną instrukcję oraz pozwala użytkownikowi na zbadanie zawartości interesujących go rejestrów i zmiennych.

Inne specjalne przerwanie jest wywoływane przez rozkaz INTO. Sprawdza on stan znacznika nadmiaru, a jeśli jest on ustawiony, to generuje przerwanie programowe o numerze 4. Instrukcja ta może być używana do testowania rezultatu operacji arytmetycznych. Należy podkreślić, że wywołanie przerwania jest bardzo kosztowne. Na przykład w wypadku instrukcji INTO procesor będzie potrzebował o wiele mniej taktów na wykonanie kodu:

```
        jno     Skip
        call    Overrtn
Skip:
```

Procedura **Overrtn** będzie wykonana wtedy, gdy znacznik nadmiaru będzie miał wartość 1. Dla procesora 8086 kod ten pozwala zaoszczędzić 9 taktów zegara. Dodatkowo wykonanie instrukcji powrotu z przerwania zajmuje o 6 taktów więcej. Ogólnie więc zaoszczędziliśmy 15 taktów zegara. W wypadku, gdy czas wykonania programu nie jest kryterium najważniejszym, można użyć instrukcji INTO. Zajmuje ona tylko jeden bajt, podczas gdy powyższy kod wymaga aż siedmiu bajtów pamięci.

WEKTORY PRZERWAŃ

Tablica 256 wektorów przerwań znajduje się w pamięci operacyjnej pod adresem **0:0**. Każdy z jej czterobajtowych elementów składa się z dwubajtowego przesunięcia oraz dwubajtowego numeru segmentu, w którym znajduje się procedura obsługi. Przed jakąkolwiek operacją, związaną z tablicą przerwań, przerwania należy wyłączyć. Służy do tego instrukcja CLI. Po odczytaniu lub zapisaniu elementu do tabeli przerwania powinny zostać włączone rozkazem STI. Powodem takiej ostrożności jest to, że w czasie, gdy została zmieniona tylko jedna część adresu (przesunięcie lub numer segmentu), może pojawić się przerwanie. Jest rzeczą nieprawdopodobną, by tak zmieniony adres wskazywał na sensowną procedurę. Poniższy przykład ustawia procedurę Func do obsługi przerwania 0Bh.

```
cli                     ; Wyłączenie przerwań
mov   ax, OFFSET Func   ; Pobierz adres funkcji
mov   bx, SEG Func
sub   cx, cx            ; Wyzeruj rejestr CX
mov   es, cx            ; Ustaw numer segmentu na zero
mov   es:[1Ch], ax      ; 1Ch = 0Bh * 4
mov   es:[1Eh], bx
sti                     ; Włącz przerwania
```

Po zakończeniu programu MS-DOS zwalnia przydzieloną mu wcześniej pamięć. Może ona zostać wykorzystana do innych celów. Gdy zakończony program zmienił pewien wektor przerwań tak, by wskazywał na ten obszar, to pojawienie się przerwania spowoduje zawieszenie się systemu.

By zapobiec tego typu wypadkom, należy zawsze pamiętać o odtwarzaniu, przed zakończeniem programu, starej zawartości tablicy przerwań. MS-DOS udostępnia dwie funkcje, które operują na tablicy przerwań. Jest to funkcja o numerze 35h, która pobiera do rejestrów ES i BX podany element tej tablicy oraz funkcja 25h, która zapisuje do niej podany adres. Następny przykład używa tych funkcji do pobrania adresu procedury obsługi przerwania o numerze 0Bh i zastępuje go innym adresem. Po zakończeniu działalności przywracany jest adres dotychczasowej procedury obsługi.

```
; Program korzystający z przerwania 0Bh
          .DATA
Old       DD    0             ; Miejsce na stary wektor
          .CODE
Start:
      mov   ax, 350bh         ; Odczyt wektora o numerze 0Bh
      int   21h
      mov   WORD PTR Old, bx  ; Zachowaj przesunięcie
```

```
                                ; dotychczasowej procedury
                                ; obsługi
      mov   WORD PTR Old+2, bx  ; Zachowaj numer segmentu
                                ; dotychczasowej
                                ; procedury obsługi
      mov   bx, OFFSET Func     ; Pobierz adres nowej funkcji
      mov   ax, SEG Func
      mov   es, ax
      mov   ax, 250bh           ; Ustaw nowy wektor
                                ; przerwania 0Bh
      int   21h
      .                         ; Dalsza część programu
      .
      .
; Przed zakończeniem musi zostać przywrócony stary
; wektor przerwania
      les   bx, Old             ; Pobranie starego wektora
      mov   ax, 250bh           ; Ustawienie wektora
                                ; przerwania 0Bh
      int   21h
; Funkcja, która będzie wywołana po pojawieniu się
; przerwania 0Bh
Func        PROC   FAR
            .
            .
            .
            iret
Func        ENDP
            .STACK
END         Start
```

Podsumowanie

W czasie swojego wykonania program wykonuje wiele skoków i powrotów. Należy jednak uważać, by ścieżka, po której się on porusza, nie stała się zbyt złożona. Można bowiem wtedy bardzo łatwo utracić kontrolę nad tym, co się w programie dzieje. Tak więc instrukcje, które sterują przepływem sterowania to rodzaj miecza obosiecznego. Zapewniają one wprawdzie możliwość przekazania sterowania w dowolne miejsce pamięci, jednakże dowolność ta może doprowadzić do takiej złożoności programu, że niemal niemożliwością może być znalezienie w nim błędu lub zrozumienie jego algorytmu.

Przy rozsądnym planowaniu programu należy używać pętli **while**, instrukcji **if,then,else** oraz podprogramów. Dzięki ich zastosowaniu, a unikaniu karkołomnych skoków, programy stają się czytelne i zrozumiałe, nawet po długiej przerwie.

Check Index Against Array Bounds 88/86 286 386
● ●

BOUND

ZASTOSOWANIE

Porównanie indeksu, który znajduje się w argumencie docelowym, z liczbami wskazywanymi przez argument źródłowy. Jeśli znajduje się on poza wyznaczonym przez nie zakresem, to generowane jest przerwanie o numerze 5.

SKŁADNIA

```
BOUND  rejestr, mem32
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Wartość, która znajduje się w rejestrze, musi być większa od młodszego i mniejsza od starszego słowa argumentu źródłowego. Jeśli ten warunek nie jest spełniony, to generowane jest przerwanie numer 5.

W komputerach, zgodnych z IBM PC, przerwanie numer 5 jest używane do drukowania zawartości ekranu. Jeśli więc ma być wykorzystywana instrukcja BOUND, to należy najpierw przejąć obsługę tego przerwania. Oznaczać to będzie utratę możliwości wydrukowania ekranu.

PATRZ

INT	*Instrukcja przerwania programowego*
INTO	*Przerwanie, gdy wystąpił nadmiar*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, mem32	01100010 mod,reg,r/m przemieszczenie	Bound si, limits	-	23/13	37/10

PRZYKŁAD

W przykładzie najpierw przechwytywana jest obsługa przerwania numer 5. Następnie sprawdzane są zakresy tablicy, po czym przywracana jest dotychczasowa procedura obsługi przerwania numer 5.

```
            .MODEL SMALL
            .DATA
OldInt5     DD   ?                    ; Miejsce na adres
                                      ; starej procedury
                                      ; obsługi
```

BOUND

```
Array       DB   150 dup(0)              ; Tablica o długości
                                         ; 150 bajtów
Limits      DD   Array + (Array+150) * 65536
Inflag      DB   ?                       ; 0 = w zakresie,
                                         ; 1 = poza zakresem
            .CODE
; Porównaj zawartość rejestru SI z zakresem, wyznaczonym
; przez zmienną Limits. Zmienna inflag jest ustawiana
; na podstawie wyniku odpowiednich testów
ChckArray   PROC
            mov   ax, 3505h              ; Funkcja systemowa,
                                         ; która pobiera
                                         ; wektor przerwań
            int   21h
            mov   WORD PTR OldInt5, bx   ; Zachowaj stary
                                         ; wektor
            mov   WORD PTR OldInt5+2, es
            mov   ax, 2505h              ; Funkcja systemowa,
                                         ; która ustawia nową
                                         ; procedurę obsługi
            push  ds                     ; Zachowaj
                                         ; rejestr DS
            push  cs                     ; Skopiuj CS do DS
            pop   ds
            mov   dx, OFFSET OutBounds
            int   21h
            pop   ds                     ; Odtwórz rejestr DS
            mov   Inflag, 0              ; Wyzeruj zmienną
                                         ; Inflag
            bound si, Limits             ; Sprawdź zakres
            mov   ax, 2505h              ; Odtwórz stary
                                         ; wektor
            push  ds
            lds   dx, OldInt5            ; Pobierz adres
                                         ; starej obsługi
                                         ; przerwania
            int   21h
            pop   ds
            ret
ChckArray   ENDP
; Procedura obsługi przerwania 5
OutBounds   PROC  FAR
            mov   Inflag, 1
            iret
OutBounds   ENDP
```

Call a Procedure

88/86	286	386
●	●	●

CALL

ZASTOSOWANIE

Odłożenie na stos adresu następnej instrukcji, a następnie przekazanie sterowania do podprogramu.

SKŁADNIA

```
CALL etykieta
CALL FAR PTR etykieta
CALL rejestr lub miejsce pamięci
CALL WORD PTR miejsce pamięci
CALL DWORD PTR miejsce pamięci
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja CALL przekazuje sterowanie do fragmentu kodu zwanego podprogramem. Po jego zakończeniu sterowanie jest przekazywane do miejsca wywołania. Ponieważ podprogram może być wywoływany z różnych miejsc programu, dlatego musi być do niego przekazany adres powrotu. Instrukcja CALL umieszcza ten adres na stosie.

Ponieważ rozmiar pojedynczego segmentu pamięci wynosi tylko 65536 bajtów, to istnieją specjalne odmiany rozkazu CALL dla wywołań funkcji bliskich oraz takich, które znajdują się poza bieżącym segmentem kodu. Jeśli argumentem wywołania jest etykieta, to do ustalenia rodzaju wywołania wykorzystywana jest jej definicja. Jeśli natomiast adres podprogramu znajduje się w rejestrze lub w komórce pamięci, wówczas o rodzaju skoku decyduje rozmiar rejestru lub miejsca pamięci. Gdy rozmiar ten lub definicja etykiety są błędne lub niejednoznaczne, wtedy do ustalenia rodzaju skoku należy użyć modyfikatorów NEAR, FAR, WORD PTR lub DWORD PTR.

PATRZ

NEAR, FAR, PTR, WORD, DWORD	*Używane do ustalenia rodzaju skoku*
JB lub JNAE, JAE lub JNB, JBE lub JNA, JA lub JNBE, JL lub JNGE, JGE lub JNL, JLE lub JNG, JG lub JNLE, JE lub JZ, JNE lub JNZ, JS, JNS, JC, JNC, JO, JNO, JP lub JPE, JNP lub JPO,	*Skoki warunkowe*
JMP	*Skok bezwarunkowy*
RET	*Instrukcja powrotu z podprogramu*

CALL

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
NEAR etykieta	11101000 przesunięcie (2 bajty)	call func	23/19	7	7
FAR etykieta	10011010 przesunięcie (4 bajty)	call farfunc	36/28	13	17
rejestr	11111111 11,010,r/m	call si	20/16	7	7
word pamięć	11111111 mod,010,r/m	call [bx]	29/21+EA	10	10
dword pamięć	11111111 mod,011,r/m	call dword ptr [bx]	53/37+EA	16	22

PRZYKŁAD

Zastosowanie procedur sprawia, że często powtarzające się czynności może wykonywać ten sam fragment programu. Pozwala to na znaczne zmniejszenie rozmiarów programu, a także upraszcza to proces testowania. Poniższy przykład zawiera wywołanie podprogramu, który wyświetla w podanym miejscu ekranu łańcuch znaków. Pozycja jest przekazywana za pośrednictwem rejestru DX (DH - numer wiersza, DL - numer kolumny). Adres łańcucha znajduje się w parze rejestrów DS:BX.

```
; Poniższy kod przygotowuje wartości parametrów
; wywołania. Zmienne Curcol oraz Str powinny zostać
; wcześniej zadeklarowane.
          .
          .
          .
          mov   dh, 24          ; Ostatni wiersz na
                                ; ekranie
          mov   dl, Curcol      ; Numer kolumny znajduje
                                ; się w zmiennej Curcol
          mov   bx, OFFSET Str  ; Pobierz adres łańcucha
          call  PrintAt
          .
          .
          .
; To jest właśnie podprogram. Do zmiany pozycji kursora użyta została procedura BIOS-u, a do wyświetlenia łańcucha znaków, funkcja DOS-a. Następnie sterowanie przekazywane jest ponownie do programu, który go wywołał.
PrintAt   PROC  NEAR
          push  bx              ; Zachowaj na stosie
```

```
                                ; adres łańcucha
        mov     ah, 4           ; Numer funkcji BIOS-u,
                                ; która zmienia pozycję
                                ; kursora
        mov     bh, 0           ; Numer strony ekranu
        int     10h             ; Wywołanie BIOS-u
        pop     dx              ; Pobranie adresu łańcucha
        mov     ah, 9           ; Numer funkcji, która
                                ; wyświetla łańcuch znaków
        int     21h             ; Wywołanie DOS-a
        ret                     ; Powrót z podprogramu
PrintAt ENDP
```

Make Stack Frame — 88/86 286 386

286 ● 386 ●

ENTER

ZASTOSOWANIE

Odłożenie na stos zawartości rejestru BP i wpisanie do niego bieżącej wartości wskaźnika stosu. Następnie, poprzez odpowiednią modyfikację wskaźnika stosu, zostaje na stosie zarezerwowane miejsce na zmienne lokalne.

SKŁADNIA

```
ENTER  bajty_zm_lok, poziom_zagłębienia
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Dzięki utworzeniu ramki stosu można w wygodny sposób przesłać do podprogramu parametry oraz zarezerwować miejsce na zmienne lokalne. Cała pamięć, przydzielona na przekazanie parametrów oraz stworzenie zmiennych lokalnych, po zakończeniu procedury zostaje zwolniona. Wykorzystanie rejestru BP pozwala w prosty sposób na odwoływanie się do tej pamięci.

W procesorach 8088 i 8086 ten sam efekt daje poniższa sekwencja instrukcji:

```
push  bp
mov   bp, sp
sub   sp, bajty_zm_lok
```

PATRZ

CALL — *Instrukcja wywołania podprogramu*

ENTER

LEAVE	*Instrukcja odwrotna do instrukcji ENTER, wykonywana po zakończeniu podprogramu*
RET	*Instrukcja powrotu z podprogramu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
stała,0	11001000 stała (2 bajty) stała (1 bajt)	enter 4,0	-	11	10
stała,1	11001000 stała (2 bajty) stała (1 bajt)	enter 0,1	-	15	12
stała,stała	11001000 stała (2 bajty) stała (1 bajt)	enter 6,2	-	15+4(n-1)	12+4(n-1)

PRZYKŁAD

Parametrami funkcji są adres tablicy znaków oraz jej długość. Funkcja przegląda tę tablicę i zlicza występujące w niej znaki przecinka (,).

```
; Poniższy podprogram używa konwencji wywoływania
; języka C. Parametry są umieszczane na stosie w porządku
; odwrotnym do ich występowania w wywołaniu funkcji.
; Do zapamiętania pozycji aktualnie przeglądanego znaku
; oraz liczby znalezionych przecinków podprogram korzysta
; z dwóch zmiennych lokalnych o rozmiarze słowa.
; Po zakończeniu funkcji liczba ta jest umieszczana
; w rejestrze AX, stos jest czyszczony i wykonuje się
; instrukcja powrotu.
; z poziomu języka C wywołanie tej funkcji powinno mieć
; postać:
; commas = Cfunc(String, Len);
        .286                        ; Udostępnienie
                                    ; instrukcji
                                    ; procesora 80286
        .MODEL SMALL
        .CODE
        PUBLIC   _Cfunc
_Cfunc  PROC
        enter  4,0                  ; Utwórz ramkę stosu
                                    ; z czterema bajtami
                                    ; zmiennych lokalnych
         mov   [bp-4], 0            ; Wyzeruj licznik
         mov   [bp-2], 0            ; Wyzeruj indeks
```

```
        mov     bx, [bp+4]          ; Pobierz adres łańcucha
Cloop:
        mov     ax, [bp+6]          ; Pobierz jego długość
        cmp     [bp-2], ax          ; Czy już przejrzany
                                    ; został cały łańcuch
        jge     Done
        cmp     BYTE PTR [bx], ',' ; Czy znak jest
                                    ; przecinkiem
        jnz     NotComma
        inc     WORD PTR [bp-4]     ; Zwiększ licznik
                                    ; przecinków
NotComma:
        inc     WORD PTR [bp-2]     ; Zwiększ indeks
        inc     bx                  ; Adres następnego
                                    ; znaku
        jmp     Cloop               ; Powtórz jeszcze raz
Done:
        leave                       ; Usuń ze stosu ramkę
        ret
_Cfunc  ENDP
```

Interrupt

88/86	286	386
●	●	●

INT

ZASTOSOWANIE

Wygenerowanie przerwania programowego.

SKŁADNIA

```
INT  numer_przerwania
```

ZNACZNIKI

Znaczniki zezwolenia na przerwanie i pracy krokowej są zerowane.

KOMENTARZ

Instrukcja INT jest podobna do rozkazu CALL, z tym że oprócz adresu powrotu odkłada ona na stos również rejestr znaczników. Inny jest też sposób wyznaczania adresu skoku. Rozkaz INT zawiera numer przerwania. Numer ten jest traktowany jako indeks w tablicy wektorów przerwań. Tablica ta zawiera adresy procedur obsługi wszystkich przerwań.
Powrót z procedury obsługi przerwania jest możliwy dzięki instrukcji IRET. Powoduje ona zdjęcie ze stosu dalekiego adresu powrotu oraz rejestru znaczników.
Ten typ przerwań jest używany przez programy, których funkcje muszą być dostępne dla innych programów. Numer przerwania może bowiem pozostać niezmieniony, mimo że pro-

cedura, obsługująca go, może znajdować się pod różnymi adresami. Dobrym przykładem w ten sposób wywoływanego programu jest MS-DOS czy BIOS.

PATRZ

CALL, RET	*Instrukcje dotyczące zwykłych procedur*
INTO	*Przerwanie w przypadku wystąpienia nadmiaru*
IRET	*Instrukcja powrotu z podprogramu obsługi przerwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
stała	11001101 stała	int 21h	71/51	23	37
3	11001100	int 3	72/52	23	33

PRZYKŁAD

MS-DOS zawiera wiele funkcji, które są dostępne poprzez przerwanie programowe o numerze 21h. By wywołać żądaną funkcję, należy jej numer umieścić w rejestrze AH, a ewentualne parametry przekazać jej w innych rejestrach. Wynik działania funkcji będzie również w nich umieszczony.

Poniższy przykład ilustruje w jaki sposób otworzyć i zamknąć plik danych. Do tego celu wykorzystuje on funkcje systemu MS-DOS.

```
; Otwarcie pliku przez system MS-DOS
; Plik po dokonaniu na nim żądanych operacji jest zamykany
          mov   dx, OFFSET Filename  ; Pobierz adres
                                     ; nazwy pliku
          mov   ah, 3ch              ; Funkcja DOS-a,
                                     ; która otwiera plik
          int   21h                  ; Wywołanie DOS-a
          jnc   OpenOK               ; MS-DOS ustawia
                                     ; znacznik
                                     ; przeniesienia, gdy
                                     ; próba otworzenia
                                     ; pliku nie
                                     ; powiodła się
          .                          ; Operacje podejmowane
                                     ; gdy zaistnieje błąd
          .
          .
OpenOK:
          mov   bx, ax               ; Deskryptor pliku
                                     ; do BX
          .                          ; Operacje na pliku
          .
          .
```

```
mov   ah, 3fh          ; Funkcja DOS-a,
                       ; która zamyka plik
int   21h
```

Interrupt on Overflow

88/86	286	386
●	●	●

INTO

ZASTOSOWANIE

Wygenerowanie przerwania o numerze 4 gdy znacznik nadmiaru jest ustawiony .

SKŁADNIA

INTO

ZNACZNIKI

Jeśli przerwanie zostało wygenerowane, to znaczniki zezwolenia na przerwanie i pracy krokowej są zerowane.

KOMENTARZ

Standardowa obsługa przerwania numer 4 składa się tylko z instrukcji IRET. Oznacza to, że rozkaz INTO odłoży na stos rejestr znaczników, rejestry IP oraz CS i spowoduje skok do instrukcji IRET. Spowoduje to zakończenie obsługi przerwania bez potrzeby podejmowania jakichkolwiek czynności. Należy samodzielnie zmienić wektor przerwań, by wskazywał na funkcję, która podejmie odpowiednią działalność w wypadku wystąpienia nadmiaru. Do ustawienia wektora powinna być wykorzystana funkcja systemowa o numerze **25h**. Stary wektor powinien zostać zapamiętany, aby później mógł być odtworzony. Służy do tego celu funkcja o numerze **35h**.

PATRZ

INT	*Wygenerowanie przerwania programowego*
IRET	*Powrót z przerwania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11001110	into	73/53	24	35

PRZYKŁAD

By prawidłowo wykorzystać instrukcję INTO, należy najpierw ustawić odpowiednią procedurę obsługi przerwania numer 4. Należy tylko pamiętać o przywróceniu dotychczasowej procedury obsługi przed zakończeniem programu. W przeciwnym wypadku wektor przerwań będzie wskazywał na procedurę, która została już z pamięci usunięta. Pojawienie się przerwania spowoduje więc zawieszenie się systemu.

INTO

Instrukcja INTO jest najczęściej używana po operacjach arytmetycznych, które mogą spowodować nadmiar. W poniższym przykładzie efektem będzie pojawienie się stosownego komunikatu.

```
; Zastąpienie starego wektora przerwania 4, wykonanie
; obliczeń, w wypadku powstania nadmiaru instrukcja INTO
; spowoduje wyświetlenie odpowiedniego komunikatu.
          .MODEL SMALL
          .DATA
OldInt    DD   0                          ; Zmienna, w której
                                          ; będzie pamiętany
                                          ; dotychczasowy
                                          ; wektor przerwań
Overmsg   DB   "Overflow error",0ah, 0dh,"$"
          .CODE
Start     PROC
          mov   ax, SEG DGROUP
          mov   ds, ax
          mov   ax, 3504h                 ; Numer funkcji,
                                          ; która odczytuje
                                          ; wektora przerwań
                                          ; (przerwanie 4)
          int   21h                       ; Wywołanie MS-DOS
          mov   WORD PTR OldInt, bx       ; Zapamiętanie
                                          ; przemieszczenia
          mov   WORD PTR OldInt+2, es     ; Zapamiętanie
                                          ; numeru segmentu
          push  ds
          push  cs                        ; Skopiowanie numeru
                                          ; segmentu kodu do
          pop   es                        ; rejestru
                                          ; segmentowego
                                          ; dodatkowego
          mov   dx, OFFSET Overhand
          mov   ax, 2504h                 ; Ustawienie wektora
                                          ; przerwania 4
          int   21h
          mov   al, 30
          mov   cl, 20
          mul   cl                        ; Instrukcja ta
                                          ; spowoduje
                                          ; powstanie nadmiaru
          into                            ; Wywołanie
                                          ; procedury do
                                          ; obsługi błędu
                                          ; nadmiaru
```

```
          push  ds
          mov   ah, 4ch                 ; Funkcja, która
                                        ; kończy program
          int   21h
Start     ENDP
; To jest procedura obsługi przerwania numer 4. Wartości
; wszystkich użytych rejestrów muszą zostać zapamiętane
; na stosie, a przed wyjściem z procedury odtworzone.
Overhand  PROC  FAR                     ; Wszystkie
                                        ; podprogramy muszą
                                        ; być traktowane
                                        ; jako dalekie
          push  ax                      ; Zachowaj rejestry
          push  dx
          push  ds
          mov   ax, SEG DGROUP          ; Pobierz numer
                                        ; segmentu danych
          mov   ds, ax                  ; i umieść go w DS
          mov   dx, OFFSET Overmsg
          mov   ah, 9                   ; Funkcja, która
                                        ; wyświetla ciąg
                                        ; znaków
          int   21h                     ; Wywołanie MS-DOS
          pop   ds
          pop   dx
          pop   ax
Overhand  ENDP
          .STACK
          END   Start
```

Interrupt Return

88/86	286	386
●	●	●

IRET/IRETD

ZASTOSOWANIE

Powrót z podprogramu wywołanego przez przerwanie. Najpierw ze stosu pobierany jest daleki adres powrotu, a następnie odtwarzane są znaczniki.

SKŁADNIA

IRET

IRETD — *Pobranie podwójnego słowa do rejestru znaczników na procesorze 80386*

IRET/IRETD

ZNACZNIKI

Znaczniki są ustawiane zgodnie z wartościami pobranymi ze stosu.

KOMENTARZ

Przerwanie może nadejść w każdej chwili. Ponieważ nie ma sposobu na stwierdzenie, jaki program został przerwany, to procedura obsługi musi przywrócić stan procesora sprzed pojawienia się przerwania. Przyjęcie przerwania zmienia stan kilku znaczników. Muszą więc one być zapamiętane, a po zakończeniu podprogramu odtworzone, właśnie dzięki instrukcji IRET.

PATRZ

CALL	*Wywołanie normalnych podprogramów*
INT	*Wygenerowanie przerwania programowego.*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11001111	iret	44/32	17	32

PRZYKŁAD

Poniższa procedura jest wywoływana po przyjściu przerwania od układu szeregowego wejścia/wyjścia. Pobiera ona daną od układu i wstawia do bufora okrężnego. Przed powrotem zeruje ona jeszcze sterownik przerwań 8259, by kolejne przerwania mogły zostać przyjęte.

```
; Odczyt portu UART po pojawieniu się przerwania
; z układu szeregowego wejścia/wyjścia
          .DATA
RingBuf   DB   1024 dup(?)       ; Bufor na dane
Bptr      DW   0                 ; Wskaźnik na pierwszy
                                 ; wolny znak w buforze
          .CODE
Serial    PROC  FAR
          push  ax               ; Zachowaj rejestry
          push  bx
          push  dx
          push  ds
          mov   dx, 3f8h         ; Adres portu szeregowego
          in    al, dx
          mov   ds, SEG DGROUP   ; Pobierz numer segmentu
                                 ; danych
          mov   bx, Bptr         ; Pobierz wskaźnik danych
                                 ; w buforze
          mov   RingBuf[bx], al
          inc   bx
          and   bx, 1023         ; Wskaźnik nie może
```

```
                                        ; przekroczyć maksymalnej
                                        ; długości bufora
        mov     Bptr, bx                ; Zachowaj nowy wskaźnik
        pop     ds                      ; Odtwórz rejestry
        pop     dx
        pop     bx
        mov     al, 20h                 ; Wyzeruj sterownik
                                        ; przerwań
        out     20h, al
        pop     ax
        iret
Serial  ENDP
```

Jump on Condition

88/86	286	386
●	●	●

JB lub JNAE, JAE lub JNB, JBE lub JNA, JA lub JNBE, JL lub JNGE, JGE lub JNL, JLE lub JNG, JG lub JNLE, JE lub JZ, JNE lub JNZ, JS, JNS, JC, JNC, JO, JNO, JP lub JPE, JNP lub JPO

ZASTOSOWANIE

Wykonanie krótkiego skoku, gdy jest spełniony pewien warunek. Warunki te zostały opisane w tabeli 6-2.

SKŁADNIA

```
Jwarunek  etykieta
```

ZNACZNIKI

Bez zmian

KOMENTARZ

Skoki warunkowe są odpowiedzialne za podejmowanie decyzji w programie. Zwykle występują one po instrukcjach CMP lub TEST.
Dla procesora 80386 skoki te mogą być również bliskie. Powoduje to zwiększenie ich zasięgu z <-128, 127> do <-32768, 32767> bajtów.

PATRZ

CMP, TEST	*Instrukcje, które zmieniają stan znaczników*
JMP	*Skok bezwarunkowy*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
short etykieta	0111cond przemieszczenie	jz dest	16(4)	7(3)	7(3)
near etykieta	00001111 1000cond	jg ndest	-	-	7(3)

W tabeli 6-3 pokazany jest sposób kodowania warunków.

Tabela 6-3. Instrukcje skoków warunkowych

Instrukcja	Kod warunku
JB lub JNAE	0010
JAE lub JNB	0011
JBE lub JNA	0110
JA lub JNBE	0111
JL lub JNGE	1100
JGE lub JNL	1101
JLE lub JNG	1110
JG lub JNLE	1111
JE lub JZ	0100
JNE lub JNZ	0101
JS	1000
JNS	1001
JC	0010
JNC	0011
JO	0000
JNO	0001
JP lub JPE	1010
JNP lub JPO	1011

PRZYKŁAD

Poniższy przykład sprawdza, czy bajt, który znajduje się w AL, jest cyfrą dziesiętną. Jeśli nie, to rejestr AL jest zerowany.

```
; Sprawdź czy w AL jest cyfrą dziesiętną.
            .CODE
IsDigit     PROC  NEAR
            cmp   al, '0'    ; Porównaj z zerem
            jb    NotDigit   ; Jeśli AL < '0', nie
                             ; jest to cyfra
            cmp   al, '9'    ; Porównaj z dziewiątką
            ja    NotDigit   ; Jeśli AL > '9', nie
```

```
                                ; jest cyfrą
            ret                 ; Wszystko jest OK, powrót
NotDigit:
            xor   al, al        ; Wyzeruj AL, gdy znajdująca
                                ; się w nim wartość nie jest
                                ; cyfrą dziesiętną
            ret
IsDigit     ENDP
```

Jump if CX is Zero

88/86	286	386
●	●	●

JCXZ, JECXZ

ZASTOSOWANIE

Wykonanie krótkiego skoku jeśli rejestr CX jest równy zeru. Dla procesora 80386 instrukcja JECXZ może być użyta do testowania rejestru ECX.

SKŁADNIA

```
JCXZ    etykieta
JECXZ   etykieta
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja **LOOPwarunek** oraz prefiks **REPwarunek** mogą zakończyć działanie pętli zanim rejestr CX osiągnie wartość zero. Czasami zachodzi potrzeba stwierdzenia, co było przyczyną wyjścia z pętli; został spełniony warunek czy też wyzerował się rejestr CX. Używając tych rozkazów, można łatwo wykryć tą przyczynę.

PATRZ

CMPS	*Porównywanie łańcuchów*
SCAS	*Przeszukiwanie łańcucha*
REPE, REPNE	*Warunkowe przedrostki repetycyjne*
LOOPE, LOOPNE	*Warunkowe instrukcje pętli*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
etykieta	11100011 przemieszczenie	jcxz notfound	18(6)	8(4)	9(5)

JMP

PRZYKŁAD

Przy przeszukiwaniu bufora znaków konieczne są dwa testy. Jeden używany jest do wykrycia szukanego znaku, drugi zaś odpowiedzialny jest za wykrycie końca bufora. Poniższy program przykładowy przegląda łańcuch znaków w poszukiwaniu nawiasu otwierającego ['(']. Długość łańcucha jest przechowywana w pierwszym jego bajcie.

```
; Szukanie w łańcuchu znaku '('. Adres łańcucha podany
; jest w rejestrze SI
            .CODE
FindParen   PROC  NEAR
            sub   cx, cx     ; Wyzeruj rejestr CX
            mov   cl, [si]   ; Pobierz długość łańcucha
            inc   si
            mov   al, '('    ; Ustal poszukiwany
                             ; znak na '('
            repne scansb     ; Wyszukaj go
            jcxz  NotFound   ; Jeśli CX = 0, znaku '('
                             ; nie znaleziono
            ret              ; Powrót, rejestr SI zawiera
                             ; adres pierwszego
                             ; wystąpienia znaku '('
NotFound:
            sub   si, si     ; Wyzeruj rejestr SI dla
                             ; oznaczenia, że wyszukiwanie
                             ; nie zakończyło się sukcesem
            ret
FindParen   ENDP
```

Jump

88/86	286	386
●	●	●

JMP

ZASTOSOWANIE

Bezwarunkowy skok pod podany adres.

SKŁADNIA

```
JMP   etykieta
JMP   SHORT etykieta
JMP   FAR PTR etykieta
JMP   rejestr_lub_pamięć
JMP   WORD PTR pamięć
JMP   DWORD PTR pamięć
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja skoku JMP ma trzy różne wersje, które różnią się między sobą tylko zasięgiem skoku. Zakres skoków krótkich wynosi od -128 do 127 bajtów. Adres skoku może być więc podany w jednym bajcie. By użyć tej wersji, etykieta musi zostać poprzedzona modyfikatorem SHORT.

Zasięg skoków bliskich zwiększa się i wynosi od -32,768 do 32,767 bajtów. W tym wypadku adres musi już być dwubajtowy. Asembler zastosuje tę wersję, gdy etykieta jest zadeklarowana jako NEAR.

Skoki dalekie pozwalają przekazać sterowanie do dowolnego miejsca pamięci. Adres jest czterobajtowy i składa się z dwóch bajtów przemieszczenia i dwóch bajtów numeru segmentu. Asembler zastosuje tę wersję, gdy etykieta będzie zadeklarowana jako FAR. Użycie skoku dalekiego można wymusić, poprzedzając etykietę modyfikatorem FAR PTR.

Przy skokach bezpośrednich adres skoku jest pobierany z rejestru lub miejsca w pamięci. Wszystkie skoki poprzez rejestr są skokami bliskimi. Natomiast rodzaj skoku poprzez pamięć zależy od jej rozmiaru. Gdy jest ona słowem, to wykonany zostanie skok bliski, gdy zaś podwójnym słowem - skok daleki. Wszelkie niejednoznaczności można rozstrzygnąć jawnie, używając odpowiednich modyfikatorów WORD lub DWORD.

PATRZ

NEAR, FAR, PTR, SHORT, DWORD, WORD	*Modyfikatory*
JB lub JNAE, JAE lub JNB, JBE lub JNA, JA lub JNBE, JL lub JNGE, JGE lub JNL, JLE lub JNG, JG lub JNLE, JE lub JZ, JNE lub JNZ, JS, JNS, JC, JNC, JO, JNO, JP lub JPE, JNP lub JPO	*Skoki warunkowe*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
short etykieta	11101011 przemieszczenie	jmp SHORT lab	15	7	7
near etykieta	11101001 przemieszczenie	jmp close	15	7	7
far etykieta	11101010 przemieszczenie	jmp distant	15	11	12
rejestr	11111111 11,100,r/m	jmp ax	11	7	7
pamięć near	11111111 mod,100,r/m przemieszczenie	jmp WORD [bx]	18+EA	11	10

LEAVE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
pamięć far	11111111 mod,101,r/m przemieszczenie	jmp DWORD [si]	24+EA	15	12

PRZYKŁAD

Jeżeli wywołujemy MS-DOS poprzez przerwanie 21h, wówczas rejestrze AH przekazywany jest numer żądanej funkcji. Sposób ten jest stosowany w wielu innych programach. Do wyznaczania adresu funkcji programy te wykorzystują tablice adresów funkcji. W poniższym przykładzie numer funkcji jest traktowany jako indeks tej tablicy i na tej podstawie jest wyznaczany adres skoku.

```
; Skok do funkcji o numerze podanym w AH. Adresy funkcji
; zgromadzone są w tablicy FuncTab
         .DATA
FuncTab  DD   Funct1, Funct2      ; Tablica adresów funkcji
         .CODE
DoFunc   PROC
         sub   bx, bx             ; Wyzeruj rejestr BX
         mov   bl, ah             ; Prześlij numer
                                  ; funkcji do BL
         shl   bx, 1              ; Pomnóż przez cztery
                                  ; dla wyznaczenia
         shl   bx, 1              ; przesunięcia w tablicy
         jmp   DWORD FuncTab[bx]  ; Wykonaj skok do funkcji
DoFunc   ENDP
```

Leave a Procedure

88/86	286	386
	●	●

LEAVE

ZASTOSOWANIE

Usunięcie ze stosu ramki stosu, utworzonej przez instrukcję ENTER.

SKŁADNIA

LEAVE

ZNACZNIKI

Bez zmian.

KOMENTARZ

Ramki stosu są używane do przekazania do podprogramu parametrów oraz do stworzenia zmiennych lokalnych. W procesorach 80286 i 80386 zarządzanie nimi ułatwiają instrukcje ENTER i LEAVE.

Instrukcja LEAVE wpisuje do wskaźnika stosu SP wartość, która znajduje się w rejestrze BP. Krok ten powoduje zwolnienie pamięci, przydzielonej na zmienne lokalne. Następnie ze stosu pobierana jest stara wartość rejestru BP.

W procesorach 8088 i 8086 rozkaz ten może być zastąpiony poniższą sekwencją instrukcji:

```
mov   sp, bp
pop   bp
```

PATRZ

CALL, RET	*Wywołanie i powrót z podprogramu*
ENTER	*Stworzenie ramki stosu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	111001000	leave	-	5	4

PRZYKŁAD

Patrz przykład do instrukcji ENTER.

Loop, Loop on Condition

88/86	286	386
●	●	●

LOOP, LOOPE, LOOPNE

ZASTOSOWANIE

Dekrementacja rejestru CX, a następnie, jeśli jego wartość jest różna od zera, wykonanie krótkiego skoku. Przed wykonaniem skoku instrukcje warunkowe badają ponadto stan znacznika zera.

SKŁADNIA

```
LOOP   etykieta
LOOPE  etykieta
LOOPNE etykieta
```

ZNACZNIKI

Bez zmian.

LOOP, LOOPE, LOOPNE

KOMENTARZ

Instrukcja LOOP pozwala na skonstruowanie pętli, która wykonuje się określoną liczbę razy. Instrukcje LOOPE i LOOPNE pozwalają ponadto, w warunku zakończenia pętli, uwzględnić stan znacznika zera.
Wykonywany skok jest skokiem krótkim. Oznacza to, że etykieta może być odległa od tej instrukcji o od -128 do 127 bajtów.

PATRZ

REP, REPE, REPNE	*Używane do tworzenia pętli, które składają się z jednej instrukcji*
JMP	*Instrukcja skoku bezwarunkowego*
JCXZ	*Skok, gdy rejestr CX jest równy zeru*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
LOOP	11100010 przemieszczenie	loop cloop	17(5)	8(4)	11
LOOPE	11100001 przemieszczenie	loope again	18(6)	8(4)	11
LOOPNE	11100000 przemieszczenie	loopne next	19(5)	8(4)	11

PRZYKŁAD

Poniższy przykład znajduje łączną długość grupy ciągów znaków. Wskaźniki do nich znajdują się w tablicy **PtrTab**. Każdy łańcuch kończy się bajtem 0, a jego długość musi być mniejsza niż 80 znaków. Łączna długość jest umieszczana w rejestrze DX.

```
; Wyznacz łączną długość grupy łańcuchów
            .DATA
Str1        "Oto pierwszy ciąg znaków", 0
Str2        "Oto drugi ciąg znaków", 0
Str3        "Oto trzeci ciąg znaków", 0
Str4        "Oto czwarty ciąg znaków", 0
Str5        "Oto piąty ciąg znaków", 0
Str6        "Oto szósty ciąg znaków", 0
Str7        "Oto siódmy ciąg znaków", 0
Str8        "Oto ósmy ciąg znaków", 0
Str9        "Oto dziewiąty ciąg znaków", 0
Str10       "Oto dziesiąty ciąg znaków", 0
PtrTab      DW   Str1, Str2, Str3, Str4, Str5
            DW   Str6, Str7, Str8, Str9, Str10
            .CODE
GroupLen    PROC
            xor   dx, dx              ; Wyzeruj rozmiar
```

```
            mov     bx, OFFSET PtrTab   ; Pobierz adres
                                        ; tablicy
            mov     cx, 10              ; Liczba łańcuchów
                                        ; w tablicy
OLoop:
            mov     si, [bx]            ; Pobierz adres
                                        ; łańcucha
            inc     bx                  ; Następny adres
            inc     bx
            push    cx                  ; Zapamiętaj stan
                                        ; licznika zewnętrznej
                                        ; pętli
            mov     cx, 80              ; Maksymalna długość
                                        ; łańcucha
ILoop:
            mov     al, [si]            ; Pobierz znak
            inc     si                  ; Następny bajt
            or      al, al              ; Czy to zerowy bajt
            loopne  ILoop               ; Jeśli nie,
                                        ; powtarzaj pętlę
            add     dx, 80              ; Dodaj do licznika
                                        ; maksymalną długość
            sub     dx, cx              ; Odejmij liczbę
                                        ; niezapełnionych
                                        ; bajtów
            pop     cx                  ; Odtwórz licznik
            loop    OLoop
            ret
GroupLen    ENDP
```

Return from a Subroutine

88/86	286	386
●	●	●

RET, RETN, RETF

ZASTOSOWANIE

Pobranie ze stosu adresu powrotu i przekazanie do niego sterowania. Druga wersja tej instrukcji dodaje ponadto do wskaźnika stosu swój argument.

SKŁADNIA

```
RET
RET  rozmiar_parametrów
```

RET, RETN, RETF

ZNACZNIKI

Bez zmian.

KOMENTARZ

Funkcja może być wywoływana jako bliska lub daleka. Wpływa to na liczbę bajtów, które składają się na adres powrotu. Asembler wybiera odpowiedni rodzaj instrukcji powrotu zależnie od typu procedury (NEAR lub FAR). Wybór ten można wymusić jawnie, podając żądany rozkaz powrotu (RETN lub RETF).

Gdy do przekazania parametrów posłużył stos, to po zakończeniu podprogramu muszą one zostać z niego usunięte. Ponieważ przed argumentami wywołania znajduje się jeszcze adres powrotu, to nie można ich po prostu zdjąć ze stosu. W tym wypadku może być natomiast zastosowana instrukcja RET n. Po pobraniu ze stosu adresu powrotu doda ona do wskaźnika stosu wartość n.

PATRZ

CALL *Wywołanie podprogramu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
RETN	11000011	retn	20/16	11	10
RETN stała	11000010 stała	ret 4	24/20	11	10
RETF	11001011	retf	34/26	15	18
RETF stała	11001010 stała	retf 12	33/25	15	18

PRZYKŁAD

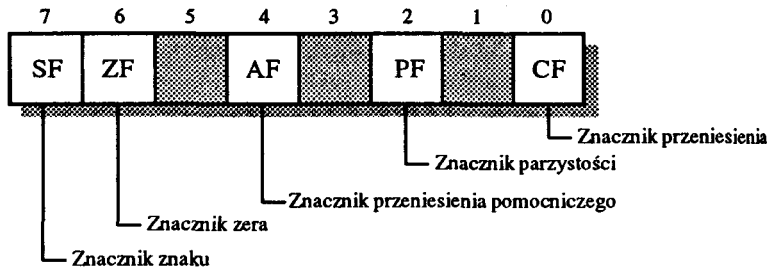
Patrz przykład do instrukcji CALL.

Rozdział 7 Sterowanie procesorem i operacje w trybie chronionym

W procesorach 80x86 istnieje jeden lub kilka rejestrów, które przechowują stan znaczników oraz zawierają inne informacje o stanie procesora. Zwykle znaczniki są ustawiane na podstawie operacji arytmetycznych lub logicznych. By sprawdzić ich stan, używa się instrukcji skoków warunkowych lub rozkazu SET. W pierwszej części rozdziału zostaną opisane instrukcje, które w sposób bezpośredni zmieniają stan znaczników.
Jednym ze sposobów na zwiększenie szybkości przetwarzania komputera jest zastosowanie większej liczby procesorów. Dwa procesory, które pracują jednocześnie nad jednym zadaniem, wykonują je o wiele szybciej niż procesor pojedynczy. W systemach 80x86 stosuje się często specjalizowane procesory zwane koprocesorami, które przejmują pewne części zadań systemu. Istnieją specjalne instrukcje sterujące procesora, które pozwalają na obsługę koprocesora. Opisane są one w drugiej części rozdziału.
Użycie wielu procesorów do zwiększenia mocy komputera jest rzeczywiście świetnym rozwiązaniem. Istnieją jednak takie sytuacje, gdy procesor bezczynnie marnuje czas, na przykład wtedy, gdy komputer czeka na wciśnięcie przez użytkownika klawisza. Można wtedy przełączyć procesor na wykonywanie innego zadania. Tak więc, gdy ty na przykład analizujesz w edytorze program asemblerowy, to w tym samym czasie mogą trwać obliczenia w jakimś arkuszu kalkulacyjnym. Dzielenie czasu procesora pomiędzy wiele zadań nazywa się wielozadaniowością.
Na wszystkich procesorach 80x86 wielozadaniowość może być zrealizowana. Jednak w procesorach 8088 i 8086 istnieje wiele problemów związanych z separacją zadań. Procesory 80286 i 80386 posiadają natomiast specjalny tryb, dzięki któremu jest to znacznie prostsze. Tryb ten zwany jest trybem chronionym. Ponadto procesory te mają kilka dodatkowych instrukcji, które ułatwiają przełączanie zadań. Ostatnia część tego rozdziału ilustruje zastosowanie trybu chronionego w systemie MS-DOS.

Operacje na znacznikach

W procesorach 80x86 wszystkie znaczniki zgrupowane są w jednym bajcie. Bajt ten może zostać przesłany do rejestru AH i odwrotnie. Operacji tych dokonują instrukcje LAHF oraz SAHF. Rozkazy te są użyteczne, gdy należy zachować lub odtworzyć stan procesora. Na przykład, w systemie wielozadaniowym należy zapamiętać stan bieżącego zadania (wszystkie rejestry i znaczniki), a odtworzyć stan zadania następnego. Rysunek 7-1 pokazuje rozmieszczenie bitów w bajcie wykorzystywanym przez instrukcje LAHF i SAHF.
Procesory 80286 i 80386 posiadają jeszcze jeden zestaw znaczników, które sterują trybem procesora. Znaczniki te są umieszczone w słowie stanu procesora (MSW - Machine Status Word). Programy z najwyższym poziomem uprzywilejowania mogą przesyłać, pobierać

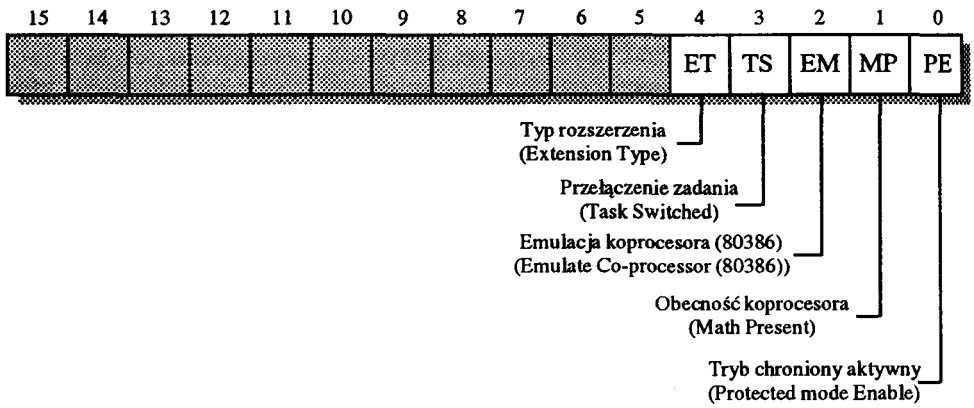


Rysunek 7-1. ***Młodsze słowo rejestru znaczników używane przez rozkazy LAHF***

i zapisywać dane do tego rejestru. Odpowiednie instrukcje to LMSW (Load MSW) i SMSW (Save MSW) o formacie:

```
LMSW    reg/mem
```

gdzie reg/mem oznacza dowolny rejestr lub komórkę pamięci. Rozkaz LMSW powoduje załadowanie MSW daną, wskazywaną przez argument, a SMSW przesyła MSW do argumentu. Rysunek 7-2 pokazuje format słowa MSW.



Rysunek 7-2. ***Słowo stanu procesora.***

Ustawianie i zerowanie znaczników

Niektóre znaczniki mogą być bezpośrednio ustawiane lub zerowane oddzielną instrukcją. Przykładem takich instrukcji są CLD i STD - w przykładach, zawartych w rozdziale 4, często stosowane do ustawiania kierunku operacji łańcuchowych.

Stan znacznika przeniesienia może być zmieniany instrukcjami STC (Ustaw znacznik przeniesienia), CMC (Zaneguj stan znacznika przeniesienia) oraz CLC (Wyzeruj znacznik przeniesienia).

Instrukcje STI oraz CLI są używane do sterowania znacznikiem zezwolenia na przerwanie. Przerwania sprzętowe, takie jak przerwanie z klawiatury czy portu szeregowego, mogą być

odebrane przez procesor tylko wtedy, gdy znacznik ten jest ustawiony. Podprogramy, które nie mogą zostać przerwane, powinny więc znacznik ten wyzerować. Należy jednak pamiętać o ponownym jego ustawieniu po zakończeniu procedury. Zaniedbanie tego spowoduje brak reakcji ze strony komputera na pojawienie się przerwania od urządzeń I/O np. na naciśnięcie klawisza.

Sterowanie procesorem

System komputerowy nie składa się tylko z samego procesora. Zwykle zawiera on jeszcze porty szeregowe, dyski, pamięć, klawiaturę itp. Wszystkie te podsystemy muszą ze sobą współpracować. Na przykład, gdy pamięć jest wolniejsza niż procesor, to potrzebne są dodatkowe cykle oczekiwania na to, by dana została pobrana z pamięci.
Każde urządzenie ma swój własny sposób na synchronizację z otoczeniem. Istnieje pięć instrukcji, które pozwalają procesorowi zatrzymywać, opóźniać, wydawać polecenia i blokować inne urządzenia.

Opóźnianie i zatrzymywanie

Gdy procesor musi czekać na wolniejsze urządzenie, to najprostszą rzeczą, jaką może wtedy robić jest "nie robienie niczego". Właśnie wykonanie instrukcji **NOP** nie odnosi żadnego skutku. Zajmuje ona jeden bajt pamięci i wymaga trzech taktów zegara. Oba te fakty mają swoje zastosowanie. Tak naprawdę rozkaz **NOP** jest zastępowany przez asembler instrukcją **XCHG AX,AX**.
Podczas uruchamiania programu może się okazać, że pewien fragment kodu wydaje się zbędny. Aby go usunąć, trzeba będzie wyjść z uruchamiacza, poprawić w edytorze tekst źródłowy, skompilować program i dopiero teraz rozpocząć ponowne testowanie. Prostszym sposobem poznania skutków usunięcia tego kodu jest zastąpienie go instrukcjami pustymi NOP. Jeśli okaże się, że tak poprawiony program działa poprawnie, to można dokonać niezbędnych zmian w plikach źródłowych.
Innym zastosowaniem instrukcji NOP jest wypełnienie miejsca, by następny rozkaz znajdował się pod parzystym adresem. Rozkazy takie wykonują się bowiem szybciej. Ponieważ instrukcja NOP również zajmuje czas, to rozwiązanie takie jest efektywne tylko wtedy, gdy spowoduje, że cały ciąg kolejnych rozkazów znajdzie się pod parzystymi adresami.
Instrukcja NOP może być również użyta przy oczekiwaniu na gotowość pewnego urządzenia. Po wysłaniu do niego komendy, np instrukcją OUT, może okazać się konieczne odczekanie pewnego czasu na to, by urządzenie mogło wysłać odpowiedź zwrotną. Kilka rozkazów NOP umieszczonych za instrukcją wejścia/wyjścia, może być wystarczającym rozwiązaniem. Jeśli okres ten ma być trochę dłuższy może zostać użyta instrukcja **JMP $+2**. Jest to po prostu skok do następnego rozkazu. W skutkach instrukcja ta jest więc podobna do instrukcji NOP, ale zajmuje ona dwa bajty i wymaga 15 taktów zegara. Przy konstruowaniu tego typu opóźnień należy zwrócić uwagę na to, że długość taktu zegara zależy od rodzaju zastosowanego procesora. Jeśli więc konieczne są precyzyjne odstępy czasu, to należy raczej skorzystać z przerwania **1Ah** lub bezpośrednio zaprogramować układ czasowy 8253.
Innym sposobem czekania na gotowość urządzenia jest zatrzymanie procesora. Robi to instrukcja **HLT**. Zatrzymany procesor nie wykonuje żadnych czynności, aż do nadejścia

przerwania. Powoduje ono "przebudzenie" procesora. Kontynuuje on dalej swoją działalność od następnej instrukcji za instrukcją HLT. Rozkaz ten nie jest często używany. Zwykle bowiem na procesor czekają jednak jakieś zadania do wykonania, a ponadto duża ilość przerwań w systemie (zegar, klawiatura, port szeregowy itp) sprawia, że procesor nie pozostaje długo w stanie zawieszenia.

Współpraca z koprocesorem

Zadaniem koprocesora jest wykonywanie specjalnych zleceń pod kontrolą jednostki centralnej. Najpopularniejszymi koprocesorami są koprocesory serii 80x87, które operują na liczbach zmiennopozycyjnych. Inne mogą zajmować się np. obsługą grafiki lub urządzeń wejścia/wyjścia. Dzięki skoncentrowaniu się na wykonywaniu tylko określonych czynności, koprocesory robią je znacznie szybciej i lepiej niż jednostka ogólnego przeznaczenia jaką jest procesor 80x86. Poza tym, w czasie, gdy koprocesor wykonuje powierzone mu zadanie, procesor może zająć się następną operacją.

BLOKOWANIE MAGISTRALI

Magistrala to grupa połączeń i układów, które je obsługują i pozwalają na komunikację pomiędzy różnymi częściami komputera. Procesor główny oraz koprocesor komunikują się z pamięcią poprzez magistralę danych. Problem pojawia się wtedy, gdy oba jednocześnie będą chciały odwołać się do tego samego miejsca pamięci. Przypuśćmy na przykład, że procesor chce zwiększyć o jeden zawartość komórki pamięci. Najpierw odczytuje on słowo z pamięci, potem zwiększa jego wartość i zapisuje ponownie. Ale co się stanie, gdy w w tym czasie koprocesor zechce również zmodyfikować to samo słowo? Jeśli odczyta on daną po procesorze głównym, to będzie ona błędna. Jeśli natomiast koprocesor pierwszy pobierze słowo z pamięci, wówczas procesor otrzyma daną nieprawidłową. W każdym razie, w obu wypadkach rezultat będzie błędny.
Rozwiązaniem tego problemu jest zastosowanie przedrostka LOCK. Powoduje on zablokowanie dostępu do pamięci innym procesorom na czas wykonania następnej instrukcji. Operacja zwiększania powinna być więc zapisana następująco:

```
lock   inc  [bx]
```

SYNCHRONIZACJA KOPROCESORA

Innym sposobem zapobieżenia kolizjom pomiędzy procesorem i koprocesorem jest wymuszenie oczekiwania przez procesor główny na zakończenie instrukcji koprocesora. Służy do tego rozkaz WAIT. Jeśli twemu programowi potrzebne są dane wygenerowane przez koprocesor, to należy zaczekać na ich skompletowanie. Każda próba odczytu danych w trakcie trwania obliczeń koprocesora spowoduje pobranie błędnego rezultatu.
Układ 8087 może w tym samym czasie wykonywać tylko pojedynczą instrukcję. Dlatego też przed każdą instrukcją zmiennoprzecinkową powinien zostać umieszczony rozkaz WAIT. Użycie dyrektywy .8087 i mnemoników instrukcji zmiennoprzecinkowych spowoduje, że Turbo Assembler automatycznie wstawi konieczne rozkazy WAIT.

WYSYŁANIE KOPROCESOROWI ROZKAZÓW

Koprocesor znajduje się pod kontrolą procesora głównego. Od niego też otrzymuje wszystkie rozkazy. Służy do tego celu instrukcja ESC o formacie:

```
ESC   instrukcja, reg/mem
```

Instrukcja ta zawiera sześć bitów informacji, które są przesyłane do koprocesora. Zwykle nie używa się bezpośrednio tego rozkazu. Asembler bowiem zapewnia zestaw instrukcji zmiennoprzecinkowych, które to dopiero przez niego są zamieniane na odpowiednie instrukcje ESC.

Tryb chroniony

Przy dostępie do wspólnych obszarów danych przez procesor i koprocesor musiały zostać podjęte specjalne środki. Podobny problem występuje, gdy na jednym procesorze wykonywanych jest kilka zadań. Zadanie, które modyfikuje pewien obszar pamięci, musi być pewne, że wszystkie pozostałe mają zablokowany dostęp do niego. Gdy zadania wiedzą o swojej obecności, to problem ten mogą rozwiązać protokoły dostępu do obszarów danych. Natomiast gdy błędnie napisane zadanie zacznie losowo zmieniać zawartość pamięci, wówczas może to wywołać zawieszenie się systemu. By zabezpieczyć się przed taką sytuacją w procesorach 80286 i wyższych został wprowadzony specjalny tryb, zwany trybem chronionym. Wszystkie zadania są w nim od siebie odseparowane dzięki temu, że dostęp do różnych zasobów kontroluje procesor. Zadanie może jedynie wykorzystać te zasoby, do których w danej chwili ma prawa dostępu. Natomiast zadania, które próbują odwołać się do nieprzydzielonej pamięci czy urządzeń wejścia/wyjścia, zostają usuwane.
Procesor nadzoruje czynności wykonywane przez zadania, przydzielając poszczególnym fragmentom kodu odpowiedni priorytet. Programy na danym poziomie uprzywilejowania mogą korzystać z kodu, który ma inny priorytet tylko w pewnych okolicznościach. Pewne instrukcje procesora mogą być natomiast wykonywane tylko przez programy o najwyższym priorytecie. Rozkazy wejścia/wyjścia mogą być używane tylko na określonym przez pewne uprzywilejowane zadanie poziomie.

Przełączanie procesora w tryb chroniony

Po włączeniu zasilania procesory 80286 i 80386 pracują w trybie rzeczywistym. Można je wtedy uznać za szybkie procesory 8086 z rozszerzonym zestawem instrukcji. By skorzystać z możliwości separacji zadań i wielozadaniowości, procesory te należy przełączyć w tryb chroniony.
W trybie chronionym niektóre programy nie będą działać. Ich przykładem jest MS-DOS i BIOS. Oznacza to, że nie można ich funkcji zastosować w operacjach wejścia/wyjścia czy do obsługi przerwań.
By więc uruchomić program w systemie MS-DOS w trybie chronionym, należałoby zablokować przerwania i zrezygnować z operacji I/O. Tego typu programów nie ma jednak wiele. Innym sposobem jest napisanie, działających w trybie chronionym, procedur obsługi przerwań sprzętowych, a przełączanie do trybu rzeczywistego przed przerwaniami programowymi, które wywołują np. funkcje MS-DOS-a czy BIOS-u.
Poniższy przykład pokazuje, jak przełączyć procesor 80386 w tryb chroniony, a następnie z powrotem w tryb rzeczywisty. W trybie chronionym wszystkie przerwania są blokowane. Ponieważ zadaniem tego programu jest tylko przełączanie pomiędzy trybami, nie korzysta on z funkcji MS-DOS-a. Jednakże procedury GoProtect i GoReal będą zastosowane w przy-

kładach w dalszej części tego rozdziału. Struktury danych oraz użyte terminy zostaną omówione na kolejnych stronach.

```
; Przełączenie procesora w tryb chroniony
; Plik PROTECT.ASM
            .MODEL  SMALL                   ; Model ten musi być
                                            ; również użyty
                                            ; w innych modułach
            .386P                           ; Udostępnienie
                                            ; instrukcji
                                            ; procesora 80386
            .DATA
EXECUT      EQU     10011010b               ; Typ 16-bitowego
                                            ; segmentu kodu
RDWRDATA    EQU     10010010b               ; Typ segmentu danych
                                            ; z możliwością zapisu
                                            ; i odczytu
STACKSEG      EQU   10010110b               ; Typ segmentu stosu
STATUS_PORT   EQU   64                      ; Port statusu
                                            ; klawiatury
A20_PORT      EQU   0d1h
A20_ON        EQU   0dfh
A20_OFF       EQU   0ddh
KBD_PORT_A    EQU   60
; Poniższe definicje danych tworzą Globalną
; Tablicę Deskryptorów
GblDT       DW      0, 0, 0, 0              ; Pierwszy deskryptor
                                            ; (zarezerwowany)
            DW      0, 0, 0, 0              ; Deskryptor segmentu
                                            ; danych
            DW      0, 0, 0, 0              ; Deskryptor segmentu
                                            ; kodu
            DW      0, 0, 0, 0              ; Deskryptor segmentu
                                            ; stosu
; Poniższa wartość jest umieszczana w rejestrze Globalnej
Tablicy Deskryptorów
GblDscTbl   DW      4 * 8 - 1               ; Granica segmentu
            DD      0
            .CODE
; Program zaczyna się w trybie rzeczywistym. Wypełnia on
; najpierw struktury danych i wywołuje procedurę GoProtect.
; By powrócić do trybu rzeczywistego używana jest
; procedura GoReal.
Start       PROC
            mov     ax, SEG _DATA           ; Załaduj rejestr
                                            ; segmentu danych
```

```
            mov   ds, ax
            mov   si, OFFSET GblDT      ; Pobierz adres GDT
            add   si, 8                 ; Długość deskryptora
                                        ; danych
            mov   bx, 0FFFFh            ; Rozmiar segmentu
            mov   cx, RDWRDATA          ; Znaczniki
            call  SetDscrpt             ; Wypełnij deskryptor
            mov   ax, SEG _TEXT         ; Pobierz numer
                                        ; segmentu kodu
            add   si, 8                 ; Długość deskryptora
                                        ; kodu
            mov   bx, 0FFFFh            ; Rozmiar segmentu
            mov   cx, EXECUT            ; Znaczniki
            call  SetDscrpt             ; Wypełnij deskryptor
            mov   ax, SEG SSEG          ; Pobierz numer
                                        ; segmentu stosu
            add   si, 8                 ; Długość deskryptora
                                        ; stosu
            mov   bx, 0FFFFh            ; Granica stosu
            mov   cx, EXECUT            ; Znaczniki
            call  SetDscrpt             ; Wypełnij deskryptor
            call  GoProtect             ; Przełącz w tryb
                                        ; chroniony
            call  GoReal
            mov   ah, 4Ch
            int   21h
Start       ENDP
; Wypełnienie deskryptora
; SI = Adres deskryptora
; AX = Adres segmentowy selektora
; BX = Rozmiar segmentu
; CX = Znaczniki segmentu
SetDscrpt   PROC
            mov   dl, ah                ; Pobierz starszy bajt
                                        ; numeru segmentu
            shl   ax, 4; AX * 16
            shr   dl, 4                 ; Starszy półbajt w DL
            mov   WORD PTR [si], bx     ; Rozmiar
            mov   WORD PTR [si+2], ax   ; Młodsze słowo
                                        ; adresu bazowego
            mov   BYTE PTR [si+4], dl   ; Starszy bajt
                                        ; adresu bazowego
            mov   WORD PTR [si+5], cx   ; Znaczniki
            mov   BYTE PTR [si+7], 0    ; Dodatki
            ret
SetDscrpt   ENDP
```

```
; Przełączenie w tryb chroniony
GoProtect  PROC
           call  enable_a20
           mov   ax, SEG _DATA          ; Pobierz adres
                                        ; segmentu danych
           mov   dl, ah                 ; i użyj go do
                                        ; wyznaczenia adresu
           shl   ax, 4                  ; bazowego globalnej
                                        ; tablicy
           shr   dl, 4                  ; deskryptorów
           sub   dh, dh
           add   ax, OFFSET GblDT       ; Dodaj
                                        ; przemieszczenie
                                        ; tej tablicy
           adc   dl, 0
           mov   GblDscTbl+2, ax        ; Zapamiętaj adres
                                        ; bazowy
           mov   GblDscTbl+4, dx
           lgtd  PWORD PTR GblDscTbl    ; Załaduj GDTR
           cli  ; Wyłącz przerwania
           smsw  ax                     ; Pobierz słowo
                                        ; stanu procesora
           and   ax, 1Fh                ; Wyzeruj bit
                                        ; stronicowania
           or    ax, 1                  ; Ustaw bit PE
           lmsw  ax                     ; Załaduj słowo
                                        ; stanu procesora
           DB    0EAh                   ; Rozkaz dalekiego
                                        ; skoku
           DW    ClearPF                ; Przemieszczenie
           DW    10h                    ; Segment w trybie
                                        ; chronionym
ClearPF:
           mov   ax, 8                  ; Selektor segmentu
                                        ; danych
           mov   ds, ax
           mov   ax, 18h                ; Selektor segmentu
                                        ; stosu
           mov   ss, ax
           ret
GoProtect  ENDP
; Powrót do trybu rzeczywistego
GoReal     PROC
           mov   eax, CR0               ; Pobierz MSW
           and   eax, 07FFFFFFEh        ; Wyzeruj bit
                                        ; stronicowania i bit
```

```
                                        ; trybu chronionego
            mov   CR0, eax              ; Załaduj MSW
            DB    0EAh                  ; Rozkaz dalekiego
                                        ; skoku
            DW    ClearPF2              ; Przemieszczenie
            DW    _TEXT                 ; Segment w trybie
                                        ; rzeczywistym
ClearPF2:
            mov   ax, SEG DGROUP        ; Odtwórz numery
                                        ; segmentów dla
            mov   ds, ax                ; trybu rzeczywistego
            mov   ax, SEG SSEG
            mov   ss, ax
            sti                         ; Włącz przerwania
            call  disable_a20
            ret
GoReal      ENDP
PROC        enable_20   NEAR
; Uaktywnia linię adresową a20 dla adresów wirtualnych
            mov   al, A20_PORT
            out   STATUS_PORT, al
            mov   al, A20_ON
            out   KBD_PORT_A, al
            ret
ENDP        enable_a20
PROC        disable_20   NEAR
; Dezaktywuje linię adresową a20
            mov   al, A20_PORT
            out   STATUS_PORT, al
            mov   al, A20_OFF
            out   KBD_PORT_A, al
            ret
ENDP        disable_a20
; Definicja segmentu stosu
SSEG  SEGMENT  WORD STACK USE16 'STACK'
            DB  1024 DUP (?)
SSEG  ENDS
END
```

Globalna tablica deskryptorów GDT przechowywana jest w zmiennej **GblDT**. Zawiera ona informacje o wszystkich segmentach w trybie chronionym. Jej elementy są wypełniane danymi o segmentach programu w pierwszej części procedury **Start**. (Opis pól tej tablicy znajduje się w części tego rozdziału zatytułowanej "Tablice deskryptorów".)
Następnie wywoływana jest procedura **GoProtect**, która przełącza procesor w tryb chroniony. Najpierw przekazuje ona procesorowi informację o położeniu i rozmiarze tablicy GDT. Dane te są umieszczane w zmiennej **GblDscTbl**. Następnie, za pośrednictwem

rozkazu LGDT, dane są przesyłane do rejestru globalnej tablicy deskryptorów GDTR w procesorze. Po wyłączeniu przerwań instrukcją CLI, procesor jest już gotowy do przełączenia. Należy ustawić bit PE (Zezwolenie na tryb chroniony) w słowie stanu procesora MSW. Niestety, nie można tego zrobić pojedynczą instrukcją. Należy więc pobrać do rejestru słowo MSW (rozkaz LMSW), zmienić w nim odpowiedni bit i zapisać zmodyfikowaną wartość do MSW instrukcją SMSW.
W trybie chronionym należy zmienić wartości rejestrów segmentowych, tak by zawierały selektory odpowiednich segmentów, a nie ich numery. Każdy selektor wskazuje pojedynczy element tablicy GDT. W programie wartości selektorów są równe indeksowi w tej tablicy pomnożonemu przez 8. Pierwszym rejestrem segmentowym, ustawianym przez procedurę **GoProtect**, jest rejestr CS. Jedyną instrukcją, która to umożliwia jest rozkaz dalekiego skoku. Istnieją jednak dwa problemy, które utrudniają użycie tej instrukcji. Pierwszy z nich polega na tym, że Turbo Assembler stwierdzi, że skok daleki nie jest w tym wypadku konieczny i przekształci go na skok krótki. Drugi zaś wynika stąd, że kompilator używa numerów segmentów, natomiast w trybie chronionym konieczne jest podanie ich selektorów. Problem ten został rozwiązany przez bezpośrednie zapisanie tego rozkazu w języku maszynowym dyrektywami DB (Definicja bajtu) i DW (Definicja słowa).
Funkcja **GoReal** przełącza procesor 80386 w tryb rzeczywisty. ze względu na kompatybilność z procesorem 80286 nie jest możliwe użycie instrukcji SMSW do wyzerowania bitu PE. Jedynym sposobem odwołania się do rejestru MSW jest użycie rozkazu MOV, który dotyczy rejestru CR0. Pierwsze trzy instrukcje procedury zerują bit PE, a to powoduje przełączenie procesora w tryb rzeczywisty. Pierwszą rzeczą, która musi być teraz zrobiona jest ponowne załadowanie rejestrów segmentowych odpowiednimi adresami.

Pamięć w trybie chronionym

Jednym z zasobów, z którego korzystają wszystkie zadania jest pamięć. W pewnych jej obszarach zawarty jest ich kod, w innych dane. W trybie chronionym pamięć jest klasyfikowana ze względu na jej wykorzystanie. Zadanie, które chce się odwołać do pamięci, musi mieć do niej prawo dostępu, a także musi wykonywać na niej prawidłową operację. Na przykład, zadanie nie może modyfikować pamięci oznaczonej jako tylko do odczytu lub użytej jako segment kodu.
Ponadto w trybie chronionym procesor może zaadresować znacznie więcej pamięci. Procesor 80286 ma dostęp do 16M pamięci, a procesor 80386 potrafi zaadresować nawet 4 gigabajty, podczas gdy 8086 czy 8088 tylko 1M pamięci. Ponadto pamięć ta nie musi być fizycznie zainstalowana. Nie używany segment może być zapisany na dysk i zastąpiony takim, który jest właśnie potrzebny.

TABLICE DESKRYPTORÓW

W trybie chronionym programy nigdy nie operują na adresach fizycznych. Zamiast tego używają one selektorów, które wskazują położenie i długość każdego segmentu pamięci. Tablice deskryptorów są przechowywane w pamięci operacyjnej, dlatego też muszą i dla nich istnieć odpowiednie deskryptory. W procesorze mogą istnieć trzy tablice. Są to globalna tablica deskryptorów (GDT), lokalna tablica deskryptorów (LDT) i tablica deskryptorów przerwań (IDT).
Tablica GDT zawiera deskryptory, które mogą być wykorzystane przez dowolne z zadań w systemie (Rysunek 7-3). Znajdujące się w niej deskryptory opisują segmenty, w których

znajdują się tablice deskryptorów, pamięć video oraz powszechnie dostępne segmenty kodu i danych. Programy na najwyższym poziomie uprzywilejowania, korzystając z instrukcji SGDT, mogą pobrać dane o tej tablicy (długość i położenie). Jej argumentem jest wskaźnik na sześciobajtowy obszar pamięci, w którym zostaną one umieszczone. Pierwszym słowem jest liczba deskryptorów w tablicy. Następne dwa słowa stanowią adres bazowy. Jest on fizycznym adresem tablicy GDT w pamięci.

O zmianie czy to długości, czy położenia tablicy należy procesor poinformować. Służy temu instrukcja LGDT. Ma ona taki sam argument jak rozkaz SGDT, lecz powoduje pobranie danych spod wskazanego miejsca pamięci i umieszczenie ich w rejestrze GDTR procesora.

Tablica IDT zawiera deskryptory wszystkich segmentów, które zawierają procedury obsługi przerwań. Zastępuje ona wektor przerwań używany w trybie rzeczywistym. Dostęp do tej tablicy jest podobny jak dostep do tablicy GDT i jest realizowany przez instrukcje SIDT i LIDT.

Każde zadanie może mieć swoją oddzielną tablicę LDT. Znajdujące się w niej deskryptory opisują pamięć, która zawiera dane specyficzne dla tego zadania. Mimo że wykorzystywana może być tylko jedna, to w pamięci może istnieć wiele takich tablic. W systemach wielozadaniowych przełączenie zadania powoduje również zmianę tablicy LDT.

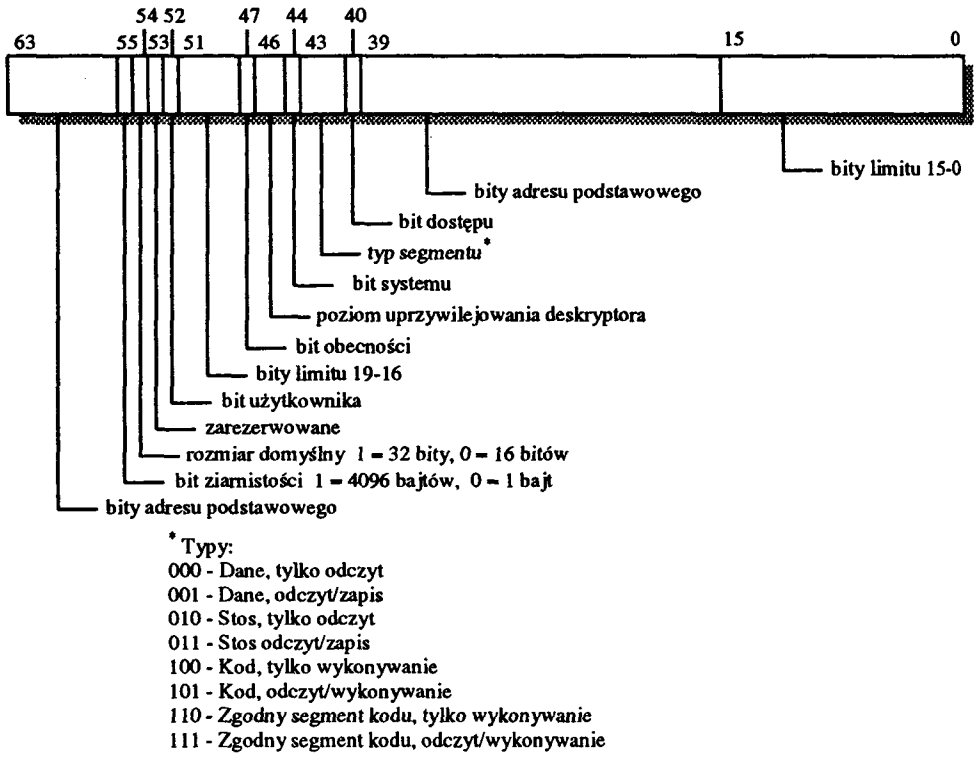
Położenie bieżącej tablicy LDT pozwalają ustalić i zapisać instrukcje SLDT i LLDT. Argumentem obu z nich jest selektor deskryptora z tablicy GDT.

SELEKTORY

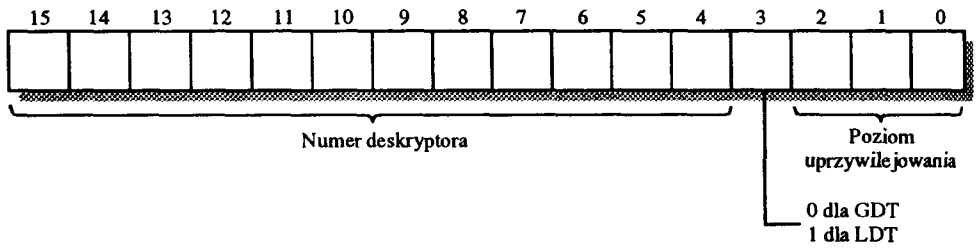
Zadania, które działają w trybie chronionym, używają tych samych rejestrów co zwykłe programy. Różna jest tylko zawartość tych rejestrów. W programach konwencjonalnych znajduje się tam adres paragrafu w pamięci. Po połączeniu numeru segmentu z przemieszczeniem powstaje adres fizyczny. W trybie chronionym natomiast rejestr segmentowy zawiera selektor. Selektor składa się z indeksu w tablicy deskryptorów, wyróżnika tablicy i żądanego poziomu uprzywilejowania. Rysunek 7-4 pokazuje położenie tych pól w selektorze.

Przed użyciem selektora musi on zostać załadowany do rejestru segmentowego. Przy każdym ładowaniu sprawdzane jest, czy podana wartość jest poprawnym selektorem i czy zadanie jest na wystarczająco wysokim poziomie uprzywilejowania. Jeśli rejestrem jest DS, ES, FS lub GS, to segment musi mieć zezwolenie dla odczytu. Dla rejestru SS segment musi mieć ustawione prawa pisania i czytania. Rejestr CS wymaga natomiast segmentu "wykonywalnego". Jeśli podany został błędny selektor, wówczas w procesorze 80386 powstanie błąd GP (General Protection fault). Skutkiem tego będzie wygenerowanie przerwania (w trybie chronionym zwanego wyjątkiem). Procedura obsługi tego przerwania może zawiesić wykonywanie zadania, które spowodowało ten błąd.

Błąd GP jest również generowany przy próbie zapisu do segmentu, który ma nadane prawa tylko do odczytu. Do sprawdzania praw dostępu danego segmentu służą trzy instrukcje. Pierwszą z nich jest LAR. Najpierw sprawdza ona poprawność podanego selektora i, jeśli jest on prawidłowy, zwraca prawa dostępu do wskazywanego przezeń segmentu. Pierwszym argumentem rozkazu jest 16- lub 32-bitowy rejestr, w którym zostaną umieszczone prawa dostępu selektora podanego w drugim. Jeśli podany selektor jest poprawny, to ustawiony zostanie znacznik zera; w przeciwnym razie znacznik ten zostanie wyzerowany, a prawa nie zostaną skopiowane. Jest to więc sposób na rozstrzygnięcie wątpliwości co do poprawności selektora przed zapisaniem go do rejestru segmentowego. Nieprawidłowy selektor



Rysunek 7-3. ***Format deskryptora.***



Rysunek 7-4. ***Pola selektora.***

nie wygeneruje bowiem błędu GP. Zwrócona dana będzie prawdopodobnie zawierać więcej informacji, niż jest to potrzebne. Jej format przedstawia tablica 7-1. Jeśli argument docelowy jest rejestrem 32-bitowym, wówczas cała ta informacja będzie do niego przesłana, w przeciwnym razie zostanie wykorzystane tylko młodsze 16 bitów.
Poniższy przykład pokazuje sposób wykorzystania instrukcji LAR do sprawdzenia poprawności selektora i jego praw dostępu:

```
; Sprawdzenie selektora
; AX = testowany selektor
        lar     ebx, ax       ; Pobierz prawa dostępu
                              ; selektora w AX
        jnz     BadSel        ; Jeśli znacznik zera nie
                              ; jest ustawiony, podano
                              ; błędny selektor
        and     ebx, 0E00h    ; Pobierz bity typu
        cmp     ebx, 200h     ; Sprawdź, czy dane można
                              ; zapisywać i odczytywać
        jne     BadSel        ; Jeśli nie, selektor jest błędny
        .                     ; OK, selektor może być użyty
        .
        .
BadSel:                       ; Operacje podejmowane,
                              ; gdy selektor był niepoprawny
        .
```

Tabela 7-1. Informacja zwracana przez instrukcję LAR.

Bity	Opis
31-24	Nie wykorzystane. Równe 0.
23	Ziarnistość. Gdy ustawiony, granica podana jest w bajtach, gdy wyzerowany w 4096 bajtowych blokach.
22	Domniemany rozmiar. Gdy ustawiony, argumenty są 32-bitowe, gdy wyzerowany 16-bitowe.
21	Zarezerwowany.
20	Nie zdefiniowany.
19-16	Nie wykorzystane.
15	Obecność. Gdy ustawiony, segment jest fizycznie załadowany do pamięci.
14-13	Poziom uprzywilejowania. 0 oznacza najbardziej, 3 najmniej uprzywilejowany.
12	System. Gdy ustawiony, segment nie jest segmentem systemowym.
11-9	Typ. Używane są następujące typy: 000 Dane, tylko do odczytu 001 Dane, do zapisu i odczytu 010 Stos, tylko do odczytu 011 Stos, do zapisu i odczytu 100 Kod, tylko wykonywanie 101 Kod, wykonywanie i odczytywanie 110 Zgodny segment kodu, tylko wykonywanie 111 Zgodny segment kodu, wykonywanie i odczytywanie
8	Dostęp. Bit jest ustawiony, gdy dana była z segmentu odczytana lub zapisana.
7-0	Nie wykorzystane. Ustawione na 0.

Instrukcja LAR nie jest zbyt wygodna, gdy zachodzi potrzeba sprawdzenia, czy pewien selektor jest poprawny oraz czy posiada prawo zapisu. W tym wypadku lepiej użyć rozkazu VERW. Ma on jeden argument, którym jest testowany selektor. Jeśli jest on poprawny i można do niego zapisywać, to znacznik zera jest ustawiany, inaczej zaś jest zerowany. Bardzo podobna instrukcja VERR pozwala na sprawdzenie, czy selektor ma prawo odczytu. Jej format i działanie jest podobne do poprzednio omówionego rozkazu. Przy użyciu tych instrukcji poprzedni przykład wygląda następująco:

```
; Sprawdź selektor, który znajduje się w AX, czy posiada
; on prawa pisania i czytania
        verr  ax          ; Czy ma prawo odczytu
        jnz   BadSel      ; Jeśli znacznik zera nie jest
                          ; ustawiony, to selektor nie ma
                          ; prawa odczytu
        verw  ax          ; Czy ma prawo zapisu
        jnz   BadSel      ; Jeśli znacznik zera nie jest
                          ; ustawiony, to selektor nie ma
                          ; prawa zapisu
        .                 ; OK, selektor może być użyty
        .
        .
BadSel:
        .                 ; Operacje podejmowane, gdy
        .                 ; selektor był niepoprawny
```

W trybie chronionym segmenty mogą być różnej długości. Dla procesora 80286 ich rozmiar może wahać się od 1 do 65,536 bajtów, a na procesorze 80386 jego długość może maksymalnie wynosić 4 gigabajty. Próba odczytu lub zapisu danej, która znajduje się poza segmentem, powoduje powstanie wyjątku GP. Do sprawdzenia długości segmentu służy instrukcja LSL o formacie:

```
lsl   reg16/32, regmem16/32
```

W pierwszym argumencie zostanie zwrócona długość segmentu, wskazywanego przez selektor umieszczony w drugim argumencie.
Ostatnie cztery instrukcje dostarczają wszelkiej niezbędnej o segmencie informacji, z wyjątkiem jego położenia w pamięci. Jednakże w trybie chronionym programy nie muszą znać rozmieszczenia segmentów.

POZIOMY UPRZYWILEJOWANIA

W trybie chronionym niektóre operacje mogą wykonywać tylko pewne programy. Na przykład, tablica GDT musi być aktualizowana, gdy zmieni się zapotrzebowanie programu na pamięć. Gdyby każdy program mógł ją modyfikować, to doprowadziłoby to do zniszczenia danych i chaosu w pamięci.
W trybie chronionym istnieją cztery różne poziomy uprzywilejowania. Poziom stawia ograniczenia na to, do jakich danych program ma dostęp, jakie podprogramy może wywołać czy też nawet jakich instrukcji wolno mu użyć.

W tablicy 7-2 przedstawione zostały instrukcje, które mogą wykonać tylko programy na najwyższym poziomie uprzywilejowania (poziom 0). Jeśli program mniej uprzywilejowany będzie próbował użyć tych rozkazów, wówczas spowoduje to wygenerowanie wyjątku.

Tablica 7-2. Instrukcje uprzywilejowane.

ARPL	Zmiana pola DPL selektora
CLTS	Wyzerowanie znacznika przełączenia zadania
HLT	Zatrzymanie procesora
LGDT, LIDT, LLDT	Ładowanie tablic deskryptorów
LMSW	Ładowanie słowa stanu procesora
LTR	Ładowanie rejestru zadania
MOV CRn/reg, reg/CRn	Ładowanie rejestrów sterujących
MOV DRn/reg, reg/DRn	Ładowanie rejestrów uruchamiania
MOV TRn/reg, reg/TRn	Ładowanie rejestrów testowania
SGDT, SIDT, SLDT	Zapisanie tablic deskryptorów

Instrukcje wejścia/wyjścia oraz instrukcje, które operują na znaczniku zezwolenia na przerwanie, mogą być wykonywane tylko przez programy, których poziom uprzywilejowania jest równy lub wyższy niż przywilej operacji I/O (IOPL). IOPL jest częścią rozszerzonego rejestru znaczników. Rejestr ten zawiera oprócz zwykłych znaczników trzy nowe flagi oraz dwa bity na IOPL. Ponieważ procedura **GoProtect** ustawia poziom uprzywilejowania na 0 (poziom najwyższy), to mogą być w niej używane instrukcje wejścia/wyjścia niezależnie od wartości IOPL. Kontrolę nad dostępem do wejścia/wyjścia można zapewnić, poprzez ustawienie poziomów zadań różnych od zera oraz wartości IOPL mniejszej od wszystkich tych poziomów. Jeśli teraz zadanie będzie próbowało użyć instrukcji wejścia/wyjścia, wówczas zostanie wygenerowane przerwanie. Jeśli zaś przerwania są zablokowane, spowoduje to wyzerowanie procesora.

Każdy segment posiada swój poziom uprzywilejowania. Zapamiętany on jest w polu DPL (Descriptor Privilege Level) deskryptora. Poziom uprzywilejowania segmentu, który jest właśnie wykonywany, jest oznaczany skrótem CPL (Current Privilege Level). Zapamiętany jest on na dwóch młodszych bitach rejestru CS:

```
mov   ax, cs    ; Pobierz selektor segmentu kodu
and   ax, 3     ; Istotne są tylko dwa najmłodsze bity
```

Wartość CPL ogranicza zestaw operacji dostępnych w danej chwili. Program może mieć dostęp do danych o poziomach niższych lub równych jego poziomowi. Dozwolone są tylko wywołania funkcji i skoki do segmentów o identycznych poziomach. Również stos musi mieć poziom uprzywilejowania równy CPL.

Oznacza to więc, że zadanie, które posiada trzeci poziom uprzywilejowania nie będzie mogło modyfikować tablicy GDT, która ma najprawdopodobniej poziom zerowy. Zadanie, które ładuje inne programy do pamięci i działa na poziomie 1, również nie może zmieniać tablicy GDT, może natomiast w obszarze zadania o poziomie 3 umieścić nową aplikację.

Program, działający na poziomie 1, nie tylko nie ma dostępu do danych poziomu 0, ale nie może również wykonać skoków czy wywołać procedur umieszczonych w tym obszarze. Co więcej, nie może on nawet wywołać żadnego podprogramu z poziomu 3.
Rozwiązaniem problemu wywoływania procedur, które znajdują się na innych poziomach uprzywilejowania, jest zastosowanie bramek wywołań (gate). Jest to specjalny deskryptor, zawierający selektor oraz przemieszczenie funkcji. W ten sposób programy mogą wywoływać tylko określone procedury z innego poziomu.
Kolejny przykład stanowi procedurę, która zmienia deskryptor w tablicy GDT na bramkę wywołania. Może być ona użyta w programie, który przełącza procesor w tryb chroniony. Powinna ona być wywoływana w czasie ustawiania wartości innych deskryptorów (jeszcze przed przełączeniem w tryb chroniony). Należy tylko pamiętać o dodaniu miejsca na deskryptor bramki wywołania w zmiennej GblDT i zmienić rozmiar w GblDscTbl.

```
; Stworzenie deskryptora bramki wywołania
; SI = Przemieszczenie modyfikowanego deskryptora
; AL = Poziom uprzywilejowania wywoływanej funkcji
; BX = Przemieszczenie funkcji
; CX = Selektor segmentu zawierającego tą funkcję
MakeCG   PROC
         mov   WORD PTR [si], bx      ; Przemieszczenie
                                      ; funkcji
         mov   WORD PTR [si+2], cx    ; Selektor
         mov   BYTE PTR [si+4], 0
         shl   al, 5
         or    al, 10001100b          ; Bity identyfikujące
                                      ; bramkę wywołania
         sub   ah, ah
         mov   WORD PTR [si+5], ax
         mov   WORD PTR [si+7], 0
         ret
MakeCG   ENDP
```

Funkcje mniej uprzywilejowane mogą, za pośrednictwem tak stworzonej bramki, wywoływać funkcje bardziej uprzywilejowane. Argument instrukcji CALL musi wskazywać na deskryptor bramki wywołania:

```
call   FAR PTR 0010:0000
```

Segmentowa część adresu (0010) wskazuje bramkę wywołania. Procesor ignoruje podawane przemieszczenie. Zamiast niego bowiem używa on przemieszczenia zawartego w deskryptorze bramki.
Niektóre procedury muszą być dostępne dla wielu programów, które działają na różnych poziomach uprzywilejowania. Na przykład, program, ładujący inne zadania do pamięci czy program wymiany pamięci, musi mieć dostęp do procedur, które zapisują i odczytują dane z dysku. Zamiast bramek wywołań lepiej jest w tym wypadku użyć zgodnego segmentu kodu. Zgodne segmenty kodu mają ten poziom uprzywilejowania, który miał podprogram wywołujący. Segment jest zgodnym segmentem kodu, jeśli są odpowiednio ustawione wartości bitów w polu typu deskryptora.

Programy uprzywilejowane mogą zmieniać poziom uprzywilejowania segmentu, modyfikując pole DPL w tablicy deskryptorów. Pozostałe programy mogą tylko zwiększać poziom uprzywilejowania segmentu. Służy do tego instrukcja ARPL o formacie:

```
arpl   selektor, reg
```

Selektor segmentu może być zawarty w rejestrze lub w pamięci. W rejestrze zawarty jest żądany poziom uprzywilejowania. Jeśli selektor jest mniej uprzywilejowany, to nadawany mu jest nowy poziom, a znacznik zera jest ustawiany. W przeciwnym razie znacznik jest zerowany.

STRONICOWANIE

W systemie MS-DOS, na komputerach kompatybilnych z IBM PC, pamięć video oraz BIOS znajdują się w obszarze pamięci o adresach od **0A0000h** do **0FFFFFh**. W systemach z procesorami 8088 i 8086 jest to końcowy obszar adresowalnej pamięci. Przy większych przestrzeniach adresowych procesorów 80286 i 80386 obszar ten wypada w środku dostępnej pamięci. Programy muszą brać pod uwagę tą nieciągłość pamięci.
Problem ten został w procesorze 80386 pokonany dzięki zastosowaniu stronicowania. Pamięć fizyczna jest podzielona na strony, które mogą być dowolnie rozmieszczone w wirtualnej przestrzeni adresowej. Programy widzą pamięć jako zbiór kolejno numerowanych komórek, choć w rzeczywistości może być ona bardzo rozproszona. Gdy program odwołuje się do pamięci, to układ zarządzania pamięcią (MMU) w procesorze przekształca podany adres wirtualny w adres fizyczny i wykonuje żądaną operację.
Jeżeli podanemu adresowi logicznemu nie odpowiada żaden adres fizyczny, wówczas MMU wywołuje program, który zapisuje stronę pamięci fizycznej na dysk, a odczytuje stronę poszukiwaną po czym dokonuje translacji adresów. Oznacza to, że programista może używać całej czterogigabajtowej przestrzeni adresowej procesora 80386 niezależnie od rozmiaru fizycznie zainstalowanej pamięci operacyjnej. Jedyną różnicą będzie tylko zwolnienie działania, które wynikają z konieczności wymiany stron pamięci.
Stronicowanie odbywa się poprzez specjalne tablice, które znajdujące się w pamięci oraz przez rejestry procesora. Jednakże szczegóły jego implementacji wykraczają poza ramy tej książki. Zalety stronicowania można wykorzystać, stosując jeden z kilku programów implementujących stronicowanie. Pod kontrolą systemem MS-DOS programista ma do dyspozycji oprogramowanie, które odpowiadają standardowi wypracowanemu przez firmy Lotus, Intel i Microsoft (LIM).

Wielozadaniowość

Na początku rozdziału przedstawiony został przykład, który pokazuje, w jaki sposób wielozadaniowość może zwiększyć wydajność komputera poprzez umożliwienie wykonywania się innym zadaniom, gdy reszta czeka na zakończenie się operacji wejścia/wyjścia. Gdy komputer przełącza się z jednego zadania na inne, to musi zapamiętać informację o przerywanym zadaniu i pobrać informację o zadaniu wznawianym. Informacja ta zawiera wartości wszystkich rejestrów procesora, stos, tablicę LDT i adres instrukcji, od której należy rozpocząć wykonywanie zadania. Jest ta tzw. kontekst zadania.

Przełączanie zadań

Pierwszym krokiem, podejmowanym przy przełączaniu zadań, jest zapamiętanie kontekstu bieżącego zadania. Następnie pobierany jest kontekst nowego zadania i na koniec jest ono uruchamiane. Procesor 80386 używa specjalnego deskryptora, który wskazuje segment stanu każdego zadania (TSS). TSS zawiera kontekst związanego z nim zadania. Inny deskryptor zwany bramką zadania używany jest w procesie przełączenia. Działanie tej bramki podobne jest do działania bramki wywołania.

Rejestr, zwany rejestrem zadania (TR), informuje procesor 80386, w którym segmencie TSS zapisać kontekst aktualnego zadania. Po przełączeniu rejestr TR otrzymuje wartość selektora zadania, które to zadanie zostanie uruchomione. Służy do tego instrukcja LTR (Load Task Register). Jedynym jej argumentem jest rejestr lub komórka pamięci, która zawiera selektor bieżącego zadania. Do sprawdzenia, w którym segmencie TSS zostanie zapisany kontekst bieżącego zadania, służy rozkaz STR. Jednakże nie można użyć zwróconego selektora do dostępu do danych znajdujących się w segmencie TSS.

Trzy funkcje przedstawione w poniższym przykładzie służą do inicjacji mechanizmu wielozadaniowości, do stworzenia nowego zadnia i dokonania przełączenia pomiędzy zadaniami. Przyjmują one, że tablica GDT zawiera deskryptor segmentu, w którym będą zapisywane dane o zadaniu. Selektor tego segmentu ma wartość 28h; następne 64 deskryptory muszą zostać zarezerwowane dla innych możliwych zadań.

```
          .MODEL   SMALL
          PUBLIC   InitTask                  ; Stwórz
                                             ; pierwsze
                                             ; zadanie
          PUBLIC   NewTask                   ; Stwórz nowe
                                             ; zadanie
          PUBLIC   SwitchTask                ; Przełącz
                                             ; zadania
          .386
TaskSeg   SEGMENT   WORD PRIVATE USE16 'TASK'
TaskCnt   DW   0                             ; Liczba zadań
CurTask   DW   0                             ; Zadanie,
                                             ; które jest
                                             ; aktualnie
                                             ; wykonywane
TskData   DB   104 * 64 dup (?)              ; Miejsce na
                                             ; dane o
                                             ; stanie zadań
TskJmp    DD   0
TaskSeg   ENDS
          .CODE
; Inicjacja mechanizmu przełączania zadań
InitTask  PROC
          mov   ax, 8h                       ; Selektor
                                             ; tablicy GDT
          mov   ds, ax                       ; Załaduj go
```

```
                                        ; do DS
        mov   si, 30h                   ; Przemieszcz-
                                        ; enie
                                        ; deskryptora
                                        ; pierwszego
                                        ; zadania
        mov   ax, 103h                  ; Rozmiar
                                        ; segmentu TSS
        mov   WORD PTR [si], ax
        mov   ax, SEG TaskSeg           ; Numer
                                        ; segmentu TSS
        mov   bl, ah
        shl   ax, 4
        shr   bl, 4
        add   ax, OFFSET TskData        ; Dodanie
                                        ; przemieszcz-
                                        ; enia danych
        adc   bl, 0                     ; zadania
        mov   WORD PTR [si+2], ax
        mov   BYTE PTR [si+4], bl
        mov   WORD PTR [si+5], 011101001b ; Znaczniki
        mov   BYTE PTR [si+7], 0
        mov   ax, 28h                   ; Selektor
                                        ; TaskSeg
        mov   ds, ax
        inc   TaskCnt
        mov   ax, 30h                   ; Pierwszy
                                        ; deskryptor
                                        ; TSS
        ltr   ax                        ; Załaduj
                                        ; rejestr
                                        ; zadania
        ret
InitTask ENDP
; Umieść nowe zadanie w tablicy
; AX = Przemieszczenie funkcji dla zadania
; BX = Selektor tej funkcji
; CX = Selektor stosu
NewTask PROC
        push  ax                        ; Zachowaj
                                        ; parametry
        push  bx
        push  cx
        mov   ax, 8h                    ; Selektor
                                        ; tablicy GDT
        mov   ds, ax                    ; Załaduj go
```

```
                                        ; do DS
mov   si, TaskCnt                       ; Numer
                                        ; następnego
                                        ; zadania
shl   si, 3                             ; pomnożony
                                        ; przez rozmiar
                                        ; deskryptora
add   si, 30h                           ; Przemieszcz-
                                        ; enie
                                        ; pierwszego
                                        ; deskryptora
                                        ; TSS
mov   ax, 103h                          ; Rozmiar
                                        ; segmentu TSS
mov   WORD PTR [si], ax
mov   ax, SEG TaskSeg                   ; Numer
                                        ; segmentu TSS
mov   bl, ah
shl   ax, 4
shr   bl, 4
mov   cx, TaskCnt
shl   cx, 3
add   cx, OFFSET TskData
adc   bl, 0
add   ax, cx                            ; Dodanie
                                        ; przemieszcz-
                                        ; enia danych
adc   bl, 0                             ; zadania
mov   WORD PTR [si+2], ax
mov   BYTE PTR [si+4], bl
mov   WORD PTR [si+5], 011101001b ; Znaczniki
mov   BYTE PTR [si+7], 0
mov   ax, 28h                           ; Selektor
                                        ; TaskSeg
mov   ds, ax
mov   si, TaskCnt                       ; Pobierz
                                        ; numer zadania
shl   si, 3
mov   ax, si
shl   si, 2
add   ax, si
shl   si, 1
add   si, ax                            ; AX razy 104
add   si, OFFSET TskData
sub   eax, eax                          ; Wyzeruj
                                        ; starsze słowo
```

```
        pop   cx                          ; Odtwórz
                                          ; parametry
        pop   bx
        pop   ax
        mov   [si+20h], eax               ; Zapisz EIP
        mov   DWORD PTR [si+38h], 1024    ; Zapisz ESP
        mov   [si+4Ch], ax
        mov   [si+50h], cx
        inc   TaskCnt
        ret
NewTask ENDP
; Przełączenie na wykonywanie następnego zadania.
; Przełączenie wywołuje pojawienie się przerwania
; zegarowego lub też zadanie, które na pewien czas nie
; potrzebuje czasu procesora
SwitchTask PROC
        mov   ax, 28h                     ; Selektor
                                          ; TaskSeg
        mov   ds, ax
        mov   ax, CurTask
        inc   ax                          ; Zmień na
                                          ; następne
                                          ; zadanie
        cmp   ax, TskCnt                  ; Sprawdź, czy
                                          ; nie jest to
                                          ; ostatnie
        jb    Sk1                         ; zadanie
                                          ; z listy
        mov   ax, 1
Sk1:
        mov   CurTask, ax
        shl   ax, 3                       ; Pomnóż
                                          ; przez 8
        add   ax, 30h                     ; Pierwszy
                                          ; selektor
        mov   WORD PTR TskJmp+2, ax
        jmp   DWORD [TskJmp]              ; Przełącz
                                          ; zadania
SwitchTask ENDP
```

TESTOWANIE WYSTĄPIENIA PRZEŁĄCZENIA ZADANIA

Dzięki zachowywaniu i odtwarzaniu kontekstu zadania, nie musi ono nawet wiedzieć o pozostałych zadaniach, działających z nim równolegle. Jednakże czasami istnieje potrzeba sprawdzenia, czy w systemie wykonują się jeszcze inne zadania. Na przykład, program, używający wspólnych segmentów danych, wie, że dane te pozostaną tak długo nie zmienione, jak długo nie pojawi się inne zadanie. W słowie stanu procesora MSW istnieje bit (bit 3),

który jest ustawiany w momencie przełączenia zadań. Instrukcja CLTS (Clear Task Switched) powoduje jego wyzerowanie. Rozkaz SMSW pozwala pobrać słowo MSW, co umożliwia testowanie tego bitu. Program, który chce sprawdzić, czy został przerwany, może wyglądać następująco:

```
          clts              ; Wyzeruj bit przełączania zadań
          .                 ; Wykonaj inne operacje
          .
          .
          smsw  ax          ; Pobierz słowo MSW
          test  ax, 4       ; Sprawdź, czy pojawił się
                            ; sygnał przełączenia zadania
          jz    noswitch
          .                 ; Upewnij się, że żadne inne
                            ; zadanie nie przerwało nas
          .
          .
noswitch:
          .                 ; Kontynuuj operacje
          .
          .
```

Podsumowanie

Zastosowanie koprocesora i wielozadaniowości pozwala na zwiększenie mocy obliczeniowej komputera. Wielozadaniowość umożliwia lepsze wykorzystanie procesora głównego, a zastosowanie koprocesora pozwala na podział zadań pomiędzy nimi. Procesory 80x86 posiadają pewne udogodnienia związane z tymi zagadnieniami.

Pojawienie się nowego procesora sprawia, że komputery stają się coraz bardziej wyrafinowane. Tryb chroniony dostarcza cech dostępnych wcześniej tylko na dużych maszynach. Dzięki stronicowaniu można dowolnie skonfigurować pamięć. Przed pojawieniem się procesora 80386 operacja ta była możliwa tylko na poziomie sprzętu.

Dla wielu programów koprocesory, tryb chroniony czy stronicowanie są tylko rzeczami ubocznymi. Bez żadnych zmian mogą one zostać uruchomione na dowolnym procesorze rodziny 80x86. Ponieważ jednak współczesne programy stają się coraz bardziej złożone, na pewno zostaną w nich zastosowane zaawansowane możliwości pracy wielozadaniowej oferowane przez nowe procesory.

Adjust Requested Privilege Level 88/86 286 386
● ●

ARPL

ZASTOSOWANIE

Zmiana poziomu uprzywilejowania selektora. Jeśli poziom uprzywilejowania selektora jest większy od wartości rejestru, to zmiana jest wykonywana. Jeśli zaś poziom uprzywilejowania selektora jest mniejszy lub równy wartości rejestru, to operacja ta nie jest przeprowadzana.

SKŁADNIA

`ARPL selektor, reg` *Selektor może znajdować się w rejestrze lub pamięci. Drugi argument zawiera nowy poziom uprzywilejowania selektora.*

ZNACZNIKI

Jeśli selektor został poprawiony, to znacznik zera jest ustawiany. Wszystkie inne znaczniki pozostają bez zmian.

KOMENTARZ

Instrukcja ta jest często używana w procedurach o wysokim poziomie ochrony przy przekazywaniu dalekich wskaźników. Jednakże dla funkcji tych nie jest ważny sam fakt zmiany poziomu uprzywilejowania, ale możliwość jego sprawdzenia. Jeśli selektor był poprawny dla podprogramu mniej uprzywilejowanego, to będzie również poprawny dla bardziej uprzywilejowanego.

Program może odwołać się do pamięci, do której nie ma dostępu, poprzez zlecenie systemowi operacyjnemu skopiowania danych do lub z chronionego obszaru. Ponieważ system operacyjny działa na najbardziej uprzywilejowanym poziomie, może on przesyłać dane pomiędzy dowolnymi selektorami. Powodowałoby to jednak naruszenie ochrony pamięci. W przykładzie przedstawiono sposób, w jaki system operacyjny może wykryć tego typu sytuacje.

PATRZ

LAR	*Ładowanie praw dostępu*
LSL	*Ładowanie rozmiaru segmentu*
VERR/VERW	*Testowanie prawa do odczytu i zapisu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	01100011 11,reg,r/m	arpl ax,bx	-	10	20
pamięć, reg	01100011 mod,reg,r/m przemieszczenie	arpl select,ax	-	11	21

CLC

PRZYKŁAD

Przykład pokazuje, w jaki sposób system operacyjny może wykryć próbę dostępu do uprzywilejowanego segmentu. Poziom uprzywilejowania wywołującej procedury znajduje się w rejestrze CS, który został odłożony na stos. Podprogram wywoływany może pobrać tę daną spod adresu sp+2 dla segmentów 16-bitowych i sp+4 dla segmentów 32-bitowych.

```
; Funkcja sprawdza poziom uprzywilejowania funkcji, która
; ją wywołuje względem selektora przekazanego
; w rejestrze BX.
          .386p
          .CODE
CopyMem   PROC FAR
          arpl    [sp+4], bx    ; Testuj selektor
          jz      error         ; Skok, gdy należy zmienić
                                ; rejestr CS
          .
          .
          .
error:
          ret
CopyMem   ENDP
```

Clear Carry Flag

88/86	286	386
●	●	●

CLC

ZASTOSOWANIE

Wyzerowanie znacznika przeniesienia.

SKŁADNIA

CLC *Wyzeruj znacznik przeniesienia*

ZNACZNIKI

Znacznik przeniesienia jest zerowany. Wszystkie inne znaczniki pozostają bez zmian.

KOMENTARZ

Zwykle znacznik przeniesienia jest ustawiany zgodnie z wynikiem operacji arytmetycznych lub logicznych. Instrukcja ta wraz z rozkazami STC i CMC pozwala bezpośrednio sterować wartością znacznika. Wiele funkcji wykorzystuje ten sposób do poinformowania programu, który je wywołał, o zaistnieniu błędu.

PATRZ

CMC *Zanegowanie wartości znacznika przeniesienia*

STC *Ustawienie znacznika przeniesienia*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11111000	clc	2	2	2

PRZYKŁAD

Poniższa funkcja pobiera numer wersji systemu MS-DOS. Jeśli nie jest to wersja 3.x, funkcja kończy się z ustawionym znacznikiem przeniesienia.

```
            .CODE
CheckVer    PROC
            mov   ah, 30h   ; Funkcja, która zwraca numer
                            ; wersji systemu
            int   21h
            cmp   al, 3     ; Sprawdź główną część numeru
            clc             ; Wyzeruj znacznik przeniesienia
            je    Done      ; Jeśli wersja 3.x, powrót
            stc             ; w przeciwnym razie ustaw
                            ; znacznik
Done:
            ret
CheckVer    ENDP
```

Clear the Direction Flag

88/86	286	386
●	●	●

CLD

ZASTOSOWANIE

Wyzerowanie znacznika kierunku.

SKŁADNIA

CLD *Wyzeruj znacznik kierunku.*

ZNACZNIKI

Znacznik kierunku jest zerowany.

KOMENTARZ

Gdy znacznik kierunku jest ustawiony, to operacje łańcuchowe powodują zmniejszanie rejestru indeksowego, w przeciwnym razie jest on zwiększany.

CLI

PATRZ

STOS/MOVS/SCANS/INS/OUTS/CMPS *Operacje łańcuchowe*
STD *Ustaw znacznik kierunku*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślny	11111100	cld	2	2	2

PRZYKŁAD

Patrz przykład do instrukcji CMPS i SCANS w rozdziale 5.

Clear Interrupt Flag

88/86	286	386
●	●	●

CLI

ZASTOSOWANIE

Wyzerowanie znacznika zezwolenia na przerwanie.

SKŁADNIA

```
CLI
```

KOMENTARZ

Wyzerowanie znacznika zezwolenia na przerwanie blokuje przyjmowanie przerwań przez procesor. Jedynym wyjątkiem jest przerwanie niemaskowalne i przerwania programowe. Jest to ważne w wypadku zmian dokonywanych w tablicy przerwań lub też przy wykonywaniu funkcji, która nie może zostać przerwana. Jeśli jest to tylko możliwe, przerwania powinny zostać włączone. Służy do tego instrukcja STI.

PATRZ

STI

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
Domyślny	11111010	cli	2	3	3

PRZYKŁAD

Przyjęcie przerwania powoduje odłożenie pewnych wartości na stos. Jeśli zaszła potrzeba zmiany położenia stosu, to nie można więc pozwolić procesorowi na przyjęcie przerwania w środku tej operacji. Dla większości procesorów 80x86 modyfikacja rejestru segmentowego powoduje zablokowanie przerwań na czas wykonania następnej instrukcji. Niestety kilka

starszych wersji procesora 8088 jest tej cechy pozbawionych. Jeśli twój program ma działać również i na tych procesorach, to bezpiecznym rozwiązaniem jest wyłączenie przerwań na czas przenosin stosu.

```
ASTACK      SEGMENT  STACK  'STACK'
            DB   1024 dup (?)        ; Przydzielenie miejsca
                                     ; na stos
TOP         LABEL   BYTE             ; Wskaźnik szczytu stosu
ASTACK      ENDS
            .CODE
NewStack    PROC
            cli                      ; Wyłącz przerwania
            mov   ss, SEG ASTACK     ; Zmień segment stosu
            mov   sp, OFFSET TOP     ; i wskaźnik stosu
            sti                      ; Włącz przerwania
            ret
NewStack    ENDP
```

Clear Task Switched Flag — 88/86 286 386 ● ●

CLTS

ZASTOSOWANIE

Wyzerowanie znacznika przełączenia zadania w słowie stanu procesora MSW.

SKŁADNIA

CLTS

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja ta może być użyta tylko w trybie chronionym. Używana jest ona przy sprawdzaniu, czy pomiędzy pewnymi dwoma punktami kodu zadanie zostało przełączone. Dzięki niej można się więc dowiedzieć, czy jakiś wspólny zasób mógł zostać zmieniony.

PATRZ

LMSW, SMSW — *Załaduj i zapisz słowo stanu procesora MSW*

CMC

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	00001111 00000110	clts	-	2	5

PRZYKŁAD

Instrukcje CLTS i LMSW mogą być użyte do sprawdzenia, czy inne zadanie zmieniło pewne parametry.

```
; Wyzeruj znacznik zadania, a następnie ustaw parametry.
; Przed ich użyciem sprawdź stan tego znacznika.
        .386p
        .CODE
Funct   PROC FAR
setup:
        clts            ; Wyzeruj znacznik przełączenia
                        ; zadania
        .               ; Ustaw parametry
        .
        .
useit:
        lmsw  ax        ; Pobierz do AX słowo MSW
        test  ax, 8     ; Sprawdź stan znacznika
        jnz   setup     ; Jeśli jest on ustawiony, powtórz
        .               ; w przeciwnym razie kontynuuj
        .
        .
Funct   ENDP
```

Complement the Carry

88/86	286	386
●	●	●

CMC

ZASTOSOWANIE

Zanegowanie wartości znacznika przeniesienia.

SKŁADNIA

CMC

ZNACZNIKI

Stan znacznika przeniesienia zmienia się na przeciwny.

KOMENTARZ

Instrukcja ta przydaje się, gdy znacznik przeniesienia jest wykorzystywany jako znacznik błędu. Czasami zdarza się, że jego stan jest przeciwny niż żądany do zasygnalizowania błędu. Użyteczna jest wtedy instrukcja CMC.

PATRZ

CLC	*Wyzeruj znacznik przeniesienia*
STC	*Ustaw znacznik przeniesienia*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11110101	cmc	2	2	2

PRZYKŁAD

Poniższy przykład usuwa wszystkie pliki o nazwach **fileXX.dat**, gdzie **XX** oznacza liczbę z przedziału od 00 do 99. Ponadto zlicza on skasowane pliki.

```
            .MODEL SMALL
            .DATA
DeleteCnt   DW    0
FileName    DB    "FILE00.DAT",0
            .CODE
DelFiles    PROC
            mov   dx, OFFSET FileName
            mov   ah, 41h
            int   21h                     ; Usuń plik
            cmc                           ; Zaneguj
                                          ; znacznik
                                          ; przeniesienia
            adc   DeleteCnt, 0            ; Dodaj
                                          ; przeniesienie
                                          ; do licznika
            mov   bx, OFFSET FileName + 5
            inc   BYTE PTR [bx]           ; Następna
                                          ; liczba
            cmp   BYTE PTR [bx], '9'      ; Porównaj z 9
            jle   DelFiles                ; Kasuj następny
                                          ; plik
            mov   BYTE PTR [bx], '0'      ; Ponownie
                                          ; wpisz zero
            dec   bx                  ; Będzie modyfikowana
                                      ; liczba dziesiątek
            inc   BYTE PTR [bx]       ; Zwiększ liczbę
                                      ; dziesiątek
            cmp   BYTE PTR [bx], '9'  ; Czy koniec
```

```
            jle   DelFiles          ; Kasuj następny plik
            ret
DelFiles    ENDP
```

Escape

88/86	286	386
●	●	●

ESC

ZASTOSOWANIE

Wysłanie na magistralę danych rozkazu dla koprocesora.

SKŁADNIA

```
ESC  instrukcja, pamięć_lub_rejestr
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Koprocesory przejmują wykonywanie niektórych zadań procesora głównego. Najczęściej stosowane są koprocesory arytmetyczne 8087, 80287 i 80387. Komunikacja z tymi układami odbywa się za pomocą instrukcji ESC. Nie jest jednak ona często stosowana, gdyż asembler zapewnia istnienie mnemoników dla każdego rozkazu koprocesora.

PATRZ

WAIT, LOCK *Inne instrukcje używane z koprocesorem*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
rejestr	11011TTT 11,LLL,r/m	esc 5,al	2	9-20	różnie
pamięć	11011TTT mod,LLL,r/m	esc 29,[bx]	12/8+EA	9-20	różnie

Uwaga: TTT stanowi trzy starsze, LLL trzy młodsze bity instrukcji dla koprocesora.

PRZYKŁAD

Użycie koprocesora może znacznie przyspieszyć wykonanie programu. Przyspieszenie to wynika nie tylko stąd, że odpada konieczność wykonania długich procedur implementujących operacje zmiennoprzecinkowe, ale także stąd, że w czasie wykonywania przez koprocesor obliczeń procesor główny może zająć się innymi operacjami.
Poniższy fragment kodu używa instrukcji ESC, aby dodać dwie liczby zmiennoprzecinkowe.

```
            .MODEL   SMALL
            .DATA
real1       DQ   1.232
real2       DQ   4.5
            .CODE
; Dodaj liczby rzeczywiste zapisane w zmiennych
; real1 i real2
AddFloat    PROC
            wait                ; Upewnienie się, że
                                ; koprocesor jest gotowy
            esc   28h, real1    ; Instrukcja FLD
            wait                ; Czekaj na jej zakończenie
            esc   20h, real2    ; Instrukcja FADD
            wait                ; Czekaj na jej zakończenie
            esc   2Ah, real1    ; Instrukcja FST
            ret
AddFloat    ENDP
```

Halt

88/86	286	386
●	●	●

HLT

ZASTOSOWANIE

Zatrzymanie procesora do chwili pojawienia się przerwania.

SKŁADNIA

HLT

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcja ta nie jest zbyt często stosowana. Zwykle bowiem istnieją zawsze do wykonania jakieś operacje, choćby nawet tylko testowanie wejścia/wyjścia. Ponadto jest tak wiele przerwań, że procesor rzadko kiedy długo pozostawałby w stanie zatrzymania.
W trybie chronionym instrukcja HLT jest rozkazem uprzywilejowanym.

PATRZ

WAIT — *Oczekiwanie na gotowość koprocesora*

HLT

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11110100	hlt	2	2	5

PRZYKŁAD

Instrukcji HLT można użyć do zatrzymania procesora w czasie oczekiwania na wciśnięcie klawisza. Ponieważ naciśnięcie klawisza powoduje wygenerowanie przerwania, to procesor zostanie wyprowadzony z tego stanu. Nie wszystkie klawisze mogą być używane (np. klawisz Shift). Ponadto inne urządzenia również wysyłają przerwania. Dlatego też w poniższym podprogramie po każdym pojawieniu się przerwania sprawdzany jest stan klawiatury.

```
; Zatrzymaj procesor czekając na znak z klawiatury
            .CODE
KeyIn       PROC FAR
            mov   ax, 40h        ; Obszar danych BIOS-u
            mov   es, ax
TestKey:
            mov   ax, es:[1Ch]   ; Adres końca bufora
                                 ; klawiatury
            cmp   es:[1Ah], ax   ; Porównanie z adresem
                                 ; początku
            jne   GotOne         ; Jeśli nie są równe,
                                 ; odczytany został znak
            hlt                  ; Czekaj
            jmp   TestKey        ; Sprawdź, czy jest znak
GotOne:
            mov   bx, es:[1Ah]   ; Pobierz do BX adres
                                 ; początku bufora
            mov   ax, es:[bx]    ; Pobierz znak
            inc   bx
            cmp   bx, 34h        ; Czy to koniec bufora
            jne   Skip
            mov   bx, 1Eh        ; Początek bufora
Skip:
            mov   es:[1Ah], bx   ; Zapisz nowy adres
                                 ; początku bufora
      ret
KeyIn       ENDP
```

Load AH from the Flags Byte

88/86	286	386
●	●	●

LAHF

ZASTOSOWANIE

Pobranie do rejestru AH młodszego bajtu rejestru znaczników.

SKŁADNIA

LAHF

ZNACZNIKI

Bez zmian.

KOMENTARZ

Znaczniki, które mogą być pobrane tą instrukcją to: znaczniki znaku, zera, parzystości, przeniesienia i przeniesienia pomocniczego.

PATRZ

SAHF	*Przesłanie rejestru AH do rejestru znaczników*
PUSHF, POPF	*Pobranie i zapisanie na stos rejestru znaczników*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	10011111	lahf	4	2	2

PRZYKŁAD

Poniższa procedura pobiera znaczniki i przechowuje je w zmiennej **OldFlags**.

```
            .MODEL    SMALL
            .DATA
OldFlags    DB    ?
            .CODE
Function    PROC
       lahf                    ; Pobierz wartości
                               ; znaczników do rejestru AH
       mov    OldFlags, ah     ; Zapisz je w zmiennej OldFlags
       .                       ; Można teraz wykonać kod
       .                       ; zmieniający stan znaczników
       mov    ah, OldFlags     ; Odtwórz zapamiętane znaczniki
       sahf
       ret
Function    ENDP
```

LAR

Load Access Rights 88/86 286 386
● ●

LAR

ZASTOSOWANIE

Pobranie praw dostępu selektora w trybie chronionym.

SKŁADNIA

```
LAR  rejestr, selektor
```

ZNACZNIKI

Znacznik zera jest ustawiany, gdy podany selektor jest poprawny i widoczny z bieżącego poziomu uprzywilejowania. Reszta znaczników pozostaje bez zmian.

KOMENTARZ

Instrukcja LAR pozwala na pobranie informacji o podanym selektorze. Zwracana dana ma następujące składowe:

Bit ziarnistości,
Bit rozmiaru,
Bit zarezerwowany przez firmę Intel
Bit zdefiniowany przez użytkownika
Bit obecności
Poziom uprzywilejowania deskryptora
Bit systemowy
Bit typu deskryptora
Bit wykorzystania

PATRZ

VERR, VERW *Testowanie praw zapisu i odczytu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	00001111 00000010 mod,reg,r/m	lar ax,bx	-	14	15
reg, mem	00001111 00000010 mod,reg,r/m przemieszczenie		-	16	16

PRZYKŁAD

Podprogram sprawdza standardowy rozmiar danych w selektorze podanym w rejestrze BX.

```
        lar   ax, bx       ; Pobierz prawa dostępu
        test  ax, 200h     ; Testuj bit rozmiaru
        jz    s16bit       ; Jeśli nie jest ustawiony,
                           ; to wykonuj operacje
                           ; 16-bitowe, w przeciwnym
                           ; razie operacje 32-bitowe
        .
        .
s16bit:
```

Load Descriptor Table

88/86	286	386
	●	●

LGDT/LIDT/LLDT

ZASTOSOWANIE

Załadowanie rejestru tablicy deskryptorów.

SKŁADNIA

`LGDT pamięć`	*Pobierz z pamięci 6 bajtów i prześlij je do rejestru globalnej tablicy deskryptorów GDT*
`LIDT pamięć`	*Pobierz z pamięci 6 bajtów i prześlij je do rejestru tablicy deskryptorów przerwań IDT*
`LLDT rejestr`	*Prześlij dwubajtowy selektor z rejestru do rejestru lokalnej tablicy deskryptorów LDT*
`LLDT pamięć`	*Pobierz z pamięci dwubajtowy selektor i prześlij go do rejestru lokalnej tablicy deskryptorów LDT*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Odwzorowanie pamięci w trybie chronionym odbywa się na podstawie tablic deskryptorów. Powyższe instrukcje używane są przez system operacyjny do ich modyfikowania.

PATRZ

SGDT, SIDT, SLDT	*Prześlij rejestry tablic deskryptorów do pamięci*

LGDT/LIDT/LLDT

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
LGDT pamięć	00001111 00000001 mod,010,r/m przemieszczenie	lgdt desctab	-	11	11
LIDT pamięć	00001111 00000001 mod,011,r/m przemieszczenie	lidt intdesc	-	12	11
LLDT rejestr	00001111 00000000 mod,010,r/m	lldt ax	-	17	20
LLDT pamięć	00001111 00000000 mod,010,r/m przemieszczenie	lldt selector	-	19	24

PRZYKŁAD

Procedura ustawia rejestr globalnej tablicy deskryptorów GDT, by wskazywał on na jednoelementową tablicę.

```
            .MODEL    SMALL
            .386P
            .DATA
Descript    STRUCT                          ; Format
                                            ; deskryptora
entry
     SegLimitLow    DW    0                 ; Bity 0-15
                                            ; rozmiaru segmentu
     SegBaseLow     DW    0                 ; Adres bazowy 0-15
     SegBaseMid     DB    0                 ; Adres
                                            ; bazowy 16-23
     Access         DB    0                 ; Bity dostępu
     MiscBits       DB    0                 ; Bity dodatkowe
                                            ; i bity 16-19
                                            ; rozmiaru
     SegBaseHi      DB    0                 ; Adres
                                            ; bazowy 24-31
Descript    ENDS
; Ten deskryptor pokrywa pierwszy megabajt pamięci
GDTble      Descript    < 0FFFFh, 0, 0, 10010010b, 00001111b,
0 >
GDTLimit    DW    7                         ; Rozmiar tablicy
```

```
                                        ; deskryptorów - 1
GDTBase     DD    ?
            .CODE
SetGDTR      PROC
             mov    ax, SEG DGROUP
             mov    bl, ah
             shl    ax, 4                ; AX razy 16
             shr    bl, 4                ; Starszy bajt
             xor    bh, bh
             add    ax, OFFSET GDTble    ; Dodaj
                                         ; przemieszczenie
                                         ; tablicy
             adc    bx, 0
             mov    WORD PTR GDTBase, ax ; Zapisz
                                         ; bezwzględny
                                         ; adres tablicy
             mov    WORD PTR GDTBase+2, bx
             lgdt   GDTBase              ; Załaduj
                                         ; rejestr GDTR
             ret
SetGDTR      ENDP
```

Load Machine Status Word — 88/86 286 ● 386 ●

LMSW

ZASTOSOWANIE

Załadowanie słowa stanu procesora MSW nową wartością.

SKŁADNIA

`LMSW reg/mem` — *Prześlij do MSW daną z rejestru lub pamięci*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Słowo stanu procesora MSW jest zbiorem znaczników, wykorzystywanych w trybie rzeczywistym i chronionym. Znajdują się w nim bity:

Bit trybu chronionego,
Bit obecności koprocesora,
Bit emulacji koprocesora,
Bit przełączenia zadania,
Bit typu rozszerzenia.

LOCK

PATRZ

SMSW *Zapamiętanie MSW*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
rejestr	00001111 00000001 mod,110,r/m	lmsw ax	-	3	10
pamięć	00001111 00000001 mod,110,r/m przemieszczenie	lmsw flags	-	6	10

PRZYKŁAD

Najmłodszy bit słowa MSW jest bitem włączenia trybu chronionego. Poniższa procedura ustawia go, włączając tryb chroniony.

```
.
.
.
smsw  ax        ; Pobierz do AX słowo MSW
or    ax, 1     ; Ustaw bit PE
lmsw  ax        ; Prześlij nowe MSW
.
.
.
```

Lock the Bus

88/86	286	386
●	●	●

LOCK

ZASTOSOWANIE

Zamknięcie dostępu do magistrali innym procesorom na czas wykonywania innej instrukcji.

SKŁADNIA

```
LOCK instrukcja
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Przedrostek LOCK jest najczęściej używany z instrukcjami zapisu lub odczytu z pamięci.

PATRZ

ESC	*Wysłanie instrukcji do koprocesora*
WAIT	*Czekanie na gotowość koprocesora*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11110000	lock inc numb	2	0	0

PRZYKŁAD

Przedrostek LOCK jest często używany w środowisku wieloprocesorowym przy dostępie do wspólnej pamięci. W poniższym przykładzie dwa procesory używają bajtu **EventCnt**, w którym jest przechowywana liczba pojawień się pewnego zdarzenia.

```
; Po przyjściu zdarzenia magistrala jest blokowana
; na czas modyfikacji licznika
          .CODE
CountEm   PROC FAR
          call  isevent        ; Sprawdź, czy pojawiło
                               ; się zdarzenie
          test  ax, eventflg
          jz    noevent
          lock  inc EventCnt   ; Zlicz zdarzenie
noevent:
          ret
CountEm   ENDP
```

Load Segment Limit — 88/86 286 ● 386 ●

LSL

ZASTOSOWANIE

Załadowanie do rejestru długości segmentu.

SKŁADNIA

`LSL reg, selektor`	*Selektor może być umieszczony w rejestrze lub pamięci. Rozmiar segmentu umieszczany jest w rejestrze reg.*

LSL

ZNACZNIKI

Znacznik zera jest ustawiany, gdy podany selektor jest poprawny i widoczny z bieżącego poziomu uprzywilejowania. Reszta znaczników pozostaje bez zmian.

KOMENTARZ

Instrukcja LSL jest dostępna tylko w trybie chronionym. Ponieważ w trybie tym segmenty mogą być różnej długości, to zapewniono rozkaz, który zwraca długość segmentu.

PATRZ

LAR *Pobranie praw dostępu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
reg, reg	00001111 00000011 mod,reg,r/m	lsl ax,bx	-	14	20
reg,pamięć	00001111 00000011 mod,reg,r/m przemieszczenie	lsl ax,selector	-	16	21

PRZYKŁAD

Procedura powoduje wyzerowanie wszystkich bajtów w segmencie wskazywanym przez BX.

```
        cld                 ; Zwiększanie adresów
        movzx ebx, bx       ; Przekształcenie selektora
                            ; na wartość 32-bitową
        lsl   ecx, ebx
        mov   eax, 0
        shr   ecx, 2        ; Podziel rozmiar segmentu
                            ; przez 4
        mov   es, bx        ; Pobranie selektora segmentu
        xor   edi, edi      ; Ustawienie przemieszczenia
cloop:
        stosd               ; Wyzeruj 4 bajty
        dec   ecx
        jnz   cloop
```

Load Task Register 88/86 286 386
● ●

LTR

ZASTOSOWANIE

Załadowanie rejestru zadania.

SKŁADNIA

`LTR reg/pamięć` *Reg/pamięć zawiera wartość, która będzie przesłana do rejestru zadania TR*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Rejestr zadania wskazuje na segment zadania aktywnego. Wszystkie konieczne działania, związane z operacją przełączania zadań, wykonuje już procesor. Instrukcja LTR służy tylko do wskazania pierwszego zadania.

PATRZ

STR

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
LTR reg	00001111 00000000 mod,001,r/m	ltr ax	-	17	23
LTR pamięć	00001111 00000000 mod,001,r/m przemieszczenie	ltr task	-	19	27

PRZYKŁAD

Trzy funkcje, przedstawione w poniższym przykładzie, służą do inicjacji mechanizmu wielozadaniowości, stworzenia nowego zadnia i dokonania przełączenia pomiędzy zadaniami. Przyjmują one, że tablica GDT zawiera deskryptor segmentu, w którym będą zapisywane dane o zadaniu. Selektor tego segmentu ma wartość 28h; następne 64 deskryptory musi być zarezerwowane dla innych możliwych zadań.

```
.MODEL    SMALL
PUBLIC    InitTask                ; Stwórz
                                  ; pierwsze
                                  ; zadanie
PUBLIC    NewTask                 ; Stwórz nowe
```

```
                                                ; zadanie
          PUBLIC   SwitchTask                   ; Przełącz
                                                ; zadania
          .386
TaskSeg   SEGMENT   WORD PRIVATE USE16 'TASK'
TaskCnt   DW   0                                ; Liczba zadań
CurTask   DW   0                                ; Zadanie,
                                                ; które jest
                                                ; aktualnie
                                                ; wykonywane
TskData   DB   104 * 64 dup (?)                 ; Miejsce na
                                                ; dane o
                                                ; stanie zadań
TskJmp    DD   0
TaskSeg   ENDS
          .CODE
; Inicjacja mechanizmu przełączania zadań
InitTask  PROC
          mov   ax, 8h                          ; Selektor
                                                ; tablicy GDT
          mov   ds, ax                          ; Załaduj go
                                                ; do DS
          mov   si, 30h                         ; Przemieszcz-
                                                ; enie
                                                ; deskryptora
                                                ; pierwszego
                                                ; zadania
          mov   ax, 103h                        ; Rozmiar
                                                ; segmentu TSS
          mov   WORD PTR [si], ax
          mov   ax, SEG TaskSeg                 ; Numer
                                                ; segmentu TSS
          mov   bl, ah
          shl   ax, 4
          shr   bl, 4
          add   ax, OFFSET TskData              ; Dodanie
                                                ; przemieszcz-
                                                ; enia danych
          adc   bl, 0                           ; zadania
          mov   WORD PTR [si+2], ax
          mov   BYTE PTR [si+4], bl
          mov   WORD PTR [si+5], 011101001b ; Znaczniki
          mov   BYTE PTR [si+7], 0
          mov   ax, 28h                         ; Selektor
                                                ; TaskSeg
          mov   ds, ax
```

```
            inc   TaskCnt
            mov   ax, 30h                      ; Pierwszy
                                               ; deskryptor
                                               ; TSS
            ltr   ax                           ; Załaduj
                                               ; rejestr
                                               ; zadania
            ret
InitTask    ENDP
; Umieść nowe zadanie w tablicy
; AX = Przemieszczenie funkcji dla zadania
; BX = Selektor tej funkcji
; CX = Selektor stosu
NewTask     PROC
            push  ax                           ; Zachowaj
                                               ; parametry
            push  bx
            push  cx
            mov   ax, 8h                       ; Selektor
                                               ; tablicy GDT
            mov   ds, ax                       ; Załaduj go
                                               ; do DS
            mov   si, TaskCnt                  ; Numer
                                               ; następnego
                                               ; zadania
            shl   si, 3                        ; pomnożony
                                               ; przez rozmiar
                                               ; deskryptora
            add   si, 30h                      ; Przemieszcz-
                                               ; enie
                                               ; pierwszego
                                               ; deskryptora
                                               ; TSS
            mov   ax, 103h                     ; Rozmiar
                                               ; segmentu TSS
            mov   WORD PTR [si], ax
            mov   ax, SEG TaskSeg              ; Numer
                                               ; segmentu TSS
            mov   bl, ah
            shl   ax, 4
            shr   bl, 4
            mov   cx, TaskCnt
            shl   cx, 3
            add   cx, OFFSET TskData
            adc   bl, 0
            add   ax, cx                       ; Dodanie
```

```
                                        ; przemieszcz-
                                        ; enia danych
        adc   bl, 0                     ; zadania
        mov   WORD PTR [si+2], ax
        mov   BYTE PTR [si+4], bl
        mov   WORD PTR [si+5], 011101001b ; Znaczniki
        mov   BYTE PTR [si+7], 0
        mov   ax, 28h                   ; Selektor
                                        ; TaskSeg
        mov   ds, ax
        mov   si, TaskCnt               ; Pobierz
                                        ; numer zadania
        shl   si, 3
        mov   ax, si
        shl   si, 2
        add   ax, si
        shl   si, 1
        add   si, ax                    ; AX razy 104
        add   si, OFFSET TskData
        sub   eax, eax                  ; Wyzeruj
                                        ; starsze słowo
        pop   cx                        ; Odtwórz
                                        ; parametry
        pop   bx
        pop   ax
        mov   [si+20h], eax             ; Zapisz EIP
        mov   DWORD PTR [si+38h], 1024  ; Zapisz ESP
        mov   [si+4Ch], ax
        mov   [si+50h], cx
        inc   TaskCnt
        ret
NewTask ENDP
; Przełączenie na wykonywanie następnego zadania.
; Przełączenie wywołuje pojawienie się przerwania
; zegarowego lub też zadanie, które na pewien czas nie
; potrzebuje czasu procesora
SwitchTask PROC
          mov   ax, 28h                 ; Selektor
                                        ; TaskSeg
          mov   ds, ax
          mov   ax, CurTask
          inc   ax                      ; Zmień na
                                        ; następne
                                        ; zadanie
          cmp   ax, TskCnt              ; Sprawdź, czy
                                        ; nie jest to
```

```
                                        ; ostatnie
        jb      Sk1                     ; zadanie
                                        ; z listy
        mov     ax, 1
Sk1:
        mov     CurTask, ax
        shl     ax, 3                   ; Pomnóż
                                        ; przez 8
        add     ax, 30h                 ; Pierwszy
                                        ; selektor
        mov     WORD PTR TskJmp+2, ax
        jmp     DWORD [TskJmp]          ; Przełącz
                                        ; zadania
SwitchTask ENDP
```

Store AH into the Flag Register

88/86	286	386
●	●	●

SAHF

ZASTOSOWANIE

Przepisanie bajtu z rejestru AH do rejestru znaczników.

SKŁADNIA

SAHF

ZNACZNIKI

Wartości znaczników znaku, zera, parzystości, przeniesienia, i przeniesienia pomocniczego są ustawiane według danej z rejestru AH.

KOMENTARZ

Interpretacja bitów rejestru AX jest następująca:

LSB MSB

Przeniesienie, rezer, Parzystość, rezer, Przeniesienie pomocnicze, rezer, Zero, Znak

PATRZ

LAHF *Pobranie znaczników*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	10011110	sahf	4	2	3

SGDT/SIDT/SLDT

PRZYKŁAD

Patrz przykład do instrukcji LAHF.

Store Descriptor Table — 88/86 | 286 ● | 386 ●

SGDT/SIDT/SLDT

ZASTOSOWANIE

Przesłanie bieżącej wartości rejestru tablicy deskryptorów do pamięci.

SKŁADNIA

`SGDT pamięć`	*Przesłanie do pamięci rejestru globalnej tablicy deskryptorów GDT*
`SIDT pamięć`	*Przesłanie do pamięci rejestru tablicy deskryptorów przerwań IDT*
`SLDT rejestr`	*Przesłanie selektora lokalnej tablicy deskryptorów LDT do rejestru*
`SLDT pamięć`	*Przesłanie selektora lokalnej tablicy deskryptorów LDT do pamięci*

ZNACZNIKI

Bez zmian.

KOMENTARZ

Instrukcje te mogą być zastosowane w trybie chronionym do lokalizacji tablic deskryptorów. Korzysta z nich system operacyjny przy modyfikowaniu wartości deskryptorów.

PATRZ

LGDT, LIDT, LLDT — *Pobranie danej do rejestr tablicy deskryptorów*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
SGDT	00001111 00000001 mod,000,r/m przemieszczenie	sgdt global	-	11	9
SIDT	00001111 00000001 mod,001,r/m przemieszczenie	sidt intdesc	-	12	9
SLDT rejestr	00001111 00000000 mod,000,r/m	sldt ax	-	2	2

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
SLDT pamięć	00001111 00000000 mod,000,r/m przemieszczenie	sldt selector	-	3	2

PRZYKŁAD

Dzięki instrukcji SGDT system operacyjny może zlokalizować położenie globalnej tablicy deskryptorów GDT. W tablicy tej musi istnieć element, który opisuje całą pamięć wskazywaną przez segment danych.

```
; Pobierz przemieszczenie i rozmiar globalnej tablicy
; deskryptorów. Adres prześlij do rejestru EDI,
; a rozmiar do ECX.
GDTable   PWORD   1 dup(?)              ; Miejsce na
                                        ; tablicę GDT
EditGDT:
          sgdt  GDTable                 ; Pobierz dane
          mov   cx, WORD PTR GDTable    ; Rozmiar
          mov   edi, DWORD PTR GDTable
          .
          .
          .
```

Store the Machine Word — 88/86 286 ● 386 ●

SMSW

ZASTOSOWANIE

Przesłanie rejestru sterującego procesora MSW.

SKŁADNIA

```
SMSW rejestr
SMSW pamięć
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Słowo stanu procesora jest zbiorem znaczników wykorzystywanych przez procesory 80286 i 80386. Aby sprawdzić ich stan, należy je najpierw pobrać do pamięci lub rejestru.

STD

PATRZ

LMSW *Załadowanie rejestru MSW*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
rejestr	00001111 00000001 mod,100,r/m	smsw ax	-	2	10
pamięć	00001111 00000001 mod,100,r/m przemieszczenie	smsw machsw	-	3	3

PRZYKŁAD

Patrz przykład do instrukcji LMSW.

Set the Direction Flag

88/86	286	386
●	●	●

STD

ZASTOSOWANIE

Ustawienie znacznika kierunku.

SKŁADNIA

STD *Ustaw znacznik kierunku.*

ZNACZNIKI

Znacznik kierunku jest ustawiany.

KOMENTARZ

Gdy znacznik kierunku jest ustawiony, to operacje łańcuchowe powodują zmniejszanie rejestru indeksowego, w przeciwnym razie jest on zwiększany.

PATRZ

STOS/MOVS/SCANS/INS/OUTS/CMPS *Operacje łańcuchowe*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11111101	std	2	2	2

PRZYKŁAD

Patrz przykład do instrukcji CMPS i SCANS w rozdziale 5.

Set Interrupt Flag

88/86	286	386
●	●	●

STI

ZASTOSOWANIE

Ustawienie znacznika zezwolenia na przerwanie.

SKŁADNIA

`STI` *Ustaw znacznik zezwolenia na przerwanie*

KOMENTARZ

Instrukcja ta zapewnia przyjmowanie przez procesor wszystkich przerwań. Gdy w pewnym punkcie programu konieczne jest ich zablokowanie, powinny one jak najszybciej zostać włączone. To zapewni, że urządzenia, takie jak klawiatura, których działanie oparte jest na przerwaniach, będą poprawnie pracowały.

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	11111011	sti	2	2	2

PRZYKŁAD

Patrz przykład do instrukcji CLI.

Store Task Register

88/86	286	386
	●	●

STR

ZASTOSOWANIE

Pobranie wartości rejestru zadania.

SKŁADNIA

`STR reg/pamięć` *Wartość rejestru TR będzie przesłana do reg/pamięć*

ZNACZNIKI

Bez zmian.

VERR/VERW

KOMENTARZ

Zawartość rejestru zadania jest dla innych programów bezużyteczna. Wskazuje ona selektor, pod który nie można zarówno zapisywać, jak i nie można go odczytywać. Instrukcja STR istnieje tylko jako uzupełnienie rozkazu LTR.

PATRZ

LTR — *Załadowanie rejestru zadania*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
STR reg	00001111 00000000 mod,001,r/m	str bx	-	2	2
STR pamięć	00001111 00000000 mod,001,r/m przemieszczenie	str task	-	3	2

PRZYKŁAD

Patrz przykład do instrukcji LTR.

Vwrify Read or Write Status — 88/86 286 ● 386 ●

VERR/VERW

ZASTOSOWANIE

Weryfikacja segmentu w trybie chronionym do odczytu lub zapisu.

SKŁADNIA

`VERR selektor` — *Selektor jest 16-bitową daną, umieszczoną w rejestrze lub pamięci*

`VERW selektor` — *Selektor jest 16-bitową daną, umieszczoną w rejestrze lub pamięci*

ZNACZNIKI

Jeśli użyta została instrukcja VERR i podany segment ma prawo do odczytu, to ustawiany jest znacznik zera. Jest on również ustawiany, gdy użyto rozkazu VERW i segment ma prawo zapisu. Reszta znaczników pozostaje bez zmian.

KOMENTARZ

Każdy segment w trybie chronionym posiada swój zestaw atrybutów, które kontrolują dostęp do niego. Wykonanie na nim niedopuszczalnej operacji spowoduje powstanie wyjątku. Powyższe instrukcje pozwalają stwierdzić, które działania są zabronione.

PATRZ

LAR	*Ładowanie praw dostępu*
LSL	*Ładowanie długości segmentu*

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
VERR rejestr	00001111 00000000 11,100,r/m	verr ax	-	14	10
VERR pamięć	00001111 00000000 mod,100,r/m	verr selector	-	16	11
VERW rejestr	00001111 00000000 11,101,r/m	verw ax	-	14	15
VERW pamięć	00001111 00000000 mod,101,r/m	verw selector	-	16	16

PRZYKŁAD

Przed wykonaniem zapisu do segmentu, program powinien się upewnić, że operacja ta nie jest zabroniona.

```
; Sprawdź, czy do segmentu można zapisywać dane
          .386p                ; Udostępnienie instrukcji
                               ; trybu chronionego
          verw  selector       ; Testuj prawo do zapisu
          jnz   nowrite        ; Jeśli znacznik zera nie
                               ; jest ustawiony, to do
                               ; segmentu nie można
                               ; zapisywać danych
          mov   es, selektor   ; Prześlij selektor do
                               ; rejestru segmentowego
          .                    ; Wykonaj żądane operacje
          .
          .
nowrite:
```

WAIT

Wait	88/86	286	386
	●	●	●

WAIT

ZASTOSOWANIE

Oczekiwanie na gotowość koprocesora.

SKŁADNIA

```
WAIT
```

ZNACZNIKI

Bez zmian.

KOMENTARZ

Koprocesor 8087 i pozostałe mogą przyjąć w jednej chwili tylko jedną instrukcję. By upewnić się, że koprocesor może pobrać następny rozkaz, należy zastosować instrukcję WAIT.

PATRZ

ESC

TAKTOWANIE

Tryb adresowania	Kodowanie	Przykład	88/86	286	386
domyślne	10011011	wait	4	3	6

PRZYKŁAD

Patrz przykład do instrukcji ESC.

III Operatory i Dyrektywy Turbo Assemblera

» Deklarowanie segmentów

» Definicje zmiennych i ich rozmieszczenie w pamięci

» Makrodefinicje, procedury i programowanie modułowe

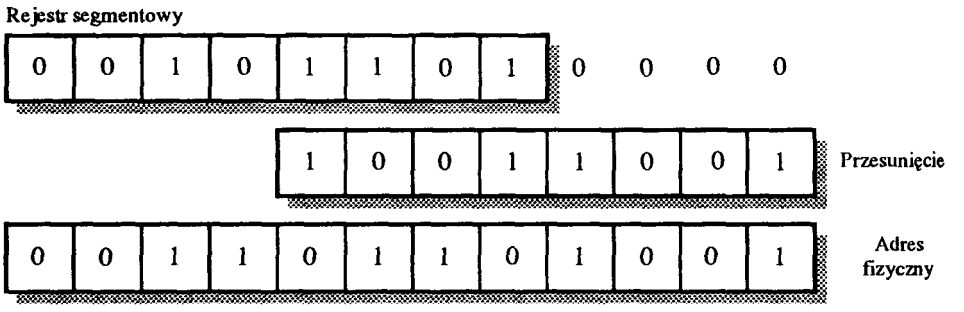
» Obsługa błędów i tworzenie raportów

Rozdział 8 *Deklarowanie segmentów*

Segmenty to jeden z bardziej złożonych aspektów programowania procesorów 80x86. Nawet najprostszy program musi zadbać o prawidłową inicjację rejestrów segmentowych. Przy pisaniu większych programów, aby działały one poprawnie, należy stale kontrolować zawartość tych rejestrów. Dzięki dyrektywom segmentowym zadanie to staje się trochę łatwiejsze. Są one, wraz z ilustrującymi je przykładami, opisane w pierwszej części rozdziału. Prócz tradycyjnych dyrektyw segmentowych Turbo Assembler posiada jeszcze uproszczone dyrektywy segmentowe. Eliminują one wiele kłopotów, definiując za programistę niektóre segmenty. Dzięki nim w wielu krótkich programach potrzebne jest tylko kilka linii dla prawidłowego zadeklarowania niezbędnych segmentów. Ich użycie opisane zostało w drugiej części rozdziału.

Co to jest segment

Różne rejestry segmentowe mają różne zastosowanie. Rejestr CS używany jest przy pobieraniu przez procesor instrukcji do wykonania. Za pośrednictwem rejestru DS realizowane są odwołania do danych, a za pomocą rejestru SS do stosu. Rejestr ES jest dodatkowym segmentem danych i może być używany do zmiany standardowego przypisania rejestrów.
W starszych procesorach do zaadresowania dużych przestrzeni adresowych wykorzystywano stronicowanie. Przez program była widziana tylko jedna strona o rozmiarze 65536 bajtów, reszta zaś była niedostępna. Natomiast na żądanie programu można było zmienić numer aktualnej strony. Stronicowanie zostało wyparte przez segmentację, która ułatwia zaadresowanie większej ilości pamięci.
Do zaadresowania jednego megabajtu pamięci procesor 8088 potrzebuje 20 bitów. By skorzystać z mechanizmu segmentacji rejestry segmentowe musiały by być również 20-bitowe. Jednak pozostałe rejestry są tylko 16-bitowe, trudno byłoby więc do rejestrów segmentowych przesyłać dane. Problem ten rozwiązano następująco: 16-bitowa wartość ładowana jest do starszych 16 bitów rejestru segmentowego, a jego młodsze bity są zawsze równe zero. Oznacza to więc, że segment może się zaczynać tylko od adresów podzielnych przez 16. Jest to rozwiązanie może niedoskonałe, ale na pewno lepsze niż stronicowanie.
Przy odwoływaniu się do pamięci procesor mnoży zawartość rejestru segmentowego przez 16 i dodaje przemieszczenie. Operacja ta jest przedstawiona na rysunku 8-1.
Każdy segment może zaczynać się co 16 bajtów, a jego rozmiar może wynosić 65536 bajtów. Oznacza to, że istnieje więcej niż jedna para segment:przemieszczenie, która wskazuje to samo miejsce w pamięci. Gdy asembler wyznacza przemieszczenie etykiety, to oblicza je względem początku segmentu, w którym się ona znajduje. Należy więc zapewnić, by w rejestrze segmentowym znajdował się numer tego segmentu, którego użył asembler. Na przykład, gdy w programie znajdują się dwa segmenty danych, to będzie istniała konieczność zmiany wartości rejestru segmentowego, zależnie od położenia żądanej zmiennej.



Rysunek 8-1. ***Dodanie rejestru segmentowego i przemieszczenia***

Deklarowanie segmentów

W Turbo Assemblerze każdy segment ma swoją nazwę. Jest ona przypisywana każdemu segmentowi w procesie deklaracji. Ponadto podawana jest informacja o sposobie umieszczenia segmentu w pamięci i jego powiązaniu z innymi segmentami.
W Turbo Assemblerze istnieją dwa sposoby deklarowania segmentów. W pierwszym należy podać całą wymaganą o nim informację. Drugi sposób pozwala na wykorzystanie kilku predefiniowanych typów segmentów. Najpierw została przedstawiona wersja dłuższa, która pozwala zapoznać się ze stosowaną terminologią. Pozwoli to na zrozumienie działania dyrektyw uproszczonych.

Dyrektywa SEGMENT

Każdy segment musi zostać opisany dyrektywą SEGMENT. Ma ona dwa formaty, zależne od trybu pracy Turbo Assemblera:

Tryb MASM:

```
nazwa SEGMENT   [rozmieszczenie] [połączenie] [rozmiar]
['nazwa_klasy']
      .         ; kod lub dane
      .
      .
[nazwa] ENDS
```

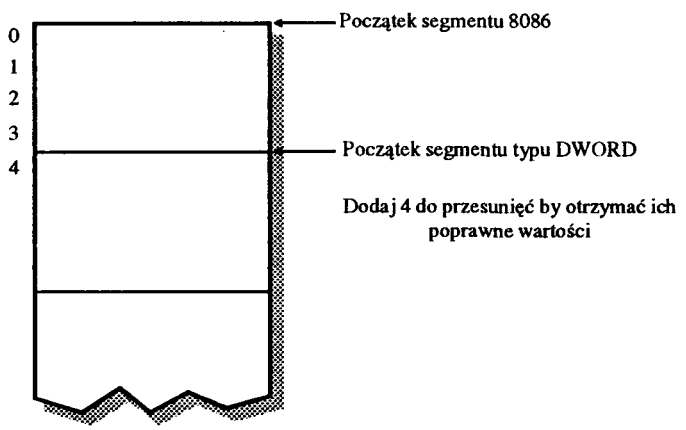
Tryb IDEAL:

```
SEGMENT   nazwa [rozmieszczenie] [połączenie] [rozmiar]
['nazwa_klasy']
      .    ; kod lub dane
      .
      .
ENDS   [nazwa]
```

W obu wypadkach segment zaczyna się dyrektywą SEGMENT, która wymaga podania jego nazwy. Kolejne argumenty są opisane poniżej. Następnie podawany jest kod lub dane, które będą umieszczone w segmencie. Dyrektywa ENDS oznacza koniec definicji segmentu.
W jednym pliku źródłowym może istnieć wiele deklaracji segmentów. Te, które mają te same nazwy, w pliku wynikowym zostaną złączone. Mimo że procesor 80x86 nie potrafi odwołać się do segmentów zagnieżdżonych, to jednak możliwość takiej deklaracji istnieje. Asembler dokona jednak ich reorganizacji. Taki sposób deklaracji nie jest zalecany.

ROZMIESZCZENIE SEGMENTÓW

Ponieważ młodsze cztery bity fizycznego adresu segmentu są zawsze równe zeru, to każdy segment musi zaczynać się na granicy paragrafu (16 bajtów), mimo że Turbo Assembler dopuszcza również inne możliwości. Przy kompilacji programu wartości rejestrów segmentowych są więc zaokrąglane do najbliższej 16-bajtowej granicy mniejszej od aktualnej wartości rejestru. Muszą również zostać poprawione wszystkie przemieszczenia. Proces ten ilustruje rysunek 8-2.



Rysunek 8-2. ***Rozmieszczenie segmentów na innych niż 16-bajtowe granicach.***

W tabeli 8-1 zostały natomiast przedstawione możliwe typy rozmieszczeń.

Tabela 8-1. Typy rozmieszczenia segmentów.

Typ	Opis
BYTE	Początek nowego segmentu rozpocznie się od następnego bajtu
WORD	Początek nowego segmentu rozpocznie się od następnego słowa o adresie parzystym
DWORD	Początek nowego segmentu rozpocznie się od następnego słowa o adresie podzielnym przez 4

Typ	Opis
PARA	Początek nowego segmentu będzie miał najbliższy wolny adres podzielny przez 16
PAGE	Początek nowego segmentu będzie miał najbliższy wolny adres podzielny przez 256

ŁĄCZENIE SEGMENTÓW

Następny argument dyrektywy **SEGMENT** oznacza sposób łączenia segmentów o tych samych nazwach. Przy zastosowaniu pierwszego typu łączenia właściwie żadne sklejanie nie jest wykonywane. Co więcej, nie jest nawet tworzony w pliku wynikowym żaden nowy segment. Typ łączenia AT jest stosowany do przypisania etykiet specjalnym obszarom pamięci. Za słowem AT występuje adres absolutny, który mówi asemblerowi o położeniu tych etykiet. Segmenty zadeklarowane ze słowem AT nie muszą zawierać ani kodu, ani danych, a tylko same etykiety. Ten typ jest używany przy odwoływaniu się do pamięci, która zawiera dane BIOS-u. Deklaracja takiego segmentu mogłaby wyglądać następująco:

```
          IDEAL; Tryb IDEAL
SEGMENT   BiosData BYTE AT 40h ; Deklaracja segmentu
                               ; umieszczanego na granicy
                               ; bajtów pod adresem
                               ; 0040:0000
          org     13h              ; Przemieszczenie 0013h
LABEL     MemSize     WORD       ; Etykieta, która pozwala
                                 ; odwołać się do słowa
                                 ; zawierającego ilość
                                 ; kilobajtów dostępnej
                                 ; pamięci
          org   17h
LABEL     KeyStatus   WORD       ; Stan klawiszy sterujących
                                 ; klawiatury
          org   1ah
LABEL     KeyHead     WORD       ; Wskaźnik początku bufora
                                 ; klawiatury
          org   1ch
LABEL     KeyTail     WORD       ; Wskaźnik końca bufora
                                 ; klawiatury
          org   1eh
LABEL     KeyBuffer   WORD       ; Pierwszy bajt bufora
ENDS      BiosData
```

Po zadeklarowaniu tego segmentu program może się odwołać do danych BIOS-u, używając odpowiednich etykiet. Poniższy kod pobiera z obszaru danych BIOS-u rozmiar pamięci:

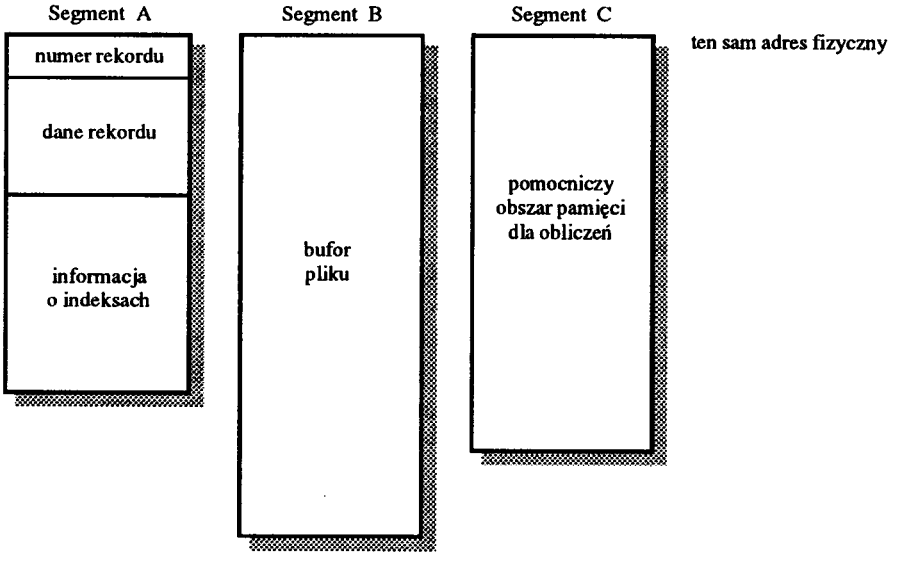
```
          .
          .
```

```
.
mov    ax, SEG BiosData ; Pobierz numer segmentu
mov    ds, ax           ; Prześlij go do rejestru
                        ; segmentowego
ASSUME    ds:BiosData   ; Poinformuj asembler
                        ; o zmianie DS
mov    ax, MemSize      ; Pobierz rozmiar pamięci
.
.
.
```

Następnym typem połączenia jest *COMMON*. W typie tym segmenty o tych samych nazwach, zaczynają się od tego samego adresu. O rozmiarze takiego segmentu decyduje więc segment najdłuższy. Na rysunku 8-3 przedstawionych jest kilka segmentów połączonych w typie COMMON.

Czasami różne moduły programu muszą inaczej traktować ten sam segment. Na przykład, dla jednego modułu może on stanowić obszar pamięci, w którym znajdują się rekordy pobierane i zapisywane na dysk. Drugi może widzieć go jako zbiór struktur, a trzeci jako ciągły obszar pamięci. Przy zastosowaniu typu COMMON każdy moduł może mieć zadeklarowany swój własny segment. Typ ten jest więc podobny do unii, występujących w językach wysokiego poziomu np. języku C.

Innym zastosowaniem połączenia typu COMMON jest wielokrotne wykorzystanie segmentów. Niektóre programy są bowiem zbyt duże. W wielu wypadkach dane, używane w jednym module, nie są już potrzebne w czasie działania innego. Na przykład, procedura sortowania może mieć segment, w którym przechowuje dane tymczasowe powstałe w czasie jej działania. Po uporządkowaniu rekordów pamięć ta staje się zbędna i może zostać



Rysunek 8-3. ***Segmenty połączone w typie COMMON.***

wykorzystana przez inny podprogram do innych celów. Ponieważ będą one używały całkiem innych zmiennych, to różny też będzie układ zadeklarowanych segmentów. Jednak połączenie ich w typie COMMON sprawi, że będą one istniały w tym samym bloku pamięci.
Jednak najczęściej stosowanym typem łączenia jest umieszczanie segmentów jeden za drugim. Długość tak powstałego segmentu jest wtedy sumą rozmiarów segmentów składowych. Ten sposób łączenia ilustruje rysunek 8-4. By go użyć należy skorzystać z typu *MEMORY* lub *PUBLIC*. Są to synonimy, a ich działanie spowoduje umieszczenie wszystkich segmentów o tej samej nazwie w ciągłym obszarze pamięci.
Typ *STACK* jest podobny do typów MEMORY czy PUBLIC. Powoduje on sklejenie segmentów o tych samych nazwach. Różnica polega tylko na innym przejawia sposobie inicjacji rejestrów w czasie ładowania programu. Do rejestru SS wpisywany jest adres początku segmentu STACK, a do SP jego długość. Jeśli w programie istnieje chociaż jeden segment tego typu, to nie jest konieczne pisanie kodu inicjującego stos, gdyż będzie to zrobione automatycznie w czasie ładowania programu.
Czasami jednak nie wolno pozwolić na połączenie segmentów. Kilka plików źródłowych może używać tych samych nazw dla różnych segmentów, które są wykorzystywane do podobnych celów. By uniknąć ich sklejenia, należy zadeklarować ich typ połączenia jako *PRIVATE*. Typ ten zostanie również przyjęty, gdy pole 'połączenie' zostanie w deklaracji segmentu pominięte.

ROZMIAR SEGMENTU

Jedną z cech procesora 80386 jest możliwość zaadresowania do czterech gigabajtów pamięci i wykorzystania rejestrów 32-bitowych. Mimo to procesor ten jest kompatybilny z wersjami starszymi. By wykorzystać obie te cechy w sposób jak najbardziej efektywny, w procesorze 80386 wprowadzono dwa rodzaje segmentów. W pierwszym z nich dane i przemieszczenia mają długość 16 bitów, w drugim zaś są 32-bitowe.
Typ segmentu może być wybrany w jego deklaracji. Najpierw trzeba jednak zezwolić na stosowanie instrukcji procesora 80386 dyrektywą **.386P**, **P386** lub **P386N**. Następnie za typem połączenia można podać typ segmentu jako USE16 (typ domniemany) lub USE32.
W poniższym przykładzie zdefiniowany został 32-bitowy segment kodu:

```
        .386P                   ; Udostępnienie instrukcji
                                ; procesora 80386
_CODE   SEGMENT   PUBLIC  USE32 'CODE'
_main   PROC  FAR
        mov   eax, 4            ; Instrukcja ta nie wymaga
                                ; zastosowania przedrostka
                                ; rozmiaru
        add   bx, ax            ; Rozkaz ten musi zostać
                                ; poprzedzony przedrostkiem
                                ; rozmiaru
        ret
_CODE   ENDS
```

KLASY SEGMENTÓW

Ostatni argument dyrektywy SEGMENT definiuje klasę segmentu. Jest ona używana przy porządkowaniu segmentów w pamięci. Program łączący umieści w pliku wykonywalnym wszystkie segmenty tej samej klasy obok siebie. Należy zauważyć, że nie są one w żaden sposób łączone, a po prostu umieszczane obok siebie. Pole klasy musi być objęte pojedynczymi (') lub podwójnymi (") cudzysłowami. Nazwa klasy może być natomiast dowolna. Często używane są 'CODE', 'DATA' i 'STACK'.

Łączenie i grupowanie segmentów

Dyrektywa SEGMENT pozwala w pewien sposób sterować rozmieszczeniem segmentów w pamięci. Ustawienie typu łączenia pozwala na umieszczenie segmentów o tych samych nazwach w pojedynczym segmencie. Pole klasy steruje położeniem segmentów w pliku wykonywalnym. Dodatkowo Turbo Assembler dostarcza kilka dyrektyw, które zwiększają kontrolę nad grupowaniem i porządkowaniem segmentów.

GRUPY

Dyrektywa **GROUP** poleca umieścić razem kilka segmentów. Istnieje ona zarówno w trybie MASM jak i IDEAL. Jej format jest następujący:

Tryb MASM:

```
NazwaGrupy   GROUP   NazwaSegmentu [, NazwaSegmentu]...
```

Tryb IDEAL:

```
GROUP        NazwaGrupy NazwaSegmentu [, NazwaSegmentu]...
```

Nazwa grupy w programie może być wykorzystywana na równi z nazwami segmentów. Podane segmenty są sklejane w jedną całość. A ponieważ grupa może być traktowana jako segment, jej rozmiar nie może przekroczyć 65536 bajtów.
Grupy służą do łączenia różnych rodzajów segmentów. Na przykład w pliku typy .COM, segmenty kodu, danych i stosu muszą znajdować się w jednym obszarze pamięci o maksymalnym rozmiarze 64k. Do ich sklejenia można wykorzystać właśnie dyrektywę **GROUP**:

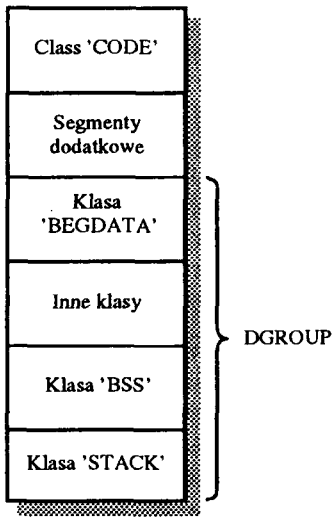
```
          ORG       100h
Program   GROUP     Cseg, Dseg
Cseg      SEGMENT   PARA PUBLIC 'CODE'
          ASSUME    cs:Program, ds:Program
Start     PROC  FAR
          mov   dx, OFFSET Msg
          mov   ah, 9h
          int   21h
          mov   ah, 4ch
          int   21h
Start     ENDP
Cseg      ENDS
Dseg      SEGMENT   PARA PUBLIC 'DATA'
Msg       DB        "Hello world", 0Dh, 0Ah, '$'
```

```
Dseg        ENDS
END         Start
```

UPORZĄDKOWANIE SEGMENTÓW

Zwykle Turbo Assembler umieszcza w pliku wykonywalnym segmenty w takiej kolejności, w jakiej występują one w pliku źródłowym. Jednym ze sposobów zmienienia tej kolejności jest podanie w deklaracji segmentu jego klasy. Innym jest użycie jednej z dyrektyw **.ALPHA**, **.SEQ** lub **DOSSEG**. Pierwsza z nich powoduje alfabetyczne uporządkowanie segmentów. Dyrektywa **.SEQ** zapewnia natomiast umieszczanie w pliku wynikowym segmentów w kolejności ich wystąpienia w pliku źródłowym.

Dyrektywa **DOSSEG** porządkuje segmenty w sposób stosowany w językach wysokiego poziomu. Powinna ona się pojawić tylko w tym module, w którym jest zdefiniowany adres wejścia do programu. Zinterpretowanie tej dyrektywy powoduje umieszczanie segmentów klasy 'CODE' na początku programu. Dalej występują segmenty, które nie są klasy 'CODE' i nie należą do grupy DGROUP. Po nich umieszczana jest dopiero ta grupa. W niej natomiast jako pierwsze występują segmenty klasy 'BEGDATA', a następnie segmenty, które nie są klasy 'BEGDATA', 'BSS' ani 'STACK'. Przedostatnie segmenty mają klasę 'BSS', a na końcu umieszczane są te z klasy 'STACK'. Uporządkowanie to zostało przedstawione na rysunku 8-4.



Rysunek 8-4. ***Uporządkowanie segmentów w dyrektywie DOSSEG.***

Operowanie na adresach segmentów i przemieszczeniach

System MS-DOS, ładując program do pamięci, inicjuje rejestry segmentowe. Rejestr CS wskazuje na początek segmentu kodu, rejestry DS i ES wskazują na przedrostek programu PSP. Jeśli zaś zadeklarowany został segment stosu, to rejestr SS wskazuje na jego początek, w przeciwnym razie na jeden z wewnętrznych stosów systemu MS-DOS.

Sposób załadowania rejestrów przez MS-DOS może czasami okazać się nie tym, czego wymaga poprawne działanie programu. Na przykład, w programie może istnieć wiele segmentów danych i przed odwołaniem się do zmiennych, które w nich występują, musi zostać odpowiednio załadowany rejestr DS. Poniższy przykład ilustruje użycie operatora SEG do pobrania adresu segmentu:

```
mov   ax, SEG Dseg    ; Pobierz adres segmentu Dseg
mov   ds, ax          ; Prześlij go do DS
```

Niestety, nie istnieje instrukcja, która przesyła daną natychmiastową bezpośrednio do rejestru segmentowego. Dlatego też adres segmentu jest najpierw wpisywany do AX, a dopiero potem kopiowany do DS. Inną techniką jest umieszczenie częściej stosowanych adresów w segmencie kodu i przesyłanie ich do rejestru DS, przy wykorzystaniu pośredniego trybu adresowania.

```
          mov   ds, cs:ptrdseg ; Pobierz adres
                               ; segmentu z pamięci
          .
          .
          .
ptrdseg   DW    SEG  Dseg      ; Umieszczenie adresu
                               ; segmentu w segmencie kodu
```

Każda komórka pamięci jest dostępna z wielu segmentów. Jeśli wartość rejestru segmentowego zostanie zmieniona o jeden, a wartość przemieszczenia o 16, ale w przeciwnym kierunku, to para segment:przemieszczenie będzie dalej wskazywała to samo miejsce pamięci. Zastosowanie dyrektywy **GROUP** sprawia, że dane stają się widoczne w kilku różnych segmentach. Asembler oblicza przemieszczenie, odejmując adres absolutny, który znajduje się w rejestrze segmentowym od absolutnego adresu etykiety. Ponieważ wartości tych rejestrów mogą być na bieżąco zmieniane, asembler nie ma sposobu na stwierdzenie jaka wartość znajduje się w nich aktualnie. Można go jednak o tym poinformować, stosując dyrektywę **ASSUME** o formacie:

```
ASSUME   sreg:segment [, sreg:segment]...
```

Asembler przyjmuje wtedy, że rejestr segmentowy sreg wskazuje segment podany w ostatniej dyrektywie.

Uproszczone dyrektywy segmentowe

Okazuje się, że w większości programów używane są takie same segmenty. Dane można zawsze umieszczać w segmencie o nazwie dseg, a kod w segmencie cseg. Generalnie segmenty te będą miały też ten sam typ połączenia i będą znajdowały się w tych samych grupach.

W językach wysokiego poziomu programista nie ma możliwości nadawania segmentom nazw. Robi to kompilator w czasie tworzenia pliku wynikowego, używając zawsze tych samych typów segmentów. Znając ich nazwy, można zapewnić sobie możliwość dołączenia procedur, napisanych w asemblerze, do programów stworzonych w tych językach.

Coraz więcej języków stosuje konwencję nazywania segmentów, wprowadzoną przez firmę Microsoft w swoich produktach. Turbo Assembler pozwala na jej użycie dzięki istnieniu

uproszczonych dyrektyw deklarujących niektóre często stosowane segmenty. Ich użycie pozwoli na połączenie z językami wysokiego poziomu, a także uprości proces tworzenia segmentów.

MODELE PAMIĘCI

Jednym z czynników, które mogą mieć wpływ na rozmieszczenie segmentów jest ilość użytej przez program pamięci. Jeśli cały mieści się on w 64k, ma wtedy sens stworzenie tylko jednego segmentu. Jeśli nie, to pierwszym krokiem jest umieszczenie kodu i danych w odrębnych segmentach. Gdy rozmiar kodu przekracza 64k, konieczne jest stworzenie kilku jego segmentów. Podobnie rzecz ma się z danymi.
Różne sposoby grupowania segmentów nazywane są modelami pamięci. Dyrektywy **.MODEL** i **MODEL**(tryb IDEAL) pozwalają wybrać odpowiedni z nich. Ich format jest następujący:

```
.MODEL    model_pamięci [, język]
```

W tabeli 8-2 przedstawiono wszystkie modele pamięci i ich wpływ na segmenty deklarowane dyrektywami uproszczonymi. Opcjonalne pole język mówi asemblerowi, standardu którego języka (C lub Pascal) ma on użyć przy tworzeniu ramek stosu.
Poza wpływem na rozmieszczenie segmentów dyrektywa .MODEL ustawia jeszcze wartości dwóch zmiennych **@CodeSize** i **@DataSize**. Mogą one zostać użyte do sprawdzenia, który model pamięci został wybrany. Na przykład w modelach MEDIUM, LARGE i HUGE adresy funkcji są czterobajtowe, a w modelach pozostałych tylko dwubajtowe. Zmienna **@CodeSize** może być więc użyta do zarezerwowania odpowiedniej ilości miejsca w pamięci:

```
IF    @CodeSize    EQ 0       ; Kompiluj następną instrukcję,
                              ; gdy @CodeSize jest równa 0
FUNCPTR      DW    THEFUNC    ; Zarezerwuj dwa bajty na adres
                              ; bliski
ELSE                          ; Kompiluj następną instrukcję,
                              ; gdy @CodeSize nie jest równa 0
FUNCPTR      DD    THEFUNC    ; Zarezerwuj cztery bajty na
                              ; adres daleki
ENDIF
```

Tabela 8-2. Modele pamięci.

	Segment standardowo przypisany do rejestru segmentowego			
Model	**CS**	**DS**	**ES**	**SS**
TINY	DGROUP	DGROUP	DGROUP	DGROUP
SMALL	_TEXT	DGROUP	DGROUP	DGROUP
MEDIUM	Nazwa_TEXT	DGROUP	DGROUP	DGROUP
COMPACT	_TEXT	DGROUP	FAR_DATA	DGROUP
LARGE	Nazwa_TEXT	DGROUP	FAR_DATA	DGROUP
HUGE	Nazwa_TEXT	DGROUP	FAR_DATA	DGROUP

Część Nazwa w Nazwa_TEXT jest różna dla każdego pliku źródłowego. Segment FAR_DATA jest zadeklarowany jako PRIVATE, dzięki temu każdy moduł posiada swój segment FAR_DATA.

SEGMENT KODU

Wszystko, co występuje za jedną z dyrektyw **.CODE** lub **CODESEG** jest, umieszczane przez Turbo Assembler w segmencie kodu.
Obie te dyrektywy są identyczne. W modelach MEDIUM, LARGE lub HUGE stworzonemu segmentowi nadawana jest nazwa _TEXT poprzedzona nazwą pliku źródłowego. W pozostałych modelach nazwą jest _TEXT. Do pobrania adresu segmentu nie jest konieczna znajomość nazwy segmentu kodu (_TEXT). Istnieje bowiem zmienna **@CODE**, w której jest ona przechowywana.

```
mov     ax, @CODE    ; Prześlij adres segmentu kodu do AX
mov     ds, ax       ; i do DS
ASSUME  ds:@CODE     ; Informacja dla asemblera o zmianie
                     ; zawartości rejestru segmentowego
```

Dyrektywa **ASSUME** jest konieczna, gdyż do rejestru DS została wpisana niestandardowa wartość. Gdy stosujemy uproszczone dyrektywy segmentowe, nie musimy podawać dyrektywy **ASSUME**, o ile rejestry segmentowe zawierają wartości podane w tabeli 8-2.

SEGMENT DANYCH

Dane są umieszczane w kilku segmentach zależnie od ich wykorzystania. Cztery z nich tworzą grupę zwaną DGROUP. Są to: _DATA dla danych zainicjowanych, _BSS dla danych niezainicjowanych, CONST dla danych stałych i STACK przeznaczony na stos. W modelu TINY w grupie tej znajduje się również segment kodu. Turbo Assembler automatycznie wpisuje do rejestru SS adres początku tej grupy, ale rejestr DS należy załadować już samodzielnie. Zmienna **@DATA** zawiera adres segmentu DGROUP.

```
mov   ax, @DATA      ; Pobierz adres grupy DGROUP
mov   ds, ax         ; Umieść go w DS
```

Dwa dodatkowe segmenty: **FAR_DATA** i **FAR_BSS** są używane w modelu LARGE. Podobnie FAR_DATA zawiera dane zainicjowane, a FAR_BSS dane nie zainicjowane. Są one typu PRIVATE, tak więc każdy moduł może mieć swoje, oddzielne segmenty danych. Istnieją dyrektywy uproszczone, które deklarują każdy z tych segmentów. **.DATA** i **DATASEG** służą do stworzenia segmentu _DATA. Dane nie zainicjowane umieszczane są w segmencie _BSS, który zaczyna się dyrektywą **.DATA?**. Stałe, jak na przykład, komunikaty czy adresy używane w odwołaniach pośrednich istnieją w segmencie **CONST**. Tworzony jest on dyrektywą **.CONST** lub **CONST**.
Ostatnie dwa segmenty deklarowane dyrektywami uproszczonymi przechowują dalekie dane zainicjowane (**.FARDATA** lub **FARDATA**) lub niezainicjowane (**.FARDATA?**).

SEGMENT STOSU

W systemie MS-DOS każdy program musi posiadać stos. Jest on deklarowany dyrektywami **.STACK** lub **STACK**, które powodują zarezerwowanie pewnego obszaru pamięci i zapewniają, że po załadowaniu programu rejestry SS i SP będą miały poprawne wartości.
Domniemanym rozmiarem rezerwowanego obszaru jest 1024 bajty. Można to zmienić, podając w dyrektywie **.STACK** lub **STACK** żądaną długość stosu. Powinna ona być taka, by wystarczyło miejsca na adresy powrotu funkcji, ramki stosu i zmienne lokalne. Głęboko

zagnieżdżone wywołania funkcji, które pojawiają się np. W procedurach rekurencyjnych, mogą spowodować przepełnienie stosu i zamazanie innych obszarów programu.

Definiowanie punktu wejścia do programu

Każdy program w systemie MS-DOS musi mieć określoną funkcję, która jest wykonywa jako pierwsza. W językach wysokiego poziomu rozwiązane jest to automatycznie poprzez umieszczenie pierwszej funkcji w specjalnie określonym miejscu lub poprzez nadanie jej określonej nazwy. Programy w asemblerze mogą rozpoczynać się od dowolnej funkcji. Wystarczy tylko podać jej nazwę w dyrektywie **END**. Dyrektywa ta musi wystąpić w każdym pliku źródłowym jako jego ostatnia linia. W pliku, który zawiera funkcję rozpoczynającą program, musi ona ponadto mieć argument, który byłby nazwą tego podprogramu.

Podsumowanie

Standardowe dyrektywy segmentowe pozwalają na tworzenie segmentów o dowolnych nazwach, typie, klasie itd. Dyrektywa **SEGMENT** definiuje tylko rodzaj pojedynczego segmentu. Natomiast dyrektywa **GROUP** pozwala na umieszczenie kilku z nich razem w jednym, spójnym obszarze pamięci. Dyrektywa **ASSUME** informuje zaś kompilator, które segmenty są wskazywane aktualnie przez rejestry segmentowe.
Jednak w większości programów potrzebne są tylko pewne typy segmentów. Turbo Assembler ma dyrektywy, które deklarują predefiniowane segmenty używane w językach wysokiego poziomu. Upraszcza to ich definiowanie i stwarza możliwość połączenia programów napisanych w asemblerze z programami powstałymi w innych językach.

.ALPHA, DOSSEG, .SEQ

ZASTOSOWANIE

Ustalenie sposobu uporządkowania segmentów w pliku wynikowym.

SKŁADNIA

.ALPHA	*Umieść segmenty w porządku alfabetycznym*
DOSSEG	*Umieść segmenty w porządku stosowanym w językach wysokiego poziomu*
.SEQ	*Umieść segmenty w kolejności ich występowania w plikach źródłowych*

KOMENTARZ

Normalnie Turbo Assembler zapisuje segmenty w kolejności ich występowania w plikach źródłowych. Podanie opcji /A zmienia ten porządek na alfabetyczny. Powyższe dyrektywy, niezależnie od użytej opcji, wymuszają konkretny porządek.

Dyrektywa **DOSSEG** różni się trochę od pozostałych, tym że wymaga tylko podania jej w pliku, który definiuje punkt wejścia do programu. Dyrektywy **.ALPHA** i **.SEQ** mogą natomiast wystąpić w każdym pliku źródłowym.

PATRZ

.MODEL	*Ta dyrektywa ma również wpływ na uporządkowanie segmentów*

PRZYKŁAD

W poniższym przykładzie segmenty zostaną umieszczone w porządku alfabetycznym mimo ich losowego występowania w pliku źródłowym.

```
        .ALPHA
Wseg    SEGMENT  PUBLIC  'WCLASS'
        .
        .
        .
Wseg    ENDS
Zseg    SEGMENT  PUBLIC  'ZCLASS'
        .
        .
        .
Zseg    ENDS
Aseg    SEGMENT  PUBLIC  'ACLASS'
        .
        .
        .
Aseg    ENDS
```

ASSUME

ZASTOSOWANIE

Przekazanie Turbo Assemblerowi informacji, które segmenty są wskazywane aktualnie przez rejestry segmentowe.

SKŁADNIA

```
ASSUME segreg:segment [, segreg:segment]...
ASSUME segreg:NOTHING
ASSUME NOTHING
```

KOMENTARZ

Ponieważ segmenty mogą się nakładać wiele par segment:przemieszczenie może wskazywać to samo miejsce pamięci. By móc obliczyć przemieszczenie, Turbo Assembler musi wiedzieć, jakie wartości zawierają poszczególne rejestry segmentowe. Dyrektywa **ASSUME** pozwala na przekazanie tego typu informacji.
Dyrektywa ta nie powoduje jednak załadowania rejestrów segmentowych. Należy się więc upewnić, czy w rejestrach znajduje się ta sama wartość, co użyta w dyrektywie.

PATRZ

SEGMENT

PRZYKŁAD

Poniższy przykład zawiera typową dyrektywę **ASSUME** stosowaną w programach z oddzielnymi segmentami kodu, danych i stosu. Rejestry CS i SS zostaną wypełnione podczas ładowania programu. Natomiast rejestr DS musi zostać załadowany przez program. Ponieważ zawartość rejestru ES jest przypadkowa, to przyjmuje się, że nie wskazuje on na nic.

```
ASSUME  CS:CODE, DS:DATA, SS:STACK, ES:NOTHING
mov   ax, SEG DATA  ; Umieść adres segmentu DATA w AX
mov   ds, ax        ; i prześlij go do rejestru DS
.
.
.
```

@Code

ZASTOSOWANIE

Zmienna, która zawiera adres segmentu kodu (_TEXT).

SKŁADNIA

`@CODE` *Adres segmentu kodu.*

KOMENTARZ

Zmienna **@Code** przechowuje adres segmentu kodu _TEXT. Można ją zastąpić, używając operatora **SEG** z nazwą segmentu:

```
mov  ax, SEG _TEXT
```

PATRZ

.CODE, CODESEG
.DATA, DATASEG
.FARDATA, FARDATA
.FARDATA?, FARDATA?

PRZYKŁAD

Najczęściej wykorzystuje się te zmienne do pobrania adresu określonego segmentu. W poniższym fragmencie programu do rejestru DS wpisywany jest adres segmentu danych.

```
        .MODEL   SMALL
        .DATA
        .
      ; Deklaracje zmiennych
        .
        .
        .CODE
        mov   ax, @Data     ; Pobierz adres do AX
        mov   ds, ax        ; i prześlij go do DS
        .                   ; Dalsza część programu
        .
        .
        .STACK  512
END
```

@CodeSize

ZASTOSOWANIE

Informacja o typie modelu pamięci.

SKŁADNIA

`@CodeSize` *Mówi, czy używane są procedury typu NEAR czy FAR*

.CODE, CODESEG

KOMENTARZ

Różne modele pamięci wymagają różnej długości wskaźników i różnych sposobów wykorzystania danych. Zmienna ta pozwala na napisanie kodu, który zachowuje się prawidłowo w każdym modelu pamięci.

PATRZ

.MODEL

PRZYKŁAD

W dużych modelach pamięci wskaźniki są czterobajtowe, w małych zaś dwubajtowe. Poniższy fragment programu testuje zmienną **@DataSize**, aby przydzielić odpowiednią ilość pamięci.

```
            .MODEL   SMALL
            .DATA
IF @DataSize EQ 0
DPTR   DW   SmplData     ; w małych modelach zarezerwuj
                         ; dwa bajty
ELSE
DPTR   DD   SmplData     ; w dużych modelach zarezerwuj
                         ; cztery bajty
ENDIF
IF @CodeSize EQ 0
DPTR   DW   SmplFunc     ; w małych modelach zarezerwuj
                         ; dwa bajty
ELSE
DPTR   DD   SmplFunc     ; w dużych modelach zarezerwuj
                         ; cztery bajty
ENDIF
```

.CODE, CODESEG

ZASTOSOWANIE

Dyrektywy uproszczone, które deklarują segment kodu.

SKŁADNIA

```
.CODE [nazwa]
```

KOMENTARZ

Przed użyciem jakiejkolwiek dyrektywy uproszczonej konieczne jest podanie modelu pamięci. Argument nazwa używany jest w modelach MEDIUM, LARGE i HUGE, przy definiowaniu wielu segmentów kodu w jednym pliku źródłowym. Jeśli nie został on podany, to przyjmuje się go jako nazwę aktualnie kompilowanego zbioru.

PATRZ

ASSUME, SEGMENT, ENDS
.DATA, .STACK, .FARDATA
.MODEL

PRZYKŁAD

Zastosowanie dyrektyw uproszczonych zostało przedstawione w poniższym przykładzie.

```
        .MODEL  SMALL           ; Model pamięci SMALL
        .CONST                  ; Stwórz segment
                                ; na dane stałe
Mesg    DB   "Hello World", 0dh, 0ah, '$'
        .DATA
LastDos DB    0
        .CODE                   ; Stwórz segment kodu
Start:
        mov   ax, @Data         ; Pobierz adres segmentu
                                ; danych
        mov   ds, ax            ; i umieść go w DS
        mov   dx, OFFSET Mesg
        mov   ah, 9             ; Funkcja, która wyświetla
                                ; ciąg znaków
        mov   LastDos, ah       ; Zapamiętaj ostatnią
                                ; komendę
        int   21h               ; Wywołanie MS-DOS-a
        mov   ah, 4ch           ; Funkcja, która kończy
                                ; program
        mov   LastDos, ah       ; Zapamiętaj ostatnią
                                ; komendę
        int   21h
        .STACK   512            ; Stwórz 512 bajtowy stos
END     Start
```

@CurSeg

ZASTOSOWANIE

Zmienna, która przechowuje adres bieżącego segmentu.
Patrz **@Code**.

@Data

ZASTOSOWANIE

Zmienna, która przechowuje adres segmentu danych (_DATA).
Patrz **@Code**.

.DATA, .DATA?, DATASEG

ZASTOSOWANIE

Dyrektywy uproszczone, które deklarują segment danych zainicjowanych (**.DATA**, **DATASEG**) i nie zainicjowanych (**.DATA?**).

SKŁADNIA

```
.DATA
DATASEG
.DATA?
```

KOMENTARZ

W językach wysokiego poziomu dane zainicjowane i nie zainicjowane znajdują się w oddzielnych segmentach. Nie jest to ważne dla systemu MS-DOS, ale inne systemy operacyjne mogą korzystać z tego rozróżnienia. Przy odwoływaniu się do danych umieszczonych w grupie **DGROUP** nie istnieje jednak potrzeba rozróżniania danych w tych segmentach. Przed użyciem tej i innych dyrektyw uproszczonych konieczne jest podanie modelu pamięci.

PATRZ

.CODE, .CONST, .STACK
.MODEL
UDATASEG

PRZYKŁAD

Patrz przykład dla dyrektywy **.CODE**.

@DataSize

ZASTOSOWANIE

Informacja o typie modelu pamięci.

SKŁADNIA

`@DataSize`	*Mówi, czy program jest kompilowany w modelu SMALL, LARGE czy HUGE*

KOMENTARZ

Różne modele pamięci wymagają różnej długości wskaźników i różnych sposobów wykorzystania danych. Zmienna ta pozwala na napisanie kodu, który zachowuje się prawidłowo w każdym modelu pamięci.

PATRZ

.MODEL

PRZYKŁAD

Patrz przykład do dyrektywy **.@CodeSize**.

DOSSEG

Patrz dyrektywa **.ALPHA**.

END

ZASTOSOWANIE

Zaznaczenie końca pliku źródłowego z możliwością zdefiniowania punktu wejścia do programu.

SKŁADNIA

```
END   [Funkcja_startowa]
```

KOMENTARZ

W każdym pliku źródłowym w jego ostatniej linii musi pojawić się dyrektywa **END**. Za jej pomocą można również zdefiniować punkt wejścia do programu. Jednak może być on podany tylko w jednym zbiorze źródłowym.

PATRZ

ENDS
ENDP

PRZYKŁAD

Patrz przykład do **.CODE**.

ENDS

ZASTOSOWANIE

Dyrektywa, która kończy definicję segmentu.

SKŁADNIA

Tryb MASM:

```
nazwa_segmentu ENDS
```

Tryb IDEAL:

```
ENDS nazwa_segmentu
```

KOMENTARZ

Każdy segment, zadeklarowany standardową dyrektywą segmentową, musi się kończyć dyrektywą **ENDS**. Jest ona używana również do kończenia definicji struktur (patrz rozdział 9).

PATRZ

END
ENDP

PRZYKŁAD

Patrz przykład do **GROUP**.

@FarData

ZASTOSOWANIE

Zmienna, która przechowuje adres dalekiego segmentu danych (FAR_DATA).
Patrz dyrektywa **@Code**.

@FarData?

ZASTOSOWANIE

Zmienna, która przechowuje adres dalekiego segmentu nie zainicjowanych danych (FAR_DATA?).
Patrz dyrektywa **@Code**.

.FARDATA, .FARDATA?, FARDATA

ZASTOSOWANIE

Dyrektywy uproszczone, które deklarują daleki segment danych zainicjowanych (**.FARDATA, FARDATA**) i niezainicjowanych (**.FARDATA?**). Opcjonalnie podawana nazwa może być użyta do definiowania wielu takich segmentów w jednym pliku źródłowym.

SKŁADNIA

```
.FARDATA    [nazwa]
FARDATA     [nazwa]
.FARDATA?   [nazwa]
```

KOMENTARZ

Dalekie segmenty danych są używane w dużych modelach pamięci. Każdy z nich jest typu **PRIVATE**, to sprawia, że istnieją oddzielne segmenty dla każdego modułu.

Daleki segment danych jest oddzielony od innych segmentów danych. Oznacza to, że przed odwołaniem się do niego, konieczne jest załadowanie rejestru segmentowego odpowiednim adresem. Jeśli segment znajduje się w tym samym pliku, to adres ten znajduje się w zmiennej **@FarData**. Dla odwołania się do danych, istniejących w innych modułach, potrzebny jest inny schemat.

Segment typu **CONST** jest często używany do zapamiętania adresów zmiennych, które znajdują się w dalekich segmentach danych. Sposób ten został przedstawiony w przykładzie.

PATRZ

.CODE, .DATA, .STACK
.MODEL

PRZYKŁAD

Przykład pokazuje sposób odwołania się do dalekiej danej, zdefiniowanej w innym module źródłowym.

```
; Plik: DATAFILE.ASM
; w pliku są zawarte definicje danych, z których
; korzysta plik z kodem programu
        .MODEL LARGE
        PUBLIC MesgStr    ; Uwidocznienie zmiennej
                          ; w innych modułach
        .FARDATA
MesgStr DB   "This is far data", 0Dh, 0Ah, '$'
END
; Plik: USEDATA.ASM
; Poniższy kod korzysta z danych zadeklarowanych
; w pliku DATAFILE.ASM
        .MODEL LARGE
```

GROUP

```
EXTRN MesgStr:BYTE          ; Definicja znajduje się
                            ; w innym pliku
        .CONST
MSPtr   DD MesgStr          ; Tutaj umieść adres zmiennej
                            ; MesgStr
        .CODE
Start:
        mov   ax, @Data     ; Pobierz adres segmentu danych
        mov   ds, ax        ; i skopiuj go do DS
        lds   dx, MSPtr     ; Pobierz adres zmiennej MesgStr
                            ; i umieść go w DS:DX
        mov   ah, 9         ; Funkcja, która wyświetla
                            ; ciąg znaków
        int   21h           ; Wywołanie MS-DOS-a
        mov   ah, 4ch       ; Funkcja, która kończy program
        int   21h
        .STACK   512
END     Start
```

Oba pliki należy najpierw skompilować, a następnie skonsolidować komendą:

```
tlink usedata datafile
```

GROUP

ZASTOSOWANIE

Połączenie kilku segmentów w grupę. Może być ona używana w programie jako pojedynczy segment.

SKŁADNIA

Tryb MASM:

```
Nazwa_grupy GROUP seg1 [,seg2]...
```

Tryb IDEAL:

```
GROUP Nazwa_grupy seg1 [,seg2]...
```

KOMENTARZ

Grupy są konglomeratami kilku różnych segmentów. Pozwala to na odwoływanie się do znajdujących się w nich danych bez konieczności zmiany wartości rejestru segmentowego. Ponieważ grupa jest traktowana jako pojedynczy segment, to jej łączny rozmiar nie może przekroczyć 64k.

PATRZ

SEGMENT
ASSUME

PRZYKŁAD

Uproszczone dyrektywy segmentowe powodują połączenie kilku segmentów w grupę zwaną **DGROUP**. Poniższy przykład używa do tego celu dyrektyw standardowych.

```
_TEXT        SEGMENT     WORD PUBLIC 'CODE'
_TEXT        ENDS
FAR_DATA     SEGMENT     PARA PRIVATE 'FAR_DATA'
FAR_DATA     ENDS
FAR_BSS      SEGMENT     PARA PRIVATE 'FAR_BSS'
FAR_BSS      ENDS
_DATA        SEGMENT     WORD PUBLIC 'DATA
_DATA        ENDS
CONST        SEGMENT     WORD PUBLIC 'CONST'
CONST        ENDS
_BSS         SEGMENT     WORD PUBLIC 'BSS'
_BSS         ENDS
STACK        SEGMENT     PARA STACK 'STACK'
STACK        ENDS
DGROUP GROUP       _DATA, CONST, _BSS, STACK
```

.MODEL, MODEL

ZASTOSOWANIE

Wybranie modelu pamięci.

SKŁADNIA

```
.MODEL Model_pamięci [, język]
.MODEL TPASCAL
```

KOMENTARZ

Istnieje sześć modeli pamięci: **TINY**, **SMALL**, **MEDIUM**, **COMPACT**, **LARGE** i **HUGE**. Poza modyfikowaniem deklaracji segmentów, dyrektywa ta ustawia również wartości zmiennych **@CodeSize** i **@DataSize**.

Pole język poleca asemblerowi użyć konwencji wywoływania funkcji charakterystycznej dla danego języka. Tymi językami mogą być **C**, **Pascal**, **BASIC**, **FORTRAN** czy **Prolog**. Słowo **TPASCAL** poleca asemblerowi używać konwencji stosowanej w języku Turbo Pascal.

SEGMENT

PATRZ

.CODE, .DATA, .STACK

PRZYKŁAD

Patrz przykład do **.CODE**.

SEGMENT

ZASTOSOWANIE

Deklaracja segmentu.

SKŁADNIA

Tryb MASM:

```
Nazwa_segmentu SEGMENT [rozmieszczenie] [połączenie]
[rozmiar] ['nazwa_klasy']
```

Tryb IDEAL:

```
SEGMENT Nazwa_segmentu [rozmieszczenie] [połączenie]
[rozmiar] ['nazwa_klasy']
```

KOMENTARZ

Pole rozmieszczenie określa granicę pamięci, od której będzie zaczynał się segment. Możliwe typy rozmieszczeń przedstawione są w tabeli 8-3.

Tabela 8-3. Typy rozmieszczenia segmentów.

Typ	Opis
BYTE	Początek nowego segmentu rozpocznie się od następnego bajtu
WORD	Początek nowego segmentu rozpocznie się od następnego słowa o adresie parzystym
DWORD	Początek nowego segmentu rozpocznie się od następnego słowa o adresie podzielnym przez 4
PARA	Początek nowego segmentu będzie miał najbliższy wolny adres podzielny przez 16
PAGE	Początek nowego segmentu będzie miał najbliższy wolny adres podzielny przez 256

Tabela 8-4. Typy połączeń segmentów.

Typ	Opis
AT adres	Segment rozpoczyna się od podanego adresu.
COMMON	Wszystkie segmenty o tej samej nazwie umieszczane są od tego samego adresu.
MEMORY	Sklejenie segmentów, które mają identyczne nazwy.
PUBLIC	Znaczenie takie samo jak MEMORY.
PRIVATE	Segmenty nie są w ogóle łączone.
STACK	Sklejenie segmentów, które mają identyczne nazwy. Ustawienie rejestrów SS i SP tak, by wskazywały koniec powstałego segmentu.

Pole rozmiar jest używane w trybie chronionym procesora 80386. Ma ono wartość USE16 dla segmentów, które używają 16-bitowych przemieszczeń i **USE32** dla tych, które używają przemieszczeń 32-bitowych.

Ostatnim polem jest pole klasy. Jej nazwa musi być objęta parą cudzysłowów. Pole to jest używane do sterowania rozmieszczeniem segmentów w pamięci. Segmenty, które mają tą samą klasę zostaną umieszczone w pamięci obok siebie.

PATRZ

ASSUME
ENDS

PRZYKŁAD

Patrz przykład do **GROUP**.

.SEQ

Patrz opis dyrektywy **.ALPHA**.

.STACK, STACK

ZASTOSOWANIE

Dyrektywa uproszczona deklarująca segment stosu.

SKŁADNIA

```
.STACK [rozmiar_stosu]
```

UDATASEG

KOMENTARZ

Rozmiar_stosu określa liczbę bajtów zarezerwowanych na stos. Należy podać taki rozmiar, by nie nastąpiło przepełnienie stosu, nawet przy najbardziej zagnieżdżonych wywołaniach. Jeśli rozmiar nie zostanie podany, zostanie przyjęta liczba 1024.
Użycie dyrektywy **.STACK** zapewnia, że po uruchomieniu, rejestry SS i SP będą wskazywały na koniec stosu.

PATRZ

.CODE, .DATA, .CONST

PRZYKŁAD

Patrz przykład do **.CODE**.

UDATASEG

Patrz opis dyrektywy **DATA?**.

UFARDATA

Patrz opis dyrektywy **FARDATA?**.

Rozdział 9 Definicje zmiennych i ich rozmieszczenie w pamięci

W wielu programach dane, używane przez program, są co najmniej tak ważne jak kod, który na nich operuje. Programy graficzne używają dużej ilości pamięci do zapamiętania wycinków ekranu. Z dużej ilości danych korzystają również bazy danych, procesory tekstów czy arkusze kalkulacyjne. Podstawą programowania obiektowego jest fakt sterowania przez dane wykonywaniem się programu. W Turbo Assemblerze zapewniono wiele mechanizmów operowania danymi.
Dyrektywy Turbo Assemblera, używane do deklaracji danych, odwzorowują wiele cech języków wysokiego poziomu. Zapewniony został mechanizm budowania z niewielkiego zbioru podstawowych typów danych złożonych struktur. Pozwala on stworzyć dowolną strukturę, wymaganą w programie zamiast operowania w nim na prostych typach danych. Przewagą struktur danych jest ponadto większa przejrzystość i jasność pisanych programów.
W pierwszej części rozdziału zostały opisane podstawowe typy danych, które są wykorzystywane przez asembler. Przedstawione zostały sposoby definiowania zmiennych, używania etykiet i innych symboli. Następna część zawiera opis dyrektyw, które pozwalają na tworzenie bardziej złożonych struktur danych. i na koniec przedstawiono instrukcje sterujące rozmieszczeniem danych w pamięci.

Definicje zmiennych

Zwykle dane są ciągiem, składającym się z jednego lub kilku bajtów. Jednak o wiele wygodniej jest traktować je na trochę wyższym poziomie abstrakcji, opartym na ich przeznaczeniu i wykorzystaniu. Podstawowe typy danych używane w Turbo Assemblerze to zmienne numeryczne, teksty i wskaźniki.
Dane numeryczne przechowują liczby, a teksty - ciągi znaków. Wskaźniki są również liczbami, ale specjalnego typu. Wskazują one bowiem określone miejsca w pamięci. Każdy z tych typów zawiera kilka odmian. Na przykład, dane numeryczne mogą być przechowywane w różnej ilości bajtów (typowo jest to 1, 2 lub 4 bajty); łańcuchy mogą być zakończone bajtem zerowym lub znakiem dolara ($), albo zaczynać się bajtem, który zawiera jego długość; wskaźniki mogą być natomiast bliskie lub dalekie.

Dyrektywy deklarujące dane

Definiując daną informuje się asembler o konieczności zarezerwowania dla niej miejsca w pamięci. Tabela 9-1 podaje dyrektywy, które służą do tego celu.

= Tabela 9-1. Typy definicji danych

Dyrektywa	Opis
DB	Definicja danej o rozmiarze jednego bajtu

Dyrektywa	Opis
DD	Definicja podwójnego słowa (cztery bajty)
DF	Definicja wskaźnika danych w 32-bitowym segmencie procesora 80386 Dyrektywa ta posiada opcjonalne pole, które identyfikuje typ wskazywanych danych. Format dyrektywy jest następujący: [etykieta] DF [typ PTR] dana, [,dana]...
DP	Identyczna z dyrektywą DF
DQ	Definicja poczwórnego słowa (8 bajtów)
DT	Definicja dziesięciobajtowego obszaru danych. Liczby, używane do inicjalizacji tego typu danych, mogą być zapisane w kodzie binarnym kodowanym dziesiętnie lub też w formacie zmiennoprzecinkowym, używanym przez koprocesor matematyczny.
DW	Definicja słowa

Dane, utworzone tymi dyrektywami, nie są dostępne, jeżeli nie jest znany ich adres. Sposobem na jego wyznaczenie jest dołączenie przed definicją zmiennej etykiety:

```
OneByte     DB    1        ; Stwórz jednobajtową zmienną
                           ; o nazwie OneByte i umieść
                           ; w niej wartość 1
MyWord      DW    ?        ; Stwórz zmienną o rozmiarze słowa
                           ; i nazwie MyWord, ale nie
                           ; inicjuj jej
MWPoint     DD    MyWord   ; Stwórz zmienną o rozmiarze
                           ; dwóch słów i nazwie MWPoint
                           ; i przypisz jej adres
                           ; zmiennej MyWord
```

Turbo Assembler przechowuje nazwę, położenie i rozmiar każdej zdefiniowanej w ten sposób etykiety. Jeśli w programie zostanie wykorzystana jej nazwa (tak jak np. W definicji **MWPoint**), to asembler zastąpi ją w pliku wynikowym adresem.
Każda dyrektywa może być użyta na kilka sposobów. Na przykład, można zdefiniować kilka danych jednocześnie:

```
WordLst     DW    10, 15, 21, 15    ; WordLst wskazuje na
                                    ; pierwsze ze słów
                                    ; na liście
```

Operator **DUP** pozwala na deklarowanie tablic danych, bez konieczności wymieniania ich wszystkich elementów na liście:

```
BigList     DB    128 DUP ( 0 )     ; Stwórz tablicę
                                    ; wypełnioną 128 zerami
```

Znak zapytania (?), użyty w definicji danej, oznacza, że dana ta nie zostaje zainicjowana żadną wartością:

```
UnInit     DW    107 DUP ( ? )     ; Utwórz nie zainicjowaną
                                   ; tablicę, która składa
                                   ; się ze 107 słów
```

W wielu innych wypadkach zastąpienie liczby etykietą jest wygodne. Na przykład, w programie, który rysuje na ekranie ramki, okaże się konieczne podanie w kilku miejscach rozmiarów ekranu. Jeśli zaś program zostanie uruchomiony z ekranem o innych wymiarach, to konieczne będzie odszukanie wszystkich tych miejsc i poprawienie wymiarów ekranu. O wiele prościej byłoby zdefiniować etykiety, które reprezentowałyby rozmiary ekranu. Służy do tego dyrektywa **EQU**. Etykieta, występująca po jej lewej stronie, jest równoważna wartości podanej po jej prawej stronie.

Istnieje jeszcze inny sposób utworzenia symbolu, zastępującego dane, podobny do zastosowania dyrektywy EQU. Do przypisania do symbolu danej numerycznej używa się w nim znaku równości (=). Nie może być on zastosowany w odniesieniu do ciągów znaków, ale ma tą zaletę, że wartość tak zdefiniowanego symbolu może być zmieniana. Na przykład:

```
FieldSize = 20          ; Wszystkie odwołania do symbolu
                        ; FieldSize będą zastępowane
                        ; liczbą 20
   .
   .
FieldSize = 32          ; Teraz wszystkie odwołania do
                        ; symbolu FieldSize będą
                        ; zastępowane liczbą 32
   .
   .
FieldSize = FieldSize + 10
```

Poniższy przykład używa dyrektywy EQU i dyrektyw, które definiują dane przy rysowaniu ramki na ekranie.

```
; Plik BOX.ASM
; Program rysuje dookoła ekranu ramkę
          .MODEL    SMALL
ScrnBase  EQU   0B800h        ; Adres pamięci ekranu
                              ; (dla kart
                              ; monochromatycznych 0B000h)
TopLine   EQU   0             ; Numer górnej linii ramki
BotLine   EQU   24            ; Numer dolnej linii ramki
LeftCol   EQU   0             ; Numer lewej kolumny ramki
RightCol  EQU   79            ; Numer prawej kolumny ramki
ScrnWid   EQU   80            ; Szerokość ekranu
UpLeft    EQU   0C9h          ; Znak w lewym,
                              ; górnym rogu ramki
UpRight   EQU   0BBh          ; Znak w prawym,
                              ; górnym rogu ramki
LowLeft   EQU   0C8h          ; Znak w lewym,
                              ; dolnym rogu ramki
```

```
LowRight  EQU   0BCh          ; Znak w prawym,
                               ; dolnym rogu ramki
Horz      EQU   0CDh          ; Znak w linii poziomej
Vert      EQU   0BAh          ; Znak w linii pionowej
Attrib    EQU   07h           ; Atrybuty znaków
CallDos   EQU   <int 21h>     ; Stwórz symbol zastępujący
                               ; instrukcję wywołującą MS-DOS
          .DATA
LeftAt    DW    0             ; Początek każdej linii
          .CODE
Start:
          mov   ax, ScrnBase  ; Pobierz adres segmentu
                               ; ekranu
          mov   es, ax        ; i umieść go w ES
          mov   ax, Topline   ; Pobierz numer górnej
                               ; linii ramki
          mov   bl, ScrnWid   ; Pomnóż ją przez
                               ; szerokość ekranu
          mul   bl
          add   ax, LeftCol   ; Dodaj numer lewej
                               ; kolumny
          shl   ax, 1         ; Pomnóż przez dwa
          mov   LeftAt, ax    ; i zachowaj w zmiennej
          mov   di, ax        ; oraz prześlij do DI
          mov   al, UpLeft    ; Pobierz znak
                               ; lewego-górnego rogu
          mov   ah, Attrib    ; Pobierz atrybuty
          stosw               ; Narysuj róg
          mov   al, Horz      ; Pobierz znak linii
                               ; poziomej
          mov   cx, RightCol  ; Pobierz numer prawej
                               ; kolumny
          sub   cx, LeftCol   ; Po odjęciu numeru lewej
                               ; otrzymujemy szerokość
                               ; ramki
          sub   cx, 2         ; Korekta o 2 ze względu
                               ; na krawędzie
          rep   stosw         ; Rysuj górną linię
          mov   al, UpRight   ; Pobierz znak
                               ; prawego-górnego rogu
          stosw               ; i wyświetl go
          mov   cx, Botline   ; Pobierz numer ostatniej
                               ; linii
          sub   cx, TopLine   ; Po odjęciu numeru górnej
                               ; otrzymujemy wysokość ramki
          sub   cx, 2         ; Korekta o 2 ze względu
```

```
                                ; na krawędzie
SideLp:
        push  cx                ; Zachowaj ją
        mov   di, ScrnWid       ; Pobierz szerokość
                                ; ekranu
        shl   di, 1             ; Pomnóż ją przez 2
        add   di, LeftAt        ; Następna linia
        mov   LeftAt, di        ; Zapisz jej adres
        mov   al, Vert          ; Pobierz znak linii
                                ; pionowej
        mov   ah, Attrib        ; Pobierz atrybuty
        stosw                   ; Narysuj znak linii
        mov   al, ' '           ; Wnętrze ramki wypełnij
                                ; spacjami
        mov   cx, RightCol      ; Pobierz numer prawej kolumny
        sub   cx, LeftCol       ; Po odjęciu numeru lewej
                                ; otrzymujemy szerokość ramki
        sub   cx, 2             ; Korekta o 2 ze względu
                                ; na krawędzie
        rep   stosw             ; Rysuj wnętrze ramki
        mov   al, Vert          ; Pobierz znak linii
                                ; pionowej
        stosw                   ; i wyświetl go
        pop   cx                ; Odtwórz stan licznika
        loop  SideLp            ; Rysuj kolejną linię
        mov   di, ScrnWid       ; Pobierz szerokość ekranu
        shl   di, 1             ; Pomnóż ją przez 2
        add   di, LeftAt        ; Następna linia
        mov   al, LowLeft       ; Pobierz znak
                                ; lewego-dolnego rogu
        mov   ah, Attrib        ; Pobierz atrybuty
        stosw                   ; Wyświetl róg
        mov   al, Horz          ; Pobierz znak linii
                                ; poziomej
        mov   cx, RightCol      ; Pobierz numer prawej
                                ; kolumny
        sub   cx, LeftCol       ; Po odjęciu numeru lewej
                                ; otrzymujemy szerokość ramki
        sub   cx, 2             ; Korekta o 2 ze względu
                                ; na krawędzie
        rep   stosw             ; Rysuj dolną linię
        mov   al, LowRight      ; Pobierz znak
                                ; prawego-dolnego rogu
        stosw                   ; i wyświetl go
        mov   ah, 4Ch           ; Funkcja kończąca program
        CallDos
```

```
            .STACK  512
END         Start
```

Jedyną rzeczą, jaką należy zrobić, by program pracował poprawnie z kartą EGA na ekranie z 43 liniami, jest zmiana wartości etykiety na 42. Jest to o wiele prostsze, niż szukanie wszystkich miejsc programu, w których została użyta wartość, która mówi o numerze dolnej linii ekranu.

DEFINIOWANIE DANYCH NUMERYCZNYCH

W przykładzie, gdzie rysujemy ramkę, większość liczb została podana w formacie szesnastkowym. Mówi o tym litera 'h', która pojawia się za liczbą. Bez niej dana byłaby potraktowana jako liczba dziesiętna (podstawa standardowa). Ponadto można również używać liczb w kodzie binarnym (przyrostek 'b') lub ósemkowym (przyrostek 'o').
Dyrektywą **.RADIX** lub **RADIX** można zmienić domniemaną podstawę liczenia. Jej argument musi być równy 2, 8, 10 lub 16 odpowiednio dla liczb dwójkowych, ósemkowych, dziesiętnych i szesnastkowych. Liczba zapisana w tej podstawie nie musi mieć dołączanego na końcu żadnego znaku. Dla pozostałych nadal jest konieczne podanie odpowiedniego przyrostka, nie wyłączając liczb dziesiętnych (przyrostek 'd').

DEFINIOWANIE DANYCH TEKSTOWYCH

Ciąg bajtów, którym zostały przypisane wartości kodu ASCII, nazywany jest tekstem. Najprostszym jego użyciem jest wyświetlenie na ekranie. Tekst może też zostać wykorzystany jako etykiety do danych, znaczniki wyrażeń warunkowych lub jako informacja o plikach.
W Turbo Assemblerze istnieje kilka sposobów na ograniczenie łańcucha znaków. Wybranie jednego z nich zależy od kontekstu, w którym łańcuch ma być użyty. Na przykład, jeśli ma on zostać zapisany do pamięci, to można ograniczyć go apostrofem (') lub cudzysłowem ("). Użyta może również zostać dyrektywa **DB**:

```
MyStr     DB    "Oto ciąg znaków", 0
```

Gdy łańcuch ma zostać użyty w dyrektywie **EQU**, to powinien zostać ujęty w nawiasy trójkątne (<łańcuch>). W naszym przykładzie, etykieta **CallDos** jest równoważna tekstowi int 21h:

```
CallDos   EQU   <int  21h>
```

DEFINIOWANIE WSKAŹNIKÓW

Wskaźniki są wygodnym narzędziem do operowania na dużych strukturach danych. Na przykład, program przekazując do procedury łańcuch, który jest jej argumentem mógłby go przekopiować do innego obszaru pamięci; mógłby też po prostu przekazać adres łańcucha w rejestrze czy poprzez stos. Podobnie można postąpić z wskaźnikami do tablic czy innych struktur.
Jeśli wszystkie dane, identyfikowane przez wskaźniki, znajdują się w tym samym segmencie, to konieczne jest tylko pamiętanie ich przemieszczenia (wskaźniki typu NEAR). Jeśli zaś używane jest kilka segmentów, to wskaźnik musi zawierać zarówno numer segmentu, jak i przemieszczenie danej (wskaźniki typu FAR).

By zdefiniować wskaźnik bliski NEAR, należy użyć dyrektywy **DW**, i dyrektywy **DD** dla wskaźnika typu FAR. Argumentem obu dyrektyw może być nazwa etykiety, konkretny adres lub dana, zdefiniowana w dalszej części programu. Sposób definiowania wskaźników ilustruje poniższy przykład.

```
          .MODEL   SMALL
          .DATA
Array     DW   16 DUP (?)    ; Nie zainicjowana
                             ; tablica 16 słów
Ptr1      DW   Array         ; Tutaj zostanie umieszczone
                             ; przemieszczenie tablicy Array
Ptr       DD   0040001Eh     ; Daleki wskaźnik na bufor
                             ; klawiatury
Ptr       DW   ?             ; Wskaźnik nie zainicjowany
          .CODE
Func1     PROC
          mov   ax, SEG DGROUP
          mov   ds, ax
          les   si, Ptr2     ; Załadowanie wskaźnika bufora
                             ; klawiatury do
                             ; rejestrów ES i SI
          mov   bx, Ptr1     ; Wskaźnik tablicy Array w BX
          mov   cx, 16       ; Rozmiar tablicy
CpLoop:
          mov   ax, es:[si]  ; Pobierz słowo
          mov   [bx], ax     ; Umieść je w tablicy
          add   bx, 2        ; Zwiększ wskaźniki
          add   si, 2
          loop  CpLoop
          mov   Ptr3, bx     ; Prześlij wskaźnik końca
                             ; tablicy do zmiennej Ptr3
          ret
Func1     ENDP
```

Wyrażenia

Wyrażenia to zestaw danych, połączonych operatorami np. operatorem dodawania. Opisane są one w tabeli 9-2. W zasadzie, wyrażenie może być użyte tam, gdzie dozwolone jest użycie pojedynczej danej. Na przykład, jeśli istnieje symbol NumRows, który oznacza liczbę wierszy na ekranie i TopRow, który zawiera numer najwyższej linii ekranu, to do wyznaczenia numeru ostatniej linii na ekranie można użyć wyrażenia TopRow + NumRows -1. Ważne jest aby pamiętać, że wartość wyrażeń jest ustalana w czasie kompilacji zbiorów, a nie w czasie działania programu. Dlatego też nie można w nich używać danych, które znajdują się w pamięci lub rejestrach.

= **Tabela 9-2. Operatory**
Priorytet

Operator	MASM	IDEAL	Opis
LENGTH	13	12	Zwraca liczbę elementów danego typu
MASK	13	12	Zwraca maskę bitową rekordu
SIZE	13	12	Zwraca rozmiar zmiennej w bajtach
WIDTH	13	12	Zwraca szerokość pola w rekordzie
.	12	2	Selektor pola struktury
HIGH	11	11	Zwraca starszą część wyrażenia
LOW	11	11	Zwraca młodszą część wyrażenia
+,-	10	10	Operatory znaku liczby
:	9	3	Zmiana segmentu
OFFSET	8	12	Zwraca przemieszczenie zmiennej
PTR	8	1	Ustalenie rozmiaru wyrażenia
SEG	8	12	Zwraca numer segmentu zmiennej
TYPE	8	NA	Zwraca typ symbolu
*,/	7	9	Operatory mnożenia,dzielenia
MOD	7	9	Dzielenie modulo
SHL	7	9	Przesunięcie w lewo
SHR	7	9	Przesunięcie w prawo
+,-	6	8	Dodawanie, odejmowanie
EQ	5	7	Równość
GE	5	7	Większe lub równe
GT	5	7	Większe
LE	5	7	Mniejsze lub równe
LT	5	7	Mniejsze
NE	5	7	Nierówne
NOT	4	6	Negacja
AND	3	5	Iloczyn bitów
OR	2	4	Suma bitów
XOR	2	4	Suma wyłączająca bitów
LARGE	1	1	Konwersja wyrażenia do 32 bitów
SHORT	1	1	Potraktowanie wyrażenia jako wskazania krótkiego
SMALL	1	1	Konwersja wyrażenia do 16 bitów

WYRAŻENIA NUMERYCZNE

Wyrażenia numeryczne są obliczane od lewej do prawej strony, z uwzględnieniem nawiasów i priorytetów operatorów. Operatory, o wyższym priorytecie, są wykonywane jako pierwsze. Większość z nich jest używana również w językach wysokiego poziomu np. +,-,/ czy *. Działanie innych, jak np. **SHL**, **SHR**, **AND**, **XOR** odpowiada działaniu instrukcji

asemblera. Operatory porównania **GT**, **EQ**, czy **LT** zwracają wartość 1, gdy porównanie jest prawdziwe, a 0 w przeciwnym wypadku.

WYRAŻENIA TEKSTOWE

Operatory tekstowe służą do manipulowania ciągami znaków. Część z nich nie jest bezpośrednio używana w wyrażeniach tekstowych. Na przykład, argumentem dyrektywy **SIZESTR** jest łańcuch, ale zwraca ona już wartość numeryczną (długość łańcucha):

```
StrLen    SIZESTR    <ABC>    ; StrLen = 3
```

Inne działają wyłącznie na tekście. Są to: **CATSTR** do sklejania łańcuchów, **INSTR** do wyszukania jednego łańcucha w drugim oraz **SUBSTR** do pobrania fragmentu łańcucha. Sklejenie łańcuchów powoduje utworzenie jednego poprzez umieszczenie łańcuchów w pamięci obok siebie. Oczywiście, do tak powstałego łańcucha można się odwoływać za pomocą odpowiadającej mu etykiety. W poniższym przykładzie zostają połączone łańcuchy <cats>, < and > oraz <dogs>:

```
tmp      CATSTR    <cats>, < and >    ; tmp = <cats and >
result   CATSTR    tmp, <dogs>        ; result = <cats and
dogs>
```

Dyrektywa **SUBSTR** powoduje pobranie z łańcucha pewnej jego części. Jednym z jej argumentów jest pozycja wycinanego łańcucha. Przy jej określaniu może się przydać dyrektywa **INSTR**. Zwraca ona położenie podanego podciągu znaków w łańcuchu. W poniższym przykładzie dyrektywy te są wykorzystane do pobrania fragmentu łańcucha fname, który znajduje się przed podłańcuchem <_FUNC>:

```
fname    EQU    <NEXT_FUNC>      ; Przypisanie fname łańcucha
fpos     INSTR fname, <_FUNC>    ; Pobierz pozycję łańcucha
                                 ; _FUNC (3)
result   SUBSTR fname, 0, fpos   ; Skopiuj od początku do
                                 ; pozycji fpos
                                 ; (result=<NEXT>)
```

)PERACJE NA WSKAŹNIKACH

Etykiety w wyrażeniach są traktowane jako wskaźniki. Wartością etykiety jest adres danej, którą ona wskazuje. W wyrażeniach do etykiet może zostać zastosowany dowolny z operatorów arytmetycznych. Na przykład, by obliczyć liczbę bajtów pomiędzy dwoma etykietami można dokonać odjęcia ich wartości:

```
String1    DB    "Oto ciąg znaków", 0   ; Definicja łańcucha
Length1    DW    Length1 - String1      ; Obliczenie jego
                                        ; długości
```

Gdy etykieta jest używana jako argument instrukcji, to asembler przyjmuje, że odwołanie będzie realizowane pod ten adres. Czasami jednak konieczne jest załadowanie właśnie tego adresu, a nie umieszczonej tam danej. Wtedy należy użyć dyrektyw **OFFSET** i **SEG**.

```
.MODEL   SMALL
.FAR_DATA
```

```
Array   DB   1, 2, 3, 4, 5, 6
        .CODE
        .
        .
        .
        mov   ax, SEG Array
        mov   es, ax
        mov   ax, OFFSET Array
        .
        .
        .
```

Struktury danych

Struktura to typ łączący w sobie zmienne kilku różnych typów. Na przykład, program przechowuje w pamięci adresy zmiennych. Na rysunku 9-1 przedstawiony został sposób umieszczania danych w strukturach.

Bajt 0	1	2	3
Młodszy bajt przemieszczenia	Starszy bajt przemieszczenia	Młodszy bajt segmentu	Starszy bajt segmentu

Rysunek 9-1. ***Format wskaźnika w pamięci.***

Instrukcje LES czy LDS potrafią odczytać daną, zdefiniowaną jako podwójne słowo. Ale jak dostać się tylko do jednej z części adresu: przemieszczenia lub numeru segmentu? Jednym ze sposobów jest użycie operatora PTR:

```
mov   ax, WORD PTR ptr1     ; Załaduj przemieszczenie
mov   bx, WORD PTR ptr1+2   ; Załaduj numer segmentu
```

Operator **WORD PTR** nakazuje asemblerowi zignorować definicję zmiennej **ptr1** jako podwójne słowo, a potraktować ją jako definicję pojedynczego słowa.
Oczywiście sposób ten jest poprawny i wiele programów jest tak pisanych, ale są one wtedy trudne do analizy. Poza tym pojawia się problem, gdy należy taki program przekształcić tak, by wykorzystywane były w nim 32-bitowe segmenty procesora 80386. Wtedy w całym programie wszystkie odwołania typu ptr1+2 musiałyby zostać zamienione na ptr1+4. Użycie struktur danych zapewnia powstanie łatwiejszego do zrozumienia programu, jak również ułatwia dokonywanie jego późniejszych modyfikacji.

Deklarowanie struktur danych

By rozwiązać problem ze wskaźnikami, należałoby zdefiniować nowy typ o nazwie *POINTER*, który zawierałby pola, przeznaczone na numer segmentu i przemieszczenie. Dyrektywa **STRUCT** pozwala na zdefiniowanie pól struktury. W poniższym przykładzie została zadeklarowana struktura *POINTER*:

```
POINTER     STRUCT
PtrOff      DW    0        ; Pole, które przechowuje
                           ; przemieszczenie
PtrSeg      DW    ?        ; Pole, które przechowuje
                           ; numer segmentu
POINTER     ENDS
```

Ten nowo powstały typ może być używany na równi z typami, już istniejącymi, np deklarowanymi dyrektywami **DB** lub **DW**. Oto kilka typowych deklaracji, które wykorzystują typ POINTER:

```
Ptr1    POINTER    <OFFSET msg1, SEG msg1>   ; Przechowanie
                                             ; wskaźnika
                                             ; zmiennej msg1
Ptr2    POINTER    < , 0B800h>               ; Użycie
                                             ; przemieszczenia
                                             ; domniemanego
                                             ; i segmentu
                                             ; o numerze
                                             ; 0B800h
Ptr3    POINTER    10 dup (<0,0>)            ; Utworzenie
                                             ; tablicy
                                             ; dziesięciu
                                             ; wskaźników
                                             ; zainicjowanych
                                             ; na 0:0
```

Przy deklaracji zmiennej **Ptr2** pole przemieszczenia nie zostało jawnie podane. Oznacza to przyjęcie dla niego wartości domniemanej. Wartość ta określona jest w definicji. W tym wypadku jest ona równa 0.

Odwoływanie się do pól struktury

Po zdefiniowaniu struktur i zadeklarowaniu danych, potrzebny jest tylko sposób na odwoływanie się do nich. Realizowany jest on operatorem kropki (.). Po lewej jego stronie zależy umieścić nazwę zmiennej, a po prawej nazwę pola występującego w niej. Poprzedni przykład mógłby więc wyglądać następująco:

```
mov    ax, Ptr1.PtrOff    ; Załaduj przemieszczenie
mov    bx, Ptr1.PtrSeg    ; Załaduj numer segmentu
```

Turbo Assembler przechowuje wszystkie etykiety w tablicy symboli. Adres struktury zawarty jest natomiast w etykiecie **Ptr1**. Etykiety **PtrOff** i **PtrSeg** są traktowane jako przemieszczenia względem **Ptr1**. W tym wypadku **PtrOff** równa się 0, a **PtrSeg** wynosi 2. Operator kropki powoduje po prostu dodanie przemieszczeń, które występują po jego obu stronach.

Operator kropki może być również zastosowany do dodania przemieszczenia do rejestru bazowego lub indeksowego, podobnie jak do etykiet struktur. Poniższy przykład pokazuje, w jaki sposób można odczytać pola struktur, które znajdują się w tablicy:

```
          .MODEL   SMALL
          .DATA
POINTER   STRUCT
PtrOff    DW   0
PtrSeg    DW   0
POINTER   ENDS
PtrArray  POINTER  10 dup (<>)       ; Utworzenie tablicy
                                     ; dziesięciu wskaźników
; Uaktualnij numery segmentów w tablicy. Rejestr BX
; zawiera liczbę stron, o którą każdy segment tablicy
; PtrArray został przesunięty.
          .CODE
FixSegs   PROC
          mov   si, OFFSET PtrArray ; Pobierz adres tablicy
; w następnej linii do ustalenia rozmiaru tablicy w bajtach
; zastosowany został operator SIZE. Następnie do ustalenia
; rozmiaru struktury został wykorzystany operator TYPE.
; Po wykonaniu dzielenia mamy obliczoną ilość elementów
; tablicy.
          mov   cx, ( SIZE PtrArray )/( TYPE POINTER )
PtrLoop:
          mov   ax, [si].PtrSeg      ; Pobierz do AX
                                     ; numer segmentu
          .
          .
          .
          add   si, TYPE POINTER     ; Następny element
                                     ; tablicy
          loop  PtrLoop
```

UNIE

Jednym z kłopotów, powstałych przy stosowaniu typu POINTER, zdefiniowanego w poprzednim przykładzie, jest rozwlekły kod używany do zainicjowania wskaźnika. Trzeba bowiem podać zarówno numer segmentu jak i przemieszczenie danej. Wyjściem z tej sytuacji jest zastosowanie unii. Unia jest bardzo podobna do struktury, z tym tylko, że jej pola nakładają się na siebie. Oznacza to, że do tego samego miejsca pamięci można się odwołać za pośrednictwem kilku symboli. Ważne jest to, że symbole te mogą mieć zupełnie różne typy. Definicja typu wskaźnikowego, używająca unii, może wyglądać następująco:

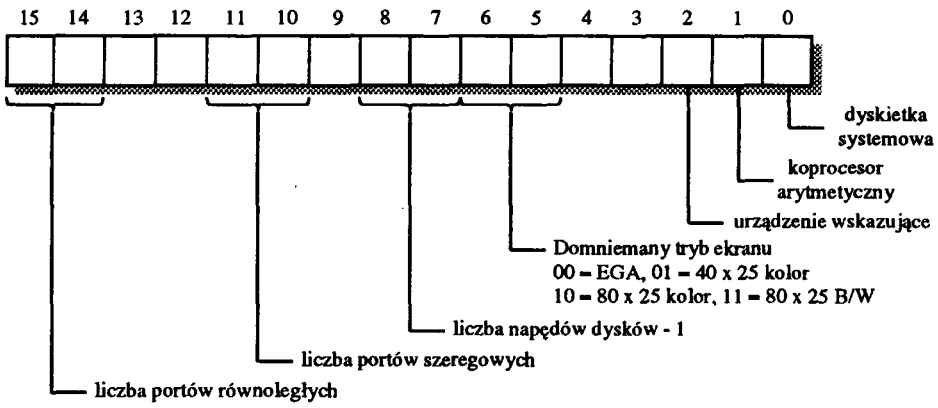
```
POINTER    UNION
           STRUCT            ; Jedną wersją stanowi wskaźnik
           PtrOff   DW  0    ; traktowany jako
                             ; przemieszczenie
           PtrSeg   DW  0    ; i numer segment
           ENDS
FarPtr     DD   0            ; Drugą wersją jest wskaźnik
```

```
                                        ; traktowany jako podwójne słowo
POINTER     ENDS
Ptr1    POINTER    <?,Mesg>             ; Inicjacja unii. Tylko jedno
                                        ; z pól musi zostać
                                        ; zainicjowane, pozostałe muszą
                                        ; zostać nie określone
```

Etykieta **Ptr1** zadeklarowana powyżej może być traktowana jako wskaźnik na podwójne słowo (**Ptr1.FarPtr**) lub na dwa oddzielne słowa (**Ptr1.PtrOff** i **Ptr1.PtrSeg**). Pozwala to zamiast inicjować po kolei wszystkie pola, zainicjować tylko jedno. W przykładzie tym pierwsze pole, składające się ze zmiennych **PtrOff** i **PtrSeg**, jest pozostawiane nie zdefiniowane. Drugiemu zaś jest przypisywane bezpośrednio dalekie wskazanie zmiennej **Mesg**.

POLA BITOWE

Pewne zmienne nigdy nie wykorzystują wszystkich ośmiu bitów bajtu. Liczba dysków w systemie rzadko przekracza cztery. Liczba ta może być zakodowana na trzech, a nawet, jeśli przyjmiemy, że w systemie istnieje co najmniej jeden, na dwóch bitach. Pozostałe sześć bitów może zostać wykorzystane w inny sposób. Rysunek 9-2 przedstawia sposób przechowywania danych o systemie przez BIOS w jednym słowie, które to słowomoże być przesłane do pojedynczego rejestru.



Rysunek 9-2. ***Format danych o systemie używany przez BIOS.***

Dyrektywa **RECORD** ułatwia operowanie na częściach słów lub bajtów. Rekordy są podobne do struktur, z tym że ich rozmiar nie przekracza bajtu lub słowa. Dyrektywa **RECORD** służy do ich definicji:

```
Nazwa    RECORD    pole:szerokość[=wyrażenie] [,pole:szero-
kość[=wyrażenie]]...
```

Etykieta **Nazwa** może być wykorzystana w programie do deklaracji zmiennych tego typu. Pola rekordu są podobne do pól struktury. Oznaczają one nie liczbę bajtów od początku struktury, a stanowią numer bitu w rekordzie. Argument szerokość oznacza rozmiar pola

w bitach, a wyrażenie określa domniemaną wartość pola. Utworzenie zmiennej, która zawiera dane o systemie wymaga następującej deklaracji:

```
          .MODEL   SMALL
          .DATA
ELREC     RECORD   Printers:2, Resrv:2, Coms:3, Resrv2:1,
Drives:2, VidMode:2, Resrv3:1, Point:1, Math:1, Boot:1
EquipLst  ELREC   ?
          .CODE
GetComs   PROC
          mov   ax, 40h        ; Segment danych BIOS-u
          mov   es, ax
          mov   ax, es:[10h]   ; Pobierz dane o systemie
          mov   EquipLst, ax   ; Prześlij je do pamięci
          and   ax, MASK Coms  ; Zamaskuj bity portów
                               ; szeregowych
          mov   cl, Coms       ; Pobierz licznik przesunięć
          shr   ax, cl         ; AX zawiera liczbę portów
                               ; szeregowych
          ret
GetComs   ENDP
END
```

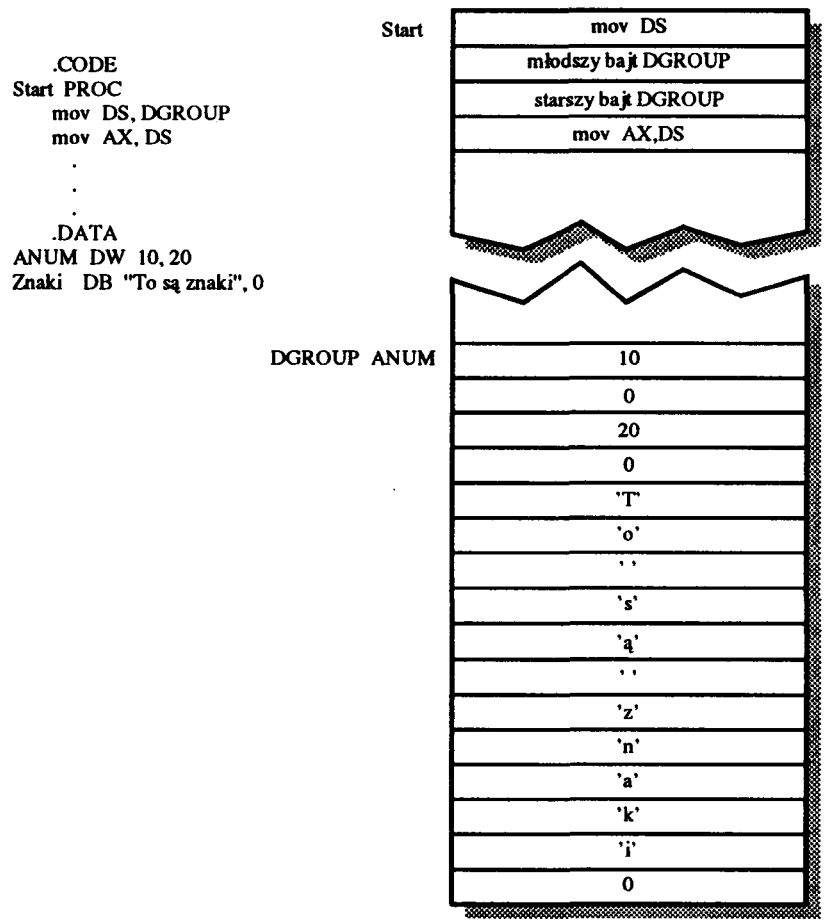
W tym przykładzie do wydzielenia bitów pola **Coms** został zastosowany operator **MASK**. W tym wypadku spowodował on utworzenie maski równej **0000111000000000b**, odpowiadającej temu polu. Następnie, po przesunięciu rezultatu w prawo, rejestr AX zawiera liczbę portów szeregowych w systemie. Etykieta **Coms** zawiera ilość potrzebnych przesunięć.

Umieszczanie danych w pamięci

Język asemblera w większym stopniu niż języki wysokiego poziomu zapewnia swobodę przy decydowaniu o rozłożeniu danych w pamięci. Zwykle dane są umieszczane w kolejności ich pojawienia się w pliku źródłowym. Ilustruje to rysunek 9-3.
Czasami jednak porządek ten nie spełnia pewnych żądanych kryteriów. Na przykład, programy typu .COM muszą zaczynać się 256 bajtów od początku segmentu kodu. Programy, które mają zostać umieszczone w pamięci ROM mogą wymagać umieszczenia pod szczególnymi adresami. Turbo Assembler posiada instrukcje, które pozwalają sterować rozłożeniem danych w pamięci.

Wskaźnik pozycji

W czasie kompilacji każdej linii programu Turbo Assembler umieszcza, wygenerowany na jej podstawie kod lub dane, w miejscu pamięci, wskazywanym przez wskaźnik pozycji. Wskaźnik ten jest następnie aktualizowany tak, by wskazywał następny wolny bajt. Program może odwołać się do niego używając znaku dolar ($).
Wskaźnik pozycji może być użyty do wyznaczenia rozmiaru fragmentu kodu lub danych. Ilustruje to poniższy przykład:



Rysunek 9-3. ***Rozmieszczenie danych i kodu w pamięci.***

```
Begin   DB   "Próbny ciąg znaków"  ; Etykieta ta stanowi
                                   ; początek elementu
                                   ; danych
Size    DW   $-Begin               ; Różnica stanowi
                                   ; jego rozmiar
```

Ustawianie wskaźnika pozycji

Wskaźnikowi pozycji może być przypisana wartość za pomocą dyrektywy **ORG**. Jej argument jest nowym przemieszczeniem, pod którym będą umieszczane generowane dane lub kod. Może być on adresem absolutnym np. **100h** lub może być podany względem bieżącego położenia np. **$+14**.

Jednym z zastosowań tej dyrektywy jest przesunięcie kodu i danych w plikach typu .COM. Służy ona także do umieszczenia etykiety lub zmiennej w obszarze wykorzystywanym przez np. MS-DOS lub BIOS. Poniższy przykład ilustruje oba zastosowania:

```
BIOS  SEGMENT   AT 40h ; Stwórz segment pod adresem 40h
          ORG   17h    ; Zmienna KeyStatus znajduje się
                       ; pod adresem 40:17
KeyStatus DB    ?
BIOS ENDS
COMSEG    SEGMENT PARA PUBLIC 'ALL'
          ASSUME CS:COMSEG, DS:COMSEG, ES:NOTHING, SS:COMSEG
          ORG   100h   ; Przemieszczenie dla plików
                       ; typu .COM
Start:
          jmp   Begin  ; Przeskocz dane
Vars      DB    0      ; Pliki typu .COM mają dane i kod
                       ; w tym samym segmencie
Begin:    .
          .
          .
COMSEG    ENDS
END  Start
```

Rozmieszczanie danych

Procesory 8086 do 80286 mają możliwość dostępu jednocześnie do 16 bitów pamięci. Gdy procesor pobiera słowo, które zaczyna się pod nieparzystym adresem, to tak naprawdę musi odczytać dwa słowa. Najpierw jest pobierane to, które zawiera młodszy bajt, a następnie to, zawierające bajt starszy. Oznacza to, że dostęp do danych pamięci jest o wiele szybszy, gdy znajdują się one pod adresami parzystymi. W procesorze 80386 istnieje podobna sytuacja, z tym, że pobiera on jednocześnie cztery bajty, tak więc najszybszy jest dostęp do danych, które zaczynają się na granicy podwójnego słowa. Rysunek 9-4 pokazuje różne sposoby rozmieszczania danych w pamięci.

Dyrektywa **EVEN** powoduje przesunięcie wskaźnika pozycji do następnego parzystego adresu. Bardzo podobna, ale bardziej elastyczna jest dyrektywa **ALIGN**. Jej argumentem jest granica, na której będą rozmieszczone dane. Musi być ona potęgą liczby 2 (2, 4, 8, 16 itd.). Następny przykład pokazuje zastosowanie dyrektyw **EVEN** i **ALIGN**.

```
Var1    DB    0        ; Rozmieszczenie nieznane
        EVEN
Var2    DW    ?        ; Umieszczona pod adresem parzystym
        ALIGN 256
Var3    DD    100      ; Umieszczona pod adresem podzielnym
                       ; przez 256
```

Aliasowanie nazw danych

Dyrektywa **LABEL** powoduje zdefiniowanie etykiety pod bieżącą pozycją, bez przydzielania dla niej pamięci. Jej argument mówi tylko o typie wskazywanej danej. Dostępne typy to: **NEAR**, **FAR**, **BYTE**, **WORD**, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE** lub nazwa struktury.

Adresy	Rozmieszczenie
0	Byte, Word, DWORD, QWORD
1	Byte
2	Byte, Word
3	Byte
4	Byte, Word, DWORD
5	Byte
6	Byte, Word
7	Byte
8	Byte, Word, DWORD, QWORD
9	Byte
10	Byte, Word
11	Byte
12	Byte, Word, DWORD
13	Byte
14	Byte, Word
15	Byte
16	Byte, Word, DWORD, QWORD, PARAGRAPH
17	Byte
18	Byte, Word
19	Byte

Rysunek 9-4. ***Typy rozmieszczenia danych w pamięci.***

Jednym z zastosowań dyrektywy **LABEL** jest tworzenie ciągów znaków stosowanych w języku Pascal. Zaczynają się one bajtem, który zawiera liczbę znaków w łańcuchu. Problem polega natomiast na tym, że przy definicji łańcucha konieczna jest znajomość jego długości. Można więc stworzyć etykietę, która wskazuje jego koniec, a następnie wykorzystywać ją w wyrażeniach, zwracających długość łańcucha:

```
         IDEAL; Tryb IDEAL
         DATASEG
PString  DB   PSEnd-PString, "Oto ciąg znaków"
PSEnd    LABEL BYTE
```

Podsumowanie

Pisząc poważniejszy program w języku asemblera, równie dużo czasu poświęca się pisaniu kodu, co tworzeniu struktur danych. Operowanie na nich może stać się prostsze dzięki stworzeniu jasnych, łatwo modyfikowalnych symboli, reprezentujących obiekty w programie.

Wszystkie złożone typy danych zbudowane są z typów podstawowych. Dane deklarowane dyrektywami **DB**, **DW**, **DD** itp. oraz łańcuchy i wskaźniki są najczęstszymi składowymi struktur i unii.

Struktury są używane do definiowania nowych typów danych i zgrupowania danych powiązanych ze sobą w pojedynczym bloku. Rekordy są rozszerzeniem pojęcia struktury i pozwalają tworzyć pola o szerokości kilku bitów. Stosowanie unii oraz użycie dyrektywy **LABEL** pozwala na przeprowadzenie kontroli typów, a to zmniejsza ilość błędów w programie.

AND

ZASTOSOWANIE

Logiczna operacja AND.

SKŁADNIA

`wyr1 AND wyr2` *Logiczna operacja AND na bitach rezultatów wyrażeń*

KOMENTARZ

Operator ten stosuje się w wyrażeniach, których wartość ma być wyliczona w trakcie procesu kompilacji. Mogą one wystąpić w tekście programu wszędzie tam, gdzie poprawne byłoby użycie dowolnej liczby.

PATRZ

MOD, SHL, SHR *Inne operatory używane w wyrażeniach*
NOT, OR, XOR *Pozostałe operatory logiczne*

PRZYKŁAD

W poniższym przykładzie został zaprezentowany sposób wykorzystania operatorów logicznych do operowania na maskach bitów.

```
          .MODEL  SMALL
Mask1     EQU     00001000b                 ; Maska bitu
                                            ; znacznika
Mask2     EQU     01000000b                 ; Inna maska
                                            ; znacznika
LowN      EQU     00001111b                 ; Maska dolnego
                                            ; półbajtu
          .DATA
FlagByte  DB    ?                           ; Testowany bajt
          .CODE
; Funkcja ta nie wykonuje w zasadzie żadnej operacji.
; Przeprowadza ona tylko kilka testów bitów.
DoFlags   PROC
          and     FlagByte, NOT Mask1       ; Wyzeruj bit
                                            ; znacznika
          test    FlagByte, Mask1 OR Mask2  ; Sprawdź czy
                                            ; którykolwiek ze
                                            ; znaczników jest
                                            ; ustawiony
          or      FlagByte, Mask1 AND LowN  ; Ustaw znacznik,
                                            ; jeśli znajduje
                                            ; się on w dolnym
                                            ; półbajcie
          and     FlagByte, (Mask1 AND LowN) XOR LowN
```

ALIGN

```
; Powyższa instrukcja zeruje cały starszy półbajt oraz bit
; znacznika, jeśli znajduje się on w półbajcie młodszym
          ret
DoFlags   ENDP
```

ALIGN

ZASTOSOWANIE

Przesunięcie wskaźnika pozycji do następnego adresu wyrównania.

SKŁADNIA

`ALIGN granica` *Granica musi być potęgą liczby dwa (2, 4, 8, itd.)*

KOMENTARZ

Efektywność pracy różnych procesorów rodziny 80x86 zależy od rozmieszczenia danych w pamięci. Umieszczenie ich w obszarach, odpowiadających rozmiarowi magistrali danych, powoduje bowiem szybszy dostęp do danych. Poza tym niektóre języki wysokiego poziomu umieszczają pewne dane pod specjalnie dobieranymi adresami. Dyrektywa **ALIGN** pozwala więc na zachowanie zgodności z nimi.

PATRZ

EVEN *Rozmieszczanie na adresach parzystych*

PRZYKŁAD

```
         .MODEL   SMALL
         .DATA
data1    DB   1       ; Dana specjalnie nie rozmieszczana
         ALIGN   4    ; Typowe rozmieszczenie przyjmowane
                      ; dla procesorów 80386 w celu
                      ; podniesienia szybkości ich działania
data2    DW   ?       ; Dana umieszczona pod adresem
                      ; podzielnym przez 4
```

BYTE

ZASTOSOWANIE

Konwersja typu wyrażenia na typ **BYTE**.

SKŁADNIA

BYTE	*Rezultat wyrażenia jest zamieniany na bajt*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub użyciu etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym (dla procesora 80386)*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

Przy przesyłaniu stałej do pamięci instrukcją **MOV**, asembler musi znać jej rozmiar. W poniższym przykładzie do jego określenia została wykorzystana dyrektywa **BYTE**.

```
          .DATA
Table     DB    1024 dup(?)          ; Rezerwacja pamięci
RecNum    DW    ?                    ; Numer rekordu
RecSize   DW    ?                    ; Rozmiar rekordu
          .CODE
          .
          .
          .
          mov   ax, RecNum           ; Pobierz numer rekordu
          mov   bx, RecSize          ; i jego rozmiar
          mul   bx                   ; Iloczyn stanowi
                                     ; przemieszczenie
                                     ; rekordu w tablicy
```

```
mov   bx, OFFSET Table     ; Pobierz adres tablicy
add   bx, ax               ; Dodaj przemieszczenie
mov   BYTE PTR [bx+10], 0  ; Prześlij bajt pod
                           ; adres BX+10
.
.
.
```

CATSTR

ZASTOSOWANIE

Konkatenacja kilku ciągów znaków.

SKŁADNIA

```
nazwa CATSTR łańcuch1 [, łańcuch2]...
```

Połączenie łańcuchów i umieszczenie rezultatu w zmiennej nazwa.

KOMENTARZ

Sklejane łańcuchy mogą być tekstem objętym w nawiasy trójkątne (<tekst>) lub wcześniej zdefiniowaną etykietą, która reprezentuje ciąg znaków.

PATRZ

SUBSTR, INSTR, SIZESTR *Inne operacje na łańcuchach*

PRZYKŁAD

```
; Definicja zmiennych łańcuchowych
Str1   EQU   <Oto przykładowy ciąg znaków>
Str2   EQU   <Inny ciąg znaków>
; Użycie ich w wyrażeniach
pred INSTR Str1, <przykładowy ciąg znaków> ; Wyszukaj
                                           ; końcową część
                                           ; łańcucha
Str3   SUBSTR  Str1, 0, pred               ; Stwórz
                                           ; łańcuch,
                                           ; który nie
                                           ; zawiera
                                           ; tej części
Str4   CATSTR  Str3, <inny ciąg znaków>    ; Stwórz
                                           ; łańcuch
                                           ; "Oto inny ciąg
                                           ; znaków"
```

DB

ZASTOSOWANIE

Przydział jednego bajtu pamięci i opcjonalne przypisanie mu etykiety.

SKŁADNIA

`[nazwa] DB dana [, dana]...` *Stworzenie elementów danych o rozmiarze bajtu*

KOMENTARZ

Dyrektywa **DB** może zainicjować zmienne liczbami mieszczącymi się w jednym bajcie. Mogą one być również zainicjowane łańcuchem ujętym w cudzysłów.

PATRZ

DUP *Używane do inicjowania danych*
DD, DF, DP, DQ, DW

PRZYKŁAD

```
        .DATA
Msg     DB    "Oto ciąg znaków", 0   ; Ciąg bajtów
Numb    DB    1, 2, 3, 255, -128     ; Liczby
Space   DB    1024   DUP(?)          ; Przydzielenie 1024
                                     ; bajtów pamięci
```

DD

ZASTOSOWANIE

Przydział dwóch słów pamięci i opcjonalne przypisanie im etykiety.

SKŁADNIA

`[nazwa] DD dana [, dana]...`
Stworzenie elementów danych o rozmiarze dwóch słów

KOMENTARZ

Dyrektywa **DD** przydziela cztery bajty pamięci. Pozwala to na przechowanie liczb bez znaku - z zakresu od 0 do 4,294,967,295 lub ze znakiem od -2,147,483,648 do 2,147,483,647. Dane o rozmiarze dwóch słów są często używane do przechowywania dalekich wskaźników.

PATRZ

DUP *Używane do inicjowania danych*
DB, DF, DP, DQ, DT, DW

DF

PRZYKŁAD

```
            .DATA
DataArea    DD    0, 4294967295, 0FF32h ; Liczby
FarPoint    DD    DataArea              ; Dalekie wskazanie
                                        ; na zmienną
                                        ; DataArea
```

DF

ZASTOSOWANIE

Przydział pamięci na daleki wskaźnik (6 bajtów) i opcjonalne przypisanie mu etykiety.

SKŁADNIA

`[nazwa] DF dana [, dana]...`

Stworzenie dalekich wskaźników do 32-bitowych segmentach

KOMENTARZ

Procesory 80386 mogą używać segmentów o rozmiarze do czterech gigabajtów. Do zaadresowania danych w tych segmentach konieczne są więc przemieszczenia 32-bitowe. Dyrektywa **DF** pozwala na przydział pamięci dla tego typu wskaźników (przemieszczenie i segment).

PATRZ

DUP *Używane do inicjowania danych*
DB, DD, DQ, DT, DW

PRZYKŁAD

```
            .386                  ; Udostępnienie instrukcji
                                  ; procesora 80386
            .DATA
DataArea    DD  0, 21, 0FF32h     ; Liczby
FarPoint    DF  DataArea          ; Dalekie wskazanie na
                                  ; zmienną DataArea
```

DQ

ZASTOSOWANIE

Przydział czterech słów pamięci i opcjonalne przypisanie im etykiety.

SKŁADNIA

```
[nazwa] DQ dana [, dana]...
```

Stworzenie elementów danych o rozmiarze czterech słów

KOMENTARZ

Dyrektywa **DD** może służyć do przydzielania pamięci na liczby rzeczywiste podwójnej precyzji. Są one używane przez koprocesor. Rzadziej jest ona stosowana do przydzielania miejsca dla bardzo wielkich liczb całkowitych.

PATRZ

DUP *Używane do inicjowania danych*
DB, DD, DF, DT, DW

PRZYKŁAD

```
        .DATA
BigNum  DQ   0FFFF0000FFFF0000h ; Duża liczba
RealNum DQ   3.14159            ; Liczba rzeczywista
```

DT

ZASTOSOWANIE

Przydział 10 bajtów pamięci i opcjonalne przypisanie im etykiety.

SKŁADNIA

```
[nazwa] DT dana [, dana]...
```

Stworzenie elementów danych o rozmiarze 10 bajtów

KOMENTARZ

Dyrektywa **DD** może służyć do przydzielania pamięci na 80-bitowe liczby rzeczywiste używane w niektórych koprocesorach. Gdy zostaną podane inne niż rzeczywiste liczby, wówczas asembler uzna, że zostały one zapisane w spakowanym kodzie BCD.

PATRZ

DUP *Używane do inicjowania danych*
DB, DD, DF, DQ, DW

PRZYKŁAD

```
BigBCD    DT  1234567890          ; Poniższa definicja
                                  ; jest jej równoważna
BigBCD2   DB  1,2,3,4,5,6,7,8,9,0
Real      DT  1.5123              ; 80-bitowa liczba
                                  ; rzeczywista
```

DUP

ZASTOSOWANIE

Utworzenie tablicy danych zainicjowanej pojedynczą wartością lub nie zainicjowanej tablicy o podanym rozmiarze.

SKŁADNIA

`ilość DUP(dana)` *Tworzy ilość elementów danych*

`ilość DUP(?)` *Tworzy nie zainicjowany obszar danych zawierający określoną liczbę elementów*

KOMENTARZ

Dyrektywa **DUP** jest używana razem z dyrektywami definiującymi dane (**DB**, **DW**, **DQ** itd). Jest to prosty sposób na utworzenie dużego zainicjowanego lub nie obszaru danych.

PATRZ

DB, DD, DF, DP, DQ, DT, DW

PRZYKŁAD

```
array1   DW   20 dup(10)    ; Utworzenie 20 słów
                            ; zainicjowanych wartością 10
spaces   DB   80 dup(' ')   ; Stworzenie 80 spacji
buffer   DB   1024 dup(?)   ; Nie zainicjowany bufor
                            ; o rozmiarze 1024 bajtów
```

DW

ZASTOSOWANIE

Przydział słowa pamięci i opcjonalne przypisanie mu etykiety.

SKŁADNIA

`[nazwa] DW dana [, dana]...`

Stworzenie elementów danych o rozmiarze słowa

KOMENTARZ

Dyrektywa **DW** jest najczęściej używana do utworzenia danych o rozmiarze rejestrów procesora 80x86. Pozwala to na przechowanie liczb bez znaku - z zakresu od 0 do 65,535 lub ze znakiem od -32,768 do 32,767. Dane o rozmiarze słowa są również często używane do przechowywania bliskich wskaźników.

PATRZ

DUP — *Używane do inicjowania danych*
DB, DD, DF, DQ, DT

PRZYKŁAD

```
            .DATA
DataArea    DW    -123, 41, 65535   ; Liczby
NearPoint   DW    DataArea          ; Bliskie wskazanie
                                    ; na zmienną DataArea
```

DWORD

ZASTOSOWANIE

Konwersja rezultatu wyrażenia na podwójne słowo.

SKŁADNIA

DWORD — *Traktowanie rezultatu wyrażenia jako podwójne słowo*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub użyciu etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR — *Używany do zmiany rozmiaru wyrażenia*

PRZYKŁAD

Patrz przykład do dyrektywy **BYTE**.

ENDS

ZASTOSOWANIE

Zaznaczenie końca definicji struktury lub unii.

SKŁADNIA

Tryb MASM:

[nazwa] ENDS — *Koniec struktury lub unii o nazwie nazwa*

EVEN, EVENDATA

Tryb IDEAL:

`ENDS [nazwa]`	*Koniec struktury lub unii o nazwie nazwa*

KOMENTARZ

Dyrektywa ta jest również używana do zaznaczenia końca segmentu. Turbo Assembler rozróżnia je poprzez podane nazwy. Jeśli żadna nazwa nie została podana, to **ENDS** odnosi się do ostatniej dyrektywy **SEGMENT**, **STRUCT** czy **UNION**.

PATRZ

SEGMENT, STRUCT, UNION

PRZYKŁAD

Patrz przykład do **STRUCT**.

EVEN, EVENDATA

ZASTOSOWANIE

Przesunięcie wskaźnika pozycji do następnego adresu parzystego.

SKŁADNIA

EVEN	*Używana w segmencie kodu (wstawia instrukcję NOP)*
EVENDATA	*Używana w segmencie danych (wstawia 0)*

KOMENTARZ

Procesory 8086 i 80286 będą działały szybciej, gdy każdy dostęp do danych będzie wykonany w pojedynczej 16-bitowej operacji.

PATRZ

ALIGN

PRZYKŁAD

```
unaligned   DB    0        ; Dana ta nie musi znajdować
                           ; się pod parzystym adresem.
            EVENDATA
Aligned     DW    ?        ; Zmienna ta zostanie utworzona
                           ; pod najbliższym parzystym adresem
```

EQ

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli są one równe, to zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 EQ wyr2` *Zwraca prawdę, gdy wyr1 równa się wyr2*

KOMENTARZ

Oba wyrażenia obliczane są w trakcie procesu kompilacji. By tego typu operacje mógł wykonywać sam program, należy po prostu napisać odpowiedni kod.

PATRZ

AND, NOT, OR, XOR *Inne operatory stosowane w wyrażeniach*
GE, GT, LE, LT, NE

PRZYKŁAD

Powyższy przykład pokazuje, w jaki sposób kompilacja warunkowa może zostać wykorzystana do pisania programów, które mogą przyjmować różne wersje, zależnie od wartości kilku symboli.

```
            .MODEL  SMALL          ; Dyrektywa ta powoduje
                                   ; ustawienie
                                   ; predefiniowanych
                                   ; zmiennych @DataSize
                                   ; i @CodeSize
BUFSIZE     = 1024                 ; Opis wykorzystywanego
                                   ; bufora
RECSIZE     = 415                  ; Opis wykorzystywanego
                                   ; rekordu
RECCNT      = 500
MEMSIZE     = 128000               ; Ilość używanej pamięci
IF RECSIZE * RECCNT GE MEMSIZE     ; Czy istnieje wystarczająco
                                   ; dużo pamięci
RECCNT      = MEMSIZE/RECSIZE      ; RECCNT zawiera liczbę
                                   ; rekordów, które mieszczą
                                   ; się w pamięci
ENDIF
            .DATA
IF @DataSize EQ 0                  ; Jeśli używane są wskaźniki
                                   ; bliskie
BuffPtr     DW   ?                 ; Przydziel słowo pamięci
ELSE                               ; w przeciwnym razie
BuffPtr     DD   ?                 ; Przydziel podwójne słowo
```

FAR

```
          .CODE
IF RECSIZE LE BUFSIZE              ; Porównanie rozmiarów
                                   ; bufora i rekordu
          mov   cx, RECSIZE        ; Pobierz rozmiar rekordu
ELSE
          mov   cx, BUFSIZE        ; Rekord jest zbyt duży,
                                   ; odczytaj dane o rozmiarze
                                   ; bufora
ENDIF
IF @DataSize NE 0                  ; Jeśli nie, bliskie
                                   ; wskazania danych
          les   bx, BuffPtr        ; Załaduj daleki wskaźnik
ELSE; w przeciwnym razie
          push  ds                 ; Prześlij DS do ES
          pop   es
          mov   bx, BuffPtr        ; oraz pobierz bliski
                                   ; wskaźnik
ENDIF
```

FAR

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na typ **FAR**.

SKŁADNIA

FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*

LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
        .
        .
        .
        call   FAR funcl ; Odwołanie w przód wymaga podania
                         ; rodzaju skoku
        .
        .
        .
funcl PROC  FAR          ; Wywoływana funkcja
```

FWORD

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na typ **FWORD**.

SKŁADNIA

FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*

GE

DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
        .MODEL   SMALL
        .386                    ; Udostępnienie instrukcji
                                ; procesora 80386
        .DATA
GDTSel  DW   8 * 24 - 1         ; Rozmiar pola dla GDTR
        DD   0                  ; Adres bazowy GDT
        .CODE
        .
        .
        .
        lgdt   FWORD GDTSel     ; Prześlij daną z pamięci
                                ; do GDTR
        .
        .
        .
```

GE

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli wynik pierwszego z nich jest większy lub równy wynikowi drugiego, wówczas zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 GE wyr2` *Zwraca prawdę, gdy wyr1 jest większe lub równe wyr2*

Patrz **EQ**

GT

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli wynik pierwszego z nich jest większy niż wynik drugiego, to zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 GT wyr2` *Zwraca prawdę, gdy wyr1 jest większe niż wyr2*

Patrz **EQ**

HIGH

ZASTOSOWANIE

Pobranie ośmiu najbardziej znaczących bitów rezultatu wyrażenia.

SKŁADNIA

`HIGH wyrażenie`

Tryb IDEAL:

`typ HIGH wyrażenie` *Pobranie tylu bitów, ile jest konieczne do wypełnienia pola podanego typu (BYTE, WORD, DWORD itd.).*

KOMENTARZ

Zdefiniowana dana może nie mieścić się w całości w rejestrze (np. zadeklarowana dyrektywą **DD** lub **DQ**). Operator **HIGH** pozwala na pobranie tylko jej starszej części.

PATRZ

LOW

PRZYKŁAD

Poniższy przykład pokazuje zastosowanie dyrektyw **HIGH** i **LOW** do pobrania czterech bajtów długiej liczby całkowitej.

```
IDEAL
```

INSTR

```
BigNum  EQU  12345678h                    ; Długa liczba
                                          ; całkowita
        .
        .
        .
        mov   al, BYTE LOW BigNum         ; Pobranie
                                          ; najmłodszego
                                          ; bajtu
        call  SendData                    ; Wysłanie go
        mov   al, BYTE HIGH ( WORD LOW BigNum ); Pobranie
                                                ; kolejnego
                                                ; bajtu
        call  SendData; Wysłanie go
        mov   al, BYTE LOW ( WORD HIGH BigNum ); Pobranie
                                                ; kolejnego
                                                ; bajtu
        call  SendData                    ; Wysłanie go
        mov   al, BYTE HIGH BigNum        ; Pobranie
                                          ; najstarszego
                                          ; bajtu
        call  SendData                    ; Wysłanie go
        .
        .
        .
```

INSTR

ZASTOSOWANIE

Odszukanie pozycji jednego łańcucha w drugim.

SKŁADNIA

`nazwa INSTR [start,] łańcuch1, łańcuch2`

Odszukaj pozycję łańcucha2 w łańcuchu1. Przeszukiwanie rozpoczynane jest od pozycji start.

KOMENTARZ

Łańcuchy mogą być tekstem objętym w nawiasy trójkątne (<tekst>), lub mogą być wcześniej zdefiniowaną etykietą, która reprezentuje łańcuch. Ponieważ dyrektywa **INSTR** jest wykonywana w czasie kompilacji programu, to operowanie na łańcuchach w czasie działania programu wymaga napisania odpowiedniego kodu.

PATRZ

CATSTR, SIZESTR, SUBSTR

PRZYKŁAD

Patrz przykład do **CATSTR**.

LABEL

ZASTOSOWANIE

Zdefiniowanie etykiety pod bieżącym wskaźnikiem pozycji. Wskaźnik ten nie jest w żaden sposób modyfikowany.

SKŁADNIA

Tryb MASM:

`nazwa LABEL typ` — *Typ może być NEAR, FAR, PROC, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE.*

TrybIDEAL:

`LABEL nazwa typ` — *Typy mogą być takie same jak w trybie MASM.*

KOMENTARZ

Dyrektywa LABEL jest trochę podobna do unii. Pozwala ona bowiem na stworzenie kilku etykiet o różnych typach, które wskazują to samo miejsce w pamięci.

PATRZ

PROC — *Utworzenie etykiety procedury*

PRZYKŁAD

```
BigPointer  LABEL DWORD ; Dostęp do danej o długości
                        ; dwóch słów
OffPtr   DW   ?         ; Ta sama dana podzielona
                        ; na dwa słowa
SegPtr   DW   ?
```

LARGE

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na typ **LARGE**.

LARGE

SKŁADNIA

LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*

KOMENTARZ

Dyrektywa **LARGE** powoduje przekształcenie rezultatu wyrażenia na typ, który odpowiada 32-bitowemu przemieszczeniu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
    .MODEL SMALL
        .386                          ; Udostępnienie
                                      ; instrukcji
                                      ; procesora 80386
                                      ; Definicja
                                      ; 32-bitowego
                                      ; segmentu
BigSeg   SEGMENT  PARA PUBLIC USE32 'DATA'
Numbs    DW   1234                    ; Przydzielenie
                                      ; miejsca na liczbę
FarPoint DF   Numbs                   ; Stworzenie dalekiego
                                      ; wskazania zmiennej
                                      ; Numbs
BigSeg   ENDS
         .CODE
         .
         .
```

```
.
mov   ebx, LARGE FarPoint+2 ; Pobranie
                            ; przemieszczenia
                            ; zmiennej Numbs
.
.
.
```

LE

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli wynik pierwszego z nich jest mniejszy lub równy wynikowi drugiego, to zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 LE wyr2` *Zwraca TRUE, gdy wyr1 jest mniejsze lub równe wyr2*

Patrz **EQ**

LENGTH

ZASTOSOWANIE

Zwraca liczbę elementów danych określonego typu powtórzonych za etykietą.

SKŁADNIA

`LENGTH nazwa` *Nazwa jest nazwą symbolu, który definiuje dane dyrektywą DUP*

KOMENTARZ

Podana etykieta musi odpowiadać zmiennej, której pamięć została przydzielona dyrektywą **DUP**. Zwrócona liczba oznacza ilość utworzonych w ten sposób elementów.

PATRZ

SIZE, TYPE

PRZYKŁAD

```
; Program zeruje obszar pamięci wskazywany przez
; zmienną DataBlk.
DataBlk  DW    411 DUP(?)          ; Definicja danych
```

```
                                ; dyrektywą DUP
sub   ax, ax                    ; Wyzeruj AX
mov   di, OFFSET DataBlk        ; Pobierz adres bloku danych
mov   cx, LENGTH DataBlk        ; Liczba elementów
                                ; bloku danych
rep   stosw                     ; Zapis do zmiennej DataBlk
```

LOW

ZASTOSOWANIE

Pobranie ośmiu najmniej znaczących bitów rezultatu wyrażenia.

SKŁADNIA

`LOW wyrażenie`

Tryb IDEAL:

`typ LOW wyrażenie` *Pobranie tylu bitów, ile jest konieczne do wypełnienia pola podanego typu (BYTE, WORD, DWORD itd.).*

KOMENTARZ

Zdefiniowana dana może nie mieścić się w całości w rejestrze (np. zadeklarowana dyrektywą **DD** lub **DQ**). Operator **LOW** pozwala na pobranie tylko jej młodszej części.

PATRZ

HIGH

PRZYKŁAD

Patrz przykład do **HIGH**.

LT

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli wynik pierwszego z nich jest mniejszy niż wynik drugiego, to zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 LT wyr2` *Zwraca TRUE, gdy wyr1 jest mniejsze niż wyr2*

Patrz **EQ**

MASK

ZASTOSOWANIE

Stworzenie maski używanej do maskowania pól rekordu.

SKŁADNIA

`MASK nazwa_pola`	*Stworzenie maski, która pokrywa pole rekordu o podanej nazwie*
`MASK rekord`	*Stworzenie maski, która pokrywa wszystkie pola rekordu*

KOMENTARZ

Dyrektywa **MASK** jest bardzo użyteczna przy manipulowaniu na częściach rekordu. Mnożenie logiczne maski i bajtu czy słowa, zawierającego rekord, powoduje pozostawienie w nim wartości tylko określonego pola. Następnie zawartość rekordu może być przesunięta o liczbę bitów równą pozycji pola w rekordzie. Operacje te powodują, więc, pobranie wartości podanego pola rekordu.

PATRZ

RECORD
WIDTH

PRZYKŁAD

Patrz przykład do **RECORD**.

MOD

ZASTOSOWANIE

Zwraca resztę z dzielenia wartości wyrażeń.

SKŁADNIA

`wyr1 MOD wyr2`	*Reszta z dzielenia wyr1 przez wyr2*

KOMENTARZ

Wartość tego operatora oraz występujących w nim wyrażeń, jest obliczana w trakcie procesu kompilacji. Przeprowadzenie tej operacji w czasie działania programu wymaga stworzenia odpowiedniego kodu.

PATRZ

AND, OR, NOT	*Inne operatory*
SHL, SHR	*Operatory przesuwania*

NE

PRZYKŁAD

```
            .MODEL  SMALL
; Wyzeruj bufor
BufSize     = 1024                      ; Liczba bajtów w
                                        ; buforze
            .DATA
Buffer      DB   BufSize DUP(?)         ; Definicja bufora
            .CODE
ClearBuf    PROC
            mov   di, OFFSET Buffer     ; Pobierz adres bufora
            sub   ax, ax                ; Wyzeruj AX
            mov   cx, BufSize SHR 1     ; Oblicz liczbę słów
                                        ; w buforze
            rep   stosw                 ; Zerowanie bufora
IF BufSize MOD 2                        ; Czy rozmiar bufora
                                        ; jest liczbą nieparzystą
            stosb                       ; Wyzeruj ostatni bajt
ENDIF
            ret
ClearBuf    ENDP
```

NE

ZASTOSOWANIE

Porównanie rezultatów dwóch wyrażeń. Jeśli nie są one równe, to zwracana jest wartość TRUE. W przeciwnym wypadku rezultatem będzie wartość FALSE.

SKŁADNIA

`wyr1 NE wyr2` *Zwraca TRUE, gdy wyr1 nie równa się wyr2*

Patrz **EQ**

NEAR

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na typ **NEAR**.

SKŁADNIA

NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

Dyrektywa **NEAR** w poniższym przykładzie została użyta w celu wymuszenia odpowiedniego rodzaju wywołania.

```
          .MODEL   LARGE          ; Standardowo wywołania
                                  ; traktowane są jako dalekie
          .CODE
Proc1      PROC
           call    NEAR SubFunc   ; Wymuszenie wywołania
                                  ; bliskiego NEAR
           ret
Proc1      ENDP
SubFunc    PROC  NEAR             ; Funkcja bliska
           .
           .
           .
SubFunc    ENDP
```

NOT

ZASTOSOWANIE

Logiczna operacja **NOT**.

SKŁADNIA

`NOT wyr` *Negacja bitów rezultatu wyrażenia*

KOMENTARZ

Operator ten stosuje się w wyrażeniach, których wartość ma być wyliczona w trakcie procesu kompilacji. Mogą one wystąpić w tekście programu wszędzie tam, gdzie poprawne byłoby użycie dowolnej liczby.

PATRZ

AND	*Logiczna operacja AND*
OR	*Logiczna operacja OR*
XOR	*Logiczna operacja XOR*

PRZYKŁAD

Patrz przykład do dyrektywy **AND**.

OFFSET

ZASTOSOWANIE

Pobranie przemieszczenia etykiety.

SKŁADNIA

`OFFSET etykieta`

KOMENTARZ

Zwykle użycie etykiety jako argumentu instrukcji powoduje pobranie danej przez nią wskazywanej. Natomiast dyrektywa **OFFSET** generuje daną natychmiastową, która jest adresem etykiety.

PATRZ

SEG	*Pobranie numeru segmentu etykiety*

PRZYKŁAD

```
            .MODEL    SMALL
            .DATA
Msg         DB    "This is a message$"
```

```
        .CODE
start:
        mov   ax, SEG DGROUP ; Pobierz numer segmentu
                             ; DGROUP
        mov   ds, ax
        mov   dx, OFFSET Msg ; Pobierz przemieszczenie
                             ; zmiennej Msg
        mov   ah, 9          ; Wyświetlenie komunikatu
        int   21h
        int   21h
        mov   ah, 4Ch        ; Zakończenie programu
        int   21h
        .STACK   512
END  start
```

OR

ZASTOSOWANIE

Logiczna operacja **OR**.

SKŁADNIA

`wyr1 OR wyr2` *Logiczna operacja OR na bitach rezultatów wyrażeń*

KOMENTARZ

Operator ten stosuje się w wyrażeniach, których wartość ma być wyliczona w trakcie procesu kompilacji. Mogą one wystąpić w tekście programu wszędzie tam, gdzie poprawne byłoby użycie dowolnej liczby.

PATRZ

AND	*Logiczna operacja AND*
NOT	*Logiczna operacja NOT*
XOR	*Logiczna operacja XOR*

PRZYKŁAD

Patrz przykład do dyrektywy **AND**.

ORG

ZASTOSOWANIE

Nadanie wskaźnikowi pozycji nowej wartości.

SKŁADNIA

`ORG NowaPozycja` — *NowaPozycja jest przemieszczeniem, które zostanie wpisane do wskaźnika pozycji*

KOMENTARZ

Dyrektywa **ORG** jest używana do umieszczania danych lub kodu programu w określonym miejscu segmentu. Na przykład programy typu .COM muszą zaczynać się od adresu **100h**. Dyrektywa ta ułatwia również dostęp do danych, zawartych w segmencie BIOS-u. Każda zmienna znajduje się pod okreslonym adresem, który może być podany w dyrektywie **ORG**.

PATRZ

SEGMENT

PRZYKŁAD

```
; Utworzenie zmiennych w obszarze danych BIOS-u.
BIOSdata     SEGMENT AT 40h
Com1Port     DW   ?          ; Adres portu COM1 znajduje
                             ; się pod adresem 0
             ORG 17h
KeyStatus    DW   ?          ; Status klawiatury
                             ; przechowywany jest pod
                             ; adresem 17h
             ORG 49h
VidMode      DB   ?          ; Tryb video umieszczony jest
                             ; pod adresem 49h
BIOSdata     ENDS
```

PROC

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na domniemany typ dla procedur.

SKŁADNIA

`PROC` — *Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu. W dyrektywie PROC aktualnie użyty typ zależy od modelu pamięci.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

Dyrektywa **NEAR** w poniższym przykładzie została użyta, aby wymusić odpowiedniego rodzaju wywołania.

```
        .MODEL  SMALL
        .CODE
Func1   PROC
        call    PROC SubFunc   ; Odwołanie w przód wymaga
                               ; podania jego typu
        ret
Func1   ENDP
SubFunc PROC                   ; Wywoływana funkcja
        .
        .
        .
SubFunc ENDP
```

PTR

ZASTOSOWANIE

Zmiana rozmiaru danej, która znajduje się pod podanym adresem.

SKŁADNIA

`typ PTR wyrażenie` *Typ może być NEAR, FAR, PROC, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE. Wyrażenie jest traktowane jako adres zmiennej.*

KOMENTARZ

Dyrektywa **PTR** jest stosowana, gdy rozmiar zmiennej nie jest znany lub jest nieodpowiedni dla przeprowadzanej operacji. Turbo Assembler nie potrafi określić rozmiaru danej, użytej w odwołaniach pośrednich poprzez rejestr np. mov [bx], 0. Wtedy dyrektywa **PTR** (mov WORD PTR [bx], 0) pozwala na określenie rozmiaru danej. Asembler nie zna również rozmiaru etykiet, zdefiniowanych w dalszej części programu. Jest to tzw. odwołanie w przód, a dyrektywa **PTR** jest używana do ustalenia jego rodzaju.

PATRZ

NEAR
FAR
PROC
BYTE
WORD
DWORD
FWORD
PWORD
QWORD
TBYTE

PRZYKŁAD

```
; Inny sposób pobrania dużych danych do rejestrów.
            .MODEL  LARGE
            .DATA
SampleData  DD   1234
            .CODE
Start       PROC
            mov   ax, SEG DGROUP            ; Załaduj
                                            ; rejestr DS
            mov   ds, ax
            mov   ax, WORD PTR SampleData   ; Pobierz
                                            ; pierwsze słowo
                                            ; zmiennej
                                            ; SampleData
```

```
            mov   dx, WORD PTR SampleData+2; Odczytaj drugie
                                           ; słowo
            call  FAR PTR DoData           ; Użycie
                                           ; dyrektywy PTR
                                           ; w odwołaniu
                                           ; w przód
            mov   BYTE PTR [bx], 0         ; Przesłanie
                                           ; bajtu pod adres
                                           ; zawarty w BX
            .
            .
Start       ENDP
DoData      PROC
            .
            .
            .
DoData      ENDP
            .STACK   1024
END Start
```

PWORD

Patrz opis dyrektywy **FWORD**.

QWORD

ZASTOSOWANIE

Konwersja rezultatu wyrażenia na podany typ.

SKŁADNIA

QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*

.RADIX, RADIX

BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
          .MODEL   SMALL
          .DATA
RealNumbs DQ   256  DUP(0)    ; Definicja tablicy liczb
          .CODE
; Na wejściu rejestr AX zawiera indeks w tablicy
AddReal   PROC
          shl   ax, 1         ; Razy 2
          shl   ax, 1         ; Razy 4
          shl   ax, 1         ; Razy 8
          mov   bx, OFFSET RealNumbs
          add   bx, ax
          fadd  QWORD [bx]    ; Dodawanie liczb z
                              ; wykorzystaniem koprocesora
          ret
AddReal   ENDP
```

.RADIX, RADIX

ZASTOSOWANIE

Zmiana domniemanej podstawy liczb.

SKŁADNIA

```
.RADIX wyrażenie        Rezultat wyrażenia musi być równy 2, 8, 10 lub 16
RADIX wyrażenie
```

KOMENTARZ

Podstawą domniemaną jest 10. Każda liczba, która nie ma przyrostka traktowana jest, jako zapisana w systemie dziesiętnym. Natomiast po użyciu dyrektywy **RADIX** wszystkie liczby bez podanej podstawy są traktowane jako zapisane przy podstawie, będącej argumentem tej dyrektywy. Bez względu jednak na wcześniejsze dyrektywy **RADIX**, ich argument jest zawsze traktowany jako zapisany w systemie dziesiętnym.

PRZYKŁAD

```
; Wszystkie liczby bez jawnie określonej podstawy
; zapisane są w systemie dziesiętnym.
        DW  1234    ; Liczba dziesiętna
        DB  15h     ; Liczba szesnastkowa (przyrostek h)
.RADIX  8           ; Teraz wszystkie liczby bez jawnie
                    ; określonej podstawy zapisane są
                    ; w systemie ósemkowym.
        DW  1234    ; Liczba ósemkowa (= 668 dziesiętnie)
        DW  14d     ; Liczba dziesiętna (przyrostek d)
```

RECORD

ZASTOSOWANIE

Definicja rekordu, który składa się z pól bitowych.

SKŁADNIA

Tryb MASM:

```
nazwa RECORD pole1:szerokość[=wyr] [,pole2:szerokość[=wyr]]...
```

Tryb IDEAL:

```
RECORD nazwa pole1:szerokość[=wyr] [,pole2:szerokość[=wyr]]...
```

KOMENTARZ

Rekordy są bardzo podobne do struktur, z tym, że ich pola są częściami bajtów lub słów. Pola przydzielane są w podanej kolejności od najbardziej do najmniej znaczącego bitu. Jeśli rozmiar rekordu jest mniejszy niż 8 lub 16 bitów, to wszystkie pola są tak przesuwane, by ostatnie z nich zawierało bit najmniej znaczący. Wyrażenia służą natomiast inicjacji pól rekordu.

SEG

Dyrektywa **RECORD** nie powoduje przydzielenia pamięci. Po zdefiniowaniu rekordu jego nazwa może służyć do definiowania zmiennych w ten sam sposób, w jaki jest używana nazwa struktury.
Nazwa pola rekordu, użyta w wyrażeniach, oznacza liczbę bitów, o jaką należy przesunąć rekord, by pole to zawierało bit najmniej znaczący.

PATRZ

STRUC

PRZYKŁAD

```
          .MODEL   SMALL
          .DATA
ELREC     RECORD  Printers:2, Resrv:2, Coms:3, Resrv2:1,
Drives:2, VidMode:2, Resrv3:1, Point:1, Math:1, Boot:1
EquipLst  ELREC   ?
          .CODE
GetComs   PROC
          mov   ax, 40h        ; Segment danych BIOS-u
          mov   es, ax
          mov   ax, es:[10h]   ; Pobierz dane o systemie
          mov   EquipLst, ax   ; Prześlij je do pamięci
          and   ax, MASK Coms  ; Zamaskuj bity portów
                               ; szeregowych
          mov   cl, Coms       ; Pobierz licznik przesunięć
          shr   ax, cl         ; AX zawiera liczbę portów
                               ; szeregowych
          ret
GetComs   ENDP
END
```

SEG

ZASTOSOWANIE

Pobranie numeru segmentu.

SKŁADNIA

`SEG wyrażenie` *Wyrażenie jest traktowane jako adres zmiennej*

KOMENTARZ

Każda etykieta znajduje się w określonym segmencie. Etykiety, wskazujące dane lub kod programu, znajdują się odpowiednio w segmentach danych i kodu. Dyrektywa **SEG** pozwala na pobranie numeru segmentu lub grupy, która zawiera podaną etykietę.

PATRZ

OFFSET

PRZYKŁAD

Przed odwołaniem się do danych każdy program musi odpowiednio załadować rejestr DS. Jeśli zastosowano dyrektywy uproszczone, to rejestr ten powinien wskazywać na grupę **DGROUP**.

```
; Ogólna postać typowego programu, który używa
; uproszczonych dyrektyw segmentowych.
          .MODEL   SMALL
          .DATA                  ; w tym segmencie należy
                                 ; umieścić definicje danych
          .
          .
          .
          .CODE                  ; w tym segmencie należy
                                 ; umieścić kod programu
Start     PROC
          mov   ax, SEG DGROUP; Pobierz adres grupy DGROUP
          mov   ds, ax           ; i prześlij go do
                                 ; rejestru DS
          .
          .
          .
          ret
Start     ENDP                   ; Koniec funkcji Start
          .                      ; Definicje innych funkcji
          .
          .
          .STACK   1024          ; Stworzenie segmentu stosu
END Start                        ; Definicja punktu wejścia
                                 ; do programu
```

SHL

ZASTOSOWANIE

Arytmetyczne przesunięcie w lewo.

SKŁADNIA

`wyr1 SHL licznik` *Przesunięcie wartości wyrażenia wyr1 w lewo o licznik razy*

Patrz opis dyrektywy **MOD**.

SHORT

ZASTOSOWANIE

Konwersja typu rezultatu wyrażenia na typ **SHORT**.

SKŁADNIA

SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
        .CODE
Func    PROC
        .
        .
        .
        jmp   SHORT FuncLbl   ; Wymuszenie użycia skoku
                              ; krótkiego
                              ; Pomiędzy tą instrukcją,
                              ; a etykietą musi
```

```
                .                    ; znajdować się mniej niż
                                     ; 127 bajtów
                .
FuncLbl:
```

SHR

ZASTOSOWANIE

Arytmetyczne przesunięcie w prawo.

SKŁADNIA

`wyr1 SHR licznik` *Przesunięcie wartości wyrażenia wyr1 w prawo o licznik razy*

Patrz opis dyrektywy **MOD**.

SIZE

ZASTOSOWANIE

Zwrócenie rozmiaru danej.

SKŁADNIA

`SIZE nazwa` *Nazwa jest nazwą uprzednio zdefiniowanej zmiennej*

KOMENTARZ

Dana jest zmienną zadeklarowaną jedną z dyrektyw, przydzielających pamięć (**DB**, **DD**, **DW** itp.). **SIZE** oznacza liczbę przydzielonych jej bajtów. Jeśli w deklaracji został użyty operator **DUP**, to **SIZE** oznacza wtedy liczbę bajtów, przydzielonych na pojedynczy element, pomnożoną przez liczbę, podaną w dyrektywie **DUP**.
W trybie MASM operator **SIZE** nie uwzględnia liczby elementów, z których składa się zmienna, a także zagnieżdżonych operatorów **DUP**. Natomiast w trybie IDEAL zwraca on łączną liczbę bajtów przydzielonych podanej zmiennej.

PATRZ

LENGTH, TYPE

PRZYKŁAD

```
IDEAL
          MODEL  SMALL
Str1      DB  "Hello\n", 0
```

SIZESTR

```
        CODESEG
Start:
        mov   cx, SIZE Str1   ; w trybie IDEAL powoduje
                              ; to załadowanie liczby 7,
                              ; a w trybie MASM liczby 1
        .
        .
        ret
        STACK   1024
END Start
```

SIZESTR

ZASTOSOWANIE

Zwrócenie liczby znaków łańcucha.

SKŁADNIA

`nazwa SIZESTR łańcuch` *Umieszczenie długości łańcucha w zmiennej nazwa*

KOMENTARZ

Łańcuchy mogą być tekstem objętym w nawiasy trójkątne (<tekst>), rezultatem działania innej dyrektywy łańcuchowej (**EQU**, **CATSTR**, **SUBSTR**). W trybie IDEAL dla łańcuchów zdefiniowanych dyrektywą **DB**, należy użyć operatora **SIZE**.

PATRZ

SUBSTR, CATSTR, INSTR
SIZE

PRZYKŁAD

W języku Pascal pierwszym znakiem łańcucha jest jego długość. Dyrektywa **SIZESTR** może okazać się użyteczna przy definiowaniu tego rodzaju literałów tekstowych.

```
; Definicja symboli, które oznaczają sam łańcuch
; oraz jego długość.
StrDef   EQU       <Oto literał znakowy>
StrLen   SIZESTR   StrDef
; Definicja zmiennej programu, która zawiera
;łańcuch i jego rozmiar
Str1     DB        StrLen, StrDef
```

SMALL

ZASTOSOWANIE

Konwersja rezultatu wyrażenia na typ **SMALL**.

SKŁADNIA

SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

PRZYKŁAD

```
          .MODEL  SMALL
          .386                          ; Udostępnienie
                                        ; instrukcji
                                        ; procesora 80386
                                        ; Definicja
                                        ; 16-bitowego
                                        ; segmentu
BigSeg    SEGMENT  PARA PUBLIC USE16 'DATA'
Numbs     DW    1234                    ; Przydzielenie
```

```
                                        ; miejsca na liczbę
FarPoint  DD    Numbs                   ; Stworzenie wskazania
                                        ; zmiennej Numbs
BigSeg    ENDS
          .CODE
          .
          .
          .
          mov   bx, SMALL FarPoint+2    ; Pobranie
                                        ; przemieszczenia
                                        ; zmiennej Numbs
          .
          .
          .
```

STRUC

ZASTOSOWANIE

Zdefiniowanie struktury.

SKŁADNIA

Tryb MASM:

```
nazwa STRUCT
[nazwa] ENDS
```

Tryb IDEAL:

```
STRUC nazwa pola
ENDS [nazwa]
```

KOMENTARZ

Struktury pozwalają zgrupować powiązane ze sobą zmienne. Dyrektywa **STRUC** definiuje sposób ich rozmieszczenia, ich nazwy oraz inicjujące je wartości. Każde pole struktury posiada swoją nazwę, za pomocą której można się do niego odwołać. Ma również swój typ. Zależy on od dyrektywy, użytej przy deklaracji tego pola (**DB**, **DD**, **DW** itp.). Zwykle w tych dyrektywach stosuje się znak zapytania (**?**), co oznacza, że wartość tego pola nie jest zdefiniowana. Jeśli natomiast zostanie podana wartość, to będzie ona wartością domniemaną pola we wszystkich deklaracjach zmiennych, które używają nazwy tej struktury.

PATRZ

ENDS, UNION

PRZYKŁAD

Struktury są często wykorzystywane do reprezentowania rekordów bazy danych. Pozwala to na traktowanie rekordu jako pojedynczego elementu danych.

```
          .MODEL  LARGE
          .DATA
; Definicja struktury dla rekordu, który zawiera nazwisko
; osoby oraz jej adres
NameAddr  STRUC
Name      DB    20 dup(?)            ; Nazwisko osoby
Street    DB    20 dup(?)            ; Adres zamieszkania
City      DB    "Kirkland"           ; Domniemaną nazwą
                                     ; miasta jest Kirkland
State     DB    "WA"                 ; Domniemanym stanem
                                     ; jest Washington
zip       DB    "980340000"          ; 9-cyfrowy kod
                                     ; pocztowy
NameAddr  ENDS
Rec1      NameAddr                   ; Deklaracja rekordu
FileHand  DW    ?                    ; Deskryptor pliku
                                     ; bazy danych
          .CODE
Start     PROC                       ; Inicjacja
          mov   ax, SEG DGROUP
          mov   ds, ax
          .
          .
          .
Start     ENDP
PrintName PROC                       ; Wyświetl pole
                                     ; nazwiska
          mov   dx, OFFSET Rec1.Name; Pobierz adres
                                     ; nazwiska
          mov   ah, 9                ; Funkcja, która
                                     ; wyświetla łańcuch
                                     ; znaków
          int   21h
          ret
PrintName ENDP
WriteRec  PROC                       ; Zapis rekordu
                                     ; do pliku
          mov   bx, FileHand         ; Pobierz deskryptor
                                     ; pliku
          mov   cx, TYPE NameAddr    ; Pobierz rozmiar
                                     ; rekordu
          mov   dx, OFFSET Rec1      ; Pobierz adres rekordu
```

```
        mov   ah, 40h          ; Funkcja, która
                               ; zapisuje rekord
                               ; do pliku
        int   21h
        ret
WriteRec ENDP
END  Start
```

SUBSTR

ZASTOSOWANIE

Pobranie części łańcucha i umieszczenie jej w innym ciągu znaków.

SKŁADNIA

`nazwa SUBSTR łańcuch, pozycja [,rozmiar]`

Umieszczenie części łańcucha, która zaczyna się od podanej pozycji i o podanym rozmiarze (lub do końca łańcucha), w zmiennej nazwa

KOMENTARZ

Łańcuchy mogą być tekstem objętym w nawiasy trójkątne (<tekst>), lub wcześniej zdefiniowaną etykietą, która reprezentuje łańcuch. Ponieważ dyrektywa **SUBSTR** jest wykonywana w czasie kompilacji programu, to operowanie na łańcuchach w czasie działania programu wymaga napisania odpowiedniego kodu.

PATRZ

CATSTR, SIZESTR, INSTR

PRZYKŁAD

Patrz przykład do **CATSTR**.

TBYTE

ZASTOSOWANIE

Konwersja typu wyrażenia na typ **TBYTE**.

SKŁADNIA

`TBYTE wyrażenie`

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR — *Używany do zmiany rozmiaru wyrażenia*

PRZYKŁAD

```
            .MODEL   SMALL
            .DATA
TBArray     DT   100 DUP(?)
            .CODE
; Indeks w tablicy znajduje się w rejestrze AX
LongLoad    PROC
            mov   bx, OFFSET TBArray
            shl   ax, 1          ; Razy 2
            add   bx, ax         ; Dodaj do adresu
            shl   ax, 1          ; Razy 4
            shl   ax, 1          ; Razy 8
            add   bx, ax
            fld   TBYTE [bx]     ; Rozkaz dla koprocesora,
                                 ; który nakazuje załadowanie
                                 ; liczby
            ret
LongLoad    ENDP
```

THIS

ZASTOSOWANIE

Stworzenie adresu równego aktualnej wartości wskaźnika pozycji.

SKŁADNIA

`THIS typ` — *Typ może być NEAR, FAR, PROC, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE. Opisuje on daną w pamięci, która znajduje się pod aktualnym adresem.*

KOMENTARZ

Operator **THIS** jest podobny do operatora **$**. Różnica między nimi polega natomiast na tym, że operator **THIS** pozwala na określenie typu danej. Jest to użyteczne przy stosowaniu dyrektyw **EQU** lub =, a jest to jedyny sposób na podanie typu zmiennej.

.TYPE, SYMTYPE

PATRZ

EQU, LABEL

PRZYKŁAD

```
Alabel    EQU    THIS DWORD    ; Utworzenie etykiety bieżącej
                               ; pozycji. Traktowana jest ona
                               ; jako zmienna typu DWORD.
```

.TYPE, SYMTYPE

ZASTOSOWANIE

Zwraca bajt, który opisuje podany symbol.

SKŁADNIA

`.TYPE nazwa`	*Zwraca typ podanej zmiennej*
`SYMTYPE nazwa`	*To samo działanie. Tryb IDEAL.*

KOMENTARZ

Bajt zwracany przez dyrektywę **.TYPE** składa się z bitów, których wartość zależy od typu podanego symbolu. Ich znaczenie zostało przedstawione w tabeli 9-3.

Tabela 9-3. Bity zwracane przez dyrektywę .TYPE

Bit	Opis
0	Symbol względny programu
1	Symbol względny danych
2	Stała
3	Pośredni tryb adresowania
4	Rejestr
5	Symbol jest zdefiniowany
7	Symbol jest zewnętrzny

PATRZ

TYPE

PRZYKŁAD

Patrz przykład do **TYPE**.

TYPE

ZASTOSOWANIE

Zwraca liczbę, która identyfikuje rozmiar lub typ symbolu.

SKŁADNIA

`TYPE wyrażenie` — *Zwraca liczbę bajtów, z których składa się typ rezultatu wyrażenia. Jeśli symbol jest funkcją typu NEAR, to TYPE zwraca wartość -1, a gdy typu FAR, zwraca -2.*

KOMENTARZ

Gdy w programie potrzebny jest rozmiar pewnego symbolu, to powinien on być ustalany za pomocą operatora **TYPE**. Jeśli bowiem w przyszłości zajdzie potrzeba zmiany tego typu, wówczas nie trzeba będzie przeglądać całego programu, aby sprawdzić, w których jego miejscach została użyta odpowiadająca mu liczba. Natomiast operator **TYPE** przeprowadzi tę operację automatycznie w czasie kompilacji programu.

PATRZ

LENGTH, SIZE

PRZYKŁAD

```
        .MODEL   SMALL
        .DATA
Var1    DW   ?
        .CODE
Start:
        mov   ax, SEG DGROUP
        mov   ds, ax
        mov   es, ax
; Wyzeruj zmienną Var1. Poniższy kod będzie działał
; poprawnie niezależnie od rozmiaru zmiennej Var1.

        sub   ax, ax          ; Wyzeruj AX
        mov   di, OFFSET Var1 ; Pobierz adres Var1
        mov   cx, TYPE Var1   ; Pobierz rozmiar Var1
        rep   stosb           ; Zerowanie zmiennej Var1
        .STACK   512
END Start
```

UNION

ZASTOSOWANIE

Definicja struktury, której wszystkie pola znajdują się w tym samym miejscu pamięci.

SKŁADNIA

Tryb MASM:

```
nazwa UNION                    Stworzenie unii o podanej nazwie
pola
[nazwa] ENDS
```

Tryb IDEAL:

```
UNION nazwa pola
pola
ENDS [nazwa]
```

KOMENTARZ

Unie definiuje się, deklaruje i korzysta się z nich w podobny sposób jak ze struktur. Różnica polega jedynie na tym, że pola unii nakładają się na siebie. Cecha ta może być wykorzystana do odwoływania się do tego samego miejsca pamięci w różny sposób. Traktowane jest ono bowiem wtedy jako zmienna, która ma kilka typów.

PATRZ

STRUC, ENDS

PRZYKŁAD

```
          .MODEL   SMALL
          .DATA
; Unia zdefiniowana poniżej pozwala traktować wskaźniki
; jako podwójne słowa dwa słowa pojedyncze.
FarPtr    UNION
Big       DD   ?
          STRUC
          PtrOff   DW   ?
          PtrSeg   DW   ?
          ENDS
FarPtr    ENDS
Msg1      DB   "Ten wskaźnik wskazuje tutaj", 0
; Utworzenie wskaźnika na zmienną Msg1
Ptr1      FarPtr   < Mag1, ? >
          .CODE
          .
          .
          .
```

```
mov   es, Ptr1.PtrSeg  ; Prześlij numer segmentu
                       ; do ES
mov   si, Ptr1.PtrOff  ; Prześlij przemieszczenie
                       ; do SI
.
.
.
```

UNKNOWN

ZASTOSOWANIE

Konwersja rezultatu wyrażenia na podany typ.

SKŁADNIA

UNKNOWN	*Rezultat wyrażenia nie jest znany*

KOMENTARZ

Działanie tej dyrektywy sprawia, że rezultat wyrażenia nie posiada żadnego typu. W przykładzie przedstawiono zastosowanie tej dyrektywy do utworzenia etykiety, którą można zaadoptować do sytuacji, w której jest używana.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Traktowanie rezultatu wyrażenia jako bajtu*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
WORD	*Traktowanie rezultatu wyrażenia jako słowa*

WIDTH

PRZYKŁAD

```
          .MODEL   SMALL
          .DATA
Buffer    DB     1024 DUP(0)     ; Utworzenie tablicy
BuffPtr   EQU    UNKNOWN Buffer  ; Utworzenie symbolu, który
                                 ; nie ma określonego typu
          .CODE
Funcl     PROC
          .
          .
          .
          mov    al, [BuffPtr]   ; Pobierz bajt z tablicy
          mov    dx, [BuffPtr]   ; Pobierz słowo z tablicy
          .
          .
          .
Funcl     ENDP
```

Bez użycia dyrektywy **UNKNOWN**, przy kompilowaniu drugiej instrukcji **MOV** powstałby błąd niezgodności typów.

WIDTH

ZASTOSOWANIE

Zwraca liczbę bitów używanych przez rekord lub jego pole.

SKŁADNIA

`WIDTH Pole_rekordu`	*Zwraca liczbę bitów pola rekordu*
`WIDTH Rekord'`	*Zwraca liczbę bitów zajmowanych przez rekord*

KOMENTARZ

Operator **WIDTH**, wraz z operatorami **RECORD** i **MASK**, dostarcza informacji niezbędnych przy operowaniu na polach rekordu. Może on służyć do stworzenia maski lub do obliczenia zakresu liczb, mieszczących się w danym polu rekordu.

PATRZ

MASK, RECORD

PRZYKŁAD

Patrz przykład do **RECORD**.

WORD

ZASTOSOWANIE

Konwersja typu wyrażenia na typ **WORD**.

SKŁADNIA

WORD	*Traktowanie rezultatu wyrażenia jako słowa*

KOMENTARZ

O ile typ rezultatu wyrażenia nie jest znany, to należy go jawnie określić. Sytuacja taka może się zdarzyć przy stosowaniu pośredniego trybu adresowania lub gdy użyjemy etykiety zdefiniowanej w dalszej części programu.

PATRZ

PTR	*Używany do zmiany rozmiaru wyrażenia*
BYTE	*Rezultat wyrażenia jest zamieniany na bajt*
DWORD	*Traktowanie rezultatu wyrażenia jako podwójne słowo*
FAR	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie kodu*
FWORD	*Traktowanie rezultatu wyrażenia jako dalekiego wskazania w segmencie 32-bitowym procesora 80386*
LARGE	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie 32-bitowym procesora 80386*
NEAR	*Traktowanie rezultatu wyrażenia jako bliskiego wskazania w segmencie kodu*
PROC	*Traktowanie rezultatu wyrażenia jako bliskiego lub dalekiego wskazania w segmencie kodu zależnie od użytego modelu pamięci*
PWORD	*Podobnie jak PWORD*
QWORD	*Traktowanie rezultatu wyrażenia jako poczwórnego słowa*
SHORT	*Traktowanie rezultatu wyrażenia jako wskazania krótkiego*
SMALL	*Traktowanie rezultatu wyrażenia jako 16-bitowego przemieszczenia*
TBYTE	*Traktowanie rezultatu wyrażenia jako wartości 10-bajtowej*
UNKNOWN	*Rezultat wyrażenia nie jest znany*

PRZYKŁAD

W przykładzie dyrektywa **WORD** została użyta do przechowania wskaźnika typu **FAR**.

```
        .MODEL   SMALL
        .DATA
OldVect DD   ?                          ; Zmienna, która
                                        ; przechowuje
                                        ; wskaźnik
        .CODE
; AL zawiera numer pobieranego wektora przerwań
GetVect PROC
```

```
        mov   ah, 35h                  ; Funkcja, która
                                       ; pobiera wektor
                                       ; przerwań
        int   21h
        mov   WORD PTR OldVect, bx     ; Zapamiętaj młodsze
                                       ; słowo
        mov   WORD PTR OldVect+2, es   ; Zapamiętaj starsze
                                       ; słowo
        ret
GetVect ENDP
```

XOR

ZASTOSOWANIE

Logiczna operacja różnicy symetrycznej.

SKŁADNIA

`wyr1 XOR wyr2`	*Logiczna operacja XOR na bitach rezultatów wyrażeń*

KOMENTARZ

Operator ten stosuje się w wyrażeniach, których wartość ma być wyliczona w trakcie procesu kompilacji. Mogą one wystąpić w tekście programu wszędzie tam, gdzie poprawne byłoby użycie dowolnej liczby.

PATRZ

AND	*Logiczna operacja AND*
NOT	*Logiczna operacja NOT*
OR	*Logiczna operacja OR*

PRZYKŁAD

Patrz przykład do dyrektywy **AND**.

Rozdział 10 Makrodefinicje, Procedury i Programowanie Modułowe

Pisanie dużych programów nie jest zadaniem łatwym. Można je jednak sobie uprościć, dzieląc projekt na wiele, dających się ogarnąć, części. Komputer, wykonując program traktuje go bowiem jako strumień bajtów, tylko od czasu do czasu zmieniany jest rozkazem skoku lub wywołania. Nie ma żadnych wymagań w stosunku do kolejności umieszczania w pamięci jego części. Powinny one tylko poprawnie wykonywać swoje zadania.
Języki wysokiego poziomu jak C lub Pascal pozwalają na logiczny podział programu, udostępniając techniki programowania strukturalnego. Ich podstawą jest użycie funkcji i procedur w celu podzielenia programu na mniejsze części. W dobrze zaprojektowanym programie każda procedura powinna być napisana i przetestowana, niezależnie od jego pozostałych części składowych.
Do zrozumienia działania takiego programu potrzebna jest tylko znajomość jego struktury. Szczegóły implementacji poszczególnych funkcji i procedur nie są bowiem ważne dla ogólnej struktury programu. Wtedy praca nad jedną częścią programu, pozwala skupić się tylko na zadaniu, które ma ona wykonywać, zapominając zupełnie o reszcie programu.
Techniki programowania strukturalnego wiążą się ściśle z modułami. Moduły powstałe przy pisaniu jednego programu mogą być potem wykorzystywane w wielu innych. Na przykład, moduł, który zawiera podprogramy, wyświetlające dane na ekranie, może przydać się w większości pisanych programów. Powoduje to więc znaczne zwiększenie wydajności programowania. Częściej używane moduły mogą zostać umieszczone w bibliotece. Natomiast program łączący wybierze z biblioteki tylko te moduły, które są konieczne do konsolidacji nowego programu.
Turbo Assembler pozwala na wykorzystanie technik programowania strukturalnego. Każdy program może się składać z wielu oddzielnych plików źródłowych. Każdy taki plik powinien być logicznym modułem programu. Moduły natomiast mogą się składać z makrodefinicji i procedur. Makrodefinicje są podobne do procedur, z tym że zamiast generowania wywołania do podprogramu, kod ten wstawiany jest w miejscu odwołania się do makrodefinicji. Komendy tworzące procedury i makrodefinicje pozwalają na dokładniejsze kontrolowanie punktów wejścia i wyjścia oraz danych programu. Jest to jeden z kluczy napisania poprawnie zbudowanego i działającego programu.
Dane w programie, napisanym strukturalnie są bardzo ważne. Każdy moduł powinien wykorzystywać tak mało efektów ubocznych, jak jest to tylko możliwe. Polecenia Turbo Assemblera, dotyczące importowania i eksportowania danych, ograniczają powstanie efektów ubocznych tylko do wyspecyfikowanych symboli.
Ważną cechą programów strukturalnych jest łatwość ich modyfikacji. Zmiana, dotycząca jednego modułu, jest niewidoczna w reszcie programu. Cecha ta może zostać wykorzystana przy pisaniu kilku wersji programu, które działają w różnych środowiskach. W Turbo Assemblerze wersje te mogą być nawet generowane przez jeden plik źródłowy dzięki istnieniu kompilacji warunkowej.

Podział programu na moduły

Pierwszym krokiem przy pisaniu programu strukturalnego jest podzielenie go na moduły. Wybór modułów ma ogromny wpływ na łatwość pisania, testowania i konserwowania ostatecznego produktu. Każdy moduł powinien wykonywać określone zadanie. Powinno ono być tak dobrane, by podprogramy, które wchodzą w jego skład wykonywały podobne operacje, operowały na tych samych danych lub odpowiadały pewnej części w strukturze programu. Zadanie modułu powinno być również tak dobrane, by ograniczyć do minimum liczbę powiązań z innymi modułami oraz ułatwiać analizę całego programu. Na przykład, może istnieć pojedynczy moduł obsługujący ekran. Ograniczenie interakcji z innymi modułami można osiągnąć przez ukrycie zmiennych dotyczących ekranu i stworzenie dobrze zdefiniowanego łącza, z apomocą którego przekazywane są dane do wyświetlenia. W innym mogą znajdować się, na przykład, funkcje jednego z menu programu. Odpowiadał by więc on pewnej części struktury programu.

Każdy moduł znajduje się w oddzielnym pliku źródłowym. Pozwala to więc na łatwe ich rozróżnianie. Asembler pozwala na pewną ochronę zmiennych, używanych w danym module przez ograniczenie zasięgu ich widoczności.

Zasięg symbolu określa miejsca programu, w których może on zostać wykorzystany. Zasięg większości symboli, zdefiniowanych w jednym pliku źródłowym, ogranicza się tylko do niego samego. Mogą one więc być używane tylko w tym właśnie pliku. Przykładem symbolu o szerszym zakresie jest segment, który ma typ połączenia zdefiniowany jako **PUBLIC**. Tego typu symbole widoczne są we wszystkich plikach wchodzących w skład programu.

Dobrze zaprojektowany moduł może być następnie wykorzystywany w wielu innych programach. Każdy z nich musi tylko dostosować się do sprzęgu zastosowanego w danym module do komunikacji z otoczeniem. Często moduły umieszczane są w bibliotekach. Biblioteki tworzymy i obsługujemy przy pomocu programu LIB. Przy konsolidacji programu należy tylko programowi, łączącemu podać informację o wykorzystanych bibliotekach. Można to zrobić, stosując jedną z komend konsolidatora lub należy zastosować dyrektywę **INCLUDELIB** o formacie:

```
INCLUDELIB    NazwaPliku
```

W dyrektywie tej *NazwaPliku* jest nazwą włączanej biblioteki. Jeśli nie zostanie podane rozszerzenie, to przyjęte ono zostanie jako .LIB. Jeśli nie zostanie podana ścieżka dostępu, wówczas asembler przeszuka najpierw katalogi podane po opcji /I, a następnie katalog bieżący.

Współdzielenie zmiennych i symboli

Turbo Assembler może wykorzystać tylko te symbole, które zostały zdefiniowane w aktualnie kompilowanym pliku źródłowym. Dzięki temu każdy plik źródłowy może posiadać swoje prywatne zmienne. Dla programowania strukturalnego wydaje się to być rozwiązaniem idealnym, między różnymi modułami nie ma żadnych powiązań. Wada tego rozumowania polega na tym, że kod i dane zdefiniowane w jednym module nigdy nie mogłyby zostać wykorzystane w innych modułach. W takim razie mogłyby one wcale nie istnieć.

Aby zlikwidować tę izolację istnieją dyrektywy, które pozwalają definiować symbole, dostępne z wielu modułów. Oczywiście powstaje wtedy zagrożenie pojawienia się efektów

ubocznych, ale zostaną one tylko ograniczone do symboli, określonych jawnie przez programistę. Upraszcza to również proces uruchamiania, gdyż w poszukiwaniu efektów ubocznych tylko te właśnie symbole muszą być śledzone.

SYMBOLE PUBLICZNE

By symbol zdefiniowany w jednym pliku mógł być dostępny w innych modułach, musi zostać zdefiniowany jako symbol publiczny. Jeśli natomiast symbol nie jest publiczny, to żaden inny moduł nie ma do niego dostępu. Dyrektywa **PUBLIC** pozwala określić symbole, które będą widoczne w innych plikach. Jej argumentem jest lista symboli, oddzielonych przecinkami.

```
PUBLIC   symbol1 [, symbol2]...
```

SYMBOLE ZEWNĘTRZNE

Symbol, zdefiniowany w jednym z modułów jako publiczny, może być wykorzystywany w pozostałych modułach. Jednakże zanim zostanie on użyty, asembler musi zostać poinformowany, że pochodzi on z innego modułu. Dodatkowo musi też zostać określony jego typ. Do definiowania symboli zewnętrznych służy dyrektywa **EXTRN** o formacie:

```
EXTRN   symbol1:typ1 [, symbol2:typ2]...
```

Typ symbolu może być określony jako: BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, NEAR, FAR lub PROC. Typ ten nie jest porównywany z typem podanym w module, który uczynił symbol publicznym. Do programisty należy więc zadbanie o ich zgodność. Poniższe przykłady pokazują, w jaki sposób dwa moduły mogą dzielić symbole.

```
; Plik MAIN.ASM
; Jest to plik główny. Używa on funkcji, zdefiniowanej w in-
nym pliku źródłowym
        .MODEL   SMALL
        EXTRN   PrintMsg:PROC ; Funkcja ta znajduje się
                              ; w innym pliku źródłowym
        .DATA
        PUBLIC   Mesg1
Mesg1   DB   "Hello, world", 0Dh, 0Ah, '$'
        .CODE
Start:
        mov   ax, SEG DGROUP  ; Pobierz adres segmentu
                              ; danych
        mov   ds, ax          ; i prześlij go do DS
        call  PrintMsg        ; Wywołaj funkcję, która
                              ; znajduje się w innym module
        mov   ah, 4Ch         ; Funkcja kończąca program
        int   21h
        .STACK   512
END     Start
```

Program ten wyświetla na ekranie komunikat "Hello, world". Zadanie to realizuje funkcja **PrintMsg**. Instrukcja **EXTRN** na początku programu informuje asembler, że symbol **PrintMsg** jest procedurą zdefiniowaną w innym pliku źródłowym.
A oto plik, który zawiera funkcję **PrintMsg**:

```
; Plik PRINTMSG.ASM
; Plik ten używa danych zdefiniowanych w pliku poprzednim
          .MODEL   SMALL
          .DATA
          EXTRN   Mesg1:BYTE
          .CODE
          PUBLIC   PrintMsg
PrintMsg  PROC
          mov   dx, OFFSET Mesg1   ; Pobierz adres danych
                                   ; znajdujących się
                                   ; w innym pliku
          mov   ah, 9                ; Funkcja wyświetlająca
          int   21h                ; łańcuch znaków
          ret
PrintMsg  ENDP
END
```

Plik ten używa dyrektywy **PUBLIC** do określenia, że symbol **PrintMsg** powinien stać się widoczny również z innych modułów. By utworzyć program wykonywalny, należy skompilować oba zbiory źródłowe i skonsolidować je komendą:

```
tlink main printmsg
```

SYMBOLE WSPÓLNE

Używając dyrektyw **EXTRN** i **PUBLIC**, zawsze jeden z modułów jest traktowany jako źródło danych, a drugi jako ich użytkownik. Czasami jednak nie ma potrzeby tak wyraźnego rozgraniczania źródła i użytkownika danych. W tym wypadku można użyć symboli wspólnych, które są jednocześnie publiczne i zewnętrzne. Do tworzenia tego typu symboli używana jest dyrektywa **COMM** o formacie:

```
COMM   zmienna1:typ[:rozmiar] [,zmienna2:typ[:rozmiar]]...
```

Pole typ oznacza to samo co w dyrektywie **EXTRN**. Opcjonalnie podawany argument rozmiar oznacza liczbę elementów, z których składa się zmienna. Jeśli nie zostanie on podany, to przyjmuje się, że zmienna składa się z jednego elementu.
Dyrektywy **EXTRN** i **PUBLIC** mogą pojawić się w dowolnym miejscu programu (jednakże dyrektywa **EXTRN** musi poprzedzać użycie deklarowanego przez siebie symbolu). Natomiast dyrektywa **COMM** musi pojawić się w segmencie, zawierającym zmienne wspólne. Ponadto segment musi być segmentem publicznym oraz tak samo zdefiniowanym we wszystkich modułach, które używają tych zmiennych. Najczęściej jest to segment deklarowany dyrektywą **.CODE**.

Kompilacja warunkowa

Dzięki kompilacji warunkowej można z jednego pliku źródłowego stworzyć kilka wersji programu. By ją wykorzystać definiuje się klika symboli, które opisują program. Są one następnie używane w wyrażeniach warunkowych, które sterują przebiegiem kompilacji. Kompilacja warunkowa jest często stosowana w modułach, które mają być wykorzystywane przez programy działające w różnych modelach pamięci. W różnych modelach pamięci różne są przecież rozmiary wskaźników, a więc do operacji na nich musi zostać zastosowany różny kod. Poniższy przykład używa predefiniowanych symboli **DATASIZE** i **CODESIZE** do określenia wymaganego rozmiaru wskaźników.

```
          .MODEL   SMALL    ; Poniższy kod będzie poprawny
                            ; również w pozostałych modelach
          .DATA
IF DATASIZE
Ptr       DD   ?            ; Dla danych w modelach dużych
                            ; użyj dwóch
ELSE
Ptr       DW   ?            ; w przeciwnym razie -
                            ; jednego słowa
ENDIF
          .CODE
          .
          .
          .
IF DATASIZE
          les   bx, Ptr     ; Pobierz segment
                            ; i przemieszczenie
          mov   ax, es:[bx]; Pobierz daną wskazywaną
                            ; przez wskaźnik
ELSE
          mov   bx, Ptr     ; Tylko przemieszczenie
          mov   ax, [bx]    ; Pobierz daną wskazywaną
                            ; przez wskaźnik
ENDIF
          .
          .
          .
END
```

Kod, znajdujący się pomiędzy dyrektywami **IF** i **ENDIF** jest kodem warunkowym. Jeśli warunek jest spełniony, to kompilowany jest kod przed **ELSE**, w przeciwnym razie kod pomiędzy dyrektywami **ELSE** i **ENDIF**. Instrukcja **ELSE** nie jest obowiązkowa. W tym ypadku kod pomiędzy **IF** a **ELSE** jest kompilowany, gdy warunek jest prawdziwy.
Oprócz **IF** istnieje jeszcze kilka innych dyrektyw warunkowych. Jest też kilka wersji instrukcji **ELSE**. Tabela 10-1 zwiera opis wszystkich dyrektyw kompilacji warunkowej.

Tabela 10-1. Dyrektywy kompilacji warunkowej

Dyrektywa	Opis
ELSE	Rozpoczyna drugą część dyrektywy warunkowej. Kod pomiędzy ELSE a ENDIF będzie skompilowany, gdy warunek będzie fałszywy.
ELSEIF[warunek]	Każda z dyrektyw IF może być łączona i tworzyć listę warunków. Tylko jedna z jej części będzie wykorzystana (ta, która pierwsza okaże się prawdziwa). IF wyrażenie1 ... ; Kompiluj, gdy wyrażenie1 jest prawdziwe ELSEIF wyrażenie2 ... ; Kompiluj, gdy wyrażenie2 jest prawdziwe ELSEIFDEF etykieta ... ; Kompiluj, gdy etykieta jest zdefiniowana ENDIF
ENDIF	Koniec bloku warunkowego.
IF wyrażenie	Kompiluj poniższe instrukcje, gdy wyrażenie jest prawdziwe.
IF1	Kompiluj poniższe instrukcje przy pierwszym przejściu asemblera.
IF2	Kompiluj poniższe instrukcje przy drugim przejściu asemblera.
IFB arg	Kompiluj poniższe instrukcje, gdy argument jest łańcuchem pustym.
IFDEF etykieta	Kompiluj poniższe instrukcje, gdy etykieta została zdefiniowana.
IFDIF arg1,arg2	Kompiluj poniższe instrukcje, gdy argumenty są różne.
IFDIFI arg1,arg2	Podobnie jak IFDIF, z tym że wielkość liter argumentów nie ma znaczenia.
IFE wyrażenie	Kompiluj poniższe instrukcje, gdy wartość wyrażenia jest równa zero.
IFIDN arg1,arg2	Kompiluj poniższe instrukcje, gdy argumenty są identyczne.
IFIDNI arg1,arg2	Podobnie jak IFIDN, z tym że wielkość liter argumentów nie ma znaczenia.
IFNB arg	Kompiluj poniższe instrukcje, gdy argument nie jest łańcuchem pustym.
IFNDEF etykieta	Kompiluj poniższe instrukcje, gdy etykieta nie została jeszcze zdefiniowana.

Włączanie plików

Jednym z kroków, jaki należy podjąć przy wykorzystywaniu kompilacji warunkowej do tworzenia kilku wersji programu z jednego pliku źródłowego, jest zdefiniowanie symboli, które sterują jej przebiegiem. Często te same symbole muszą być definiowane w wielu plikach. Każda ich zmiana wymagałaby, więc, przejrzenia wszystkich tych plików. Nie jest to sposób zbyt wygodny, ale na szczęście można temu zaradzić.

Rozwiązać ten problem można wykorzystując pliki włączane. Plik taki jest zwykłym plikiem źródłowym, ale jego zawartość w czasie kompilacji może zostać wstawiona do

innego pliku. Każdą część pliku źródłowego można umieścić w pliku włączanym, a następnie zastąpić ją dyrektywą:

```
INCLUDE [ŚCIEŻKA\]NAZWA_PLIKU[.ROZSZERZENIE]
```

NAZWA_PLIKU jest nazwą włączanego pliku. Jeśli nie zostanie podana do niego ścieżka dostępu, to Turbo Assembler zacznie poszukiwanie go w katalogach podanych po opcji **/I**, a następnie w katalogu bieżącym. Jeśli nie zostanie podane rozszerzenie pliku, wówczas asembler przyjmie je jako .ASM. W plikach włączanych może również wystąpić dyrektywa **INCLUDE**. Zagnieżdżanie to może być dowolnie głębokie.
Pliki włączane są najbardziej użyteczne, gdy instrukcje, które się w nich znajdują, używane są w wielu modułach. Bardzo często są to instrukcje definiujące stałe i łańcuchy np. dyrektywą **EQU**. Dość często umieszcza się również w nich definicje struktur i deklaracje zmiennych globalnych programu.
Dyrektywa **GLOBAL** jest przeznaczona do deklarowania danych w plikach włączanych. Jej składnia jest taka sama jak dyrektywy **COMM**, ale nie musi ona znajdować się w segmencie danych. Wymaga ona natomiast, by w jednym z modułów źródłowych, używających pliku włączanego, w którym się ona znajduje, została przydzielona pamięć dla zmiennych przez nią deklarowanych. W module tym dyrektywa **GLOBAL** działa jak dyrektywa **PUBLIC**, zaś w pozostałych jako dyrektywa **EXTRN**.
W poniższym przykładzie dane globalne zostały zadeklarowane dyrektywą **GLOBAL**. Dyrektywy kompilacji warunkowej zostały natomiast wykorzystane do sprawdzenia, czy rzeczywiście tylko jeden plik źródłowy przydziela zmiennej pamięć.

```
        .MODEL   SMALL          ; Model pamięci zostaje
                ; już zdefiniowany
GLOBAL  Symbol1:BYTE
GLOBAL  Aword:WORD
GLOBAL  Mesg:BYTE
IFDEF   MAIN
Symbol1 DB   ?
Aword   DW   1234h
Mesg    DB   "Oto ciąg znaków",0
```

By wykorzystać ten plik, w jednym z modułów, przed jego włączeniem, musi zostać zdefiniowany symbol "MAIN":

```
MAIN      EQU 1             ; Symbol MAIN musi zostać
                            ; zdefiniowany tylko w jednym
                            ; z modułów źródłowych.
                            ; w pozostałych musi pozostać nie
                            ; zdefiniowany. Oczywiście wartość
                            ; mu nadana nie ma żadnego
                            ; znaczenia, ważne jest natomiast
                            ; to czy symbole ten istnieje.
INCLUDE   DATADEFS.INC      ; w tym miejscu zostanie wstawiona
                            ; zawartość pliku
          .CODE
```

```
mov   ax, Aword ; Pobierz wartość zmiennej Aword
mov   Symboll, al; Prześlij jej młodszy bajt
                 ; do zmiennej Symboll
.
.
```

Dzielenie modułu na procedury

Gdy został już dokonany podział programu na moduły, to należy się zastanowić, jakie procedury zostaną w modułach umieszczone. Oczywiście powinny one odpowiadać tematowi modułu. Na przykład, funkcje w module obsługi ekranu powinny zajmować się właśnie tylko tym zadaniem.
Niektóre funkcje będą mogły być wywoływane również z innych modułów. By uczynić je widocznymi w innych modułach powinny one zostać zadeklarowane dyrektywą **PUBLIC**. Pozostałe podprogramy mogą być wykorzystywane tylko w tym module, w który zostały zdefiniowane. W przykładzie modułu z procedurami obsługi ekranu, funkcja, która wyświetla łańcuch powinna być publiczna. Natomiast funkcja, która przechowuje dane o bieżącej pozycji kursora i atrybutach ekranu, powinna być prywatna.
Po stworzeniu całego modułu może on być traktowany tylko jako zbiór symboli publicznych, które wykonują określone zadanie. Szczegóły implementacyjne dla reszty programu przestają być ważne. Jest to bardzo istotne, gdy moduł ma być stosowany w wielu różnych programach. Może się przecież okazać, że moduł obsługi ekranu doskonale odpowiada wymaganiom innego programu. Jeśli został on poprawnie zaprojektowany, to aby móc wyświetlić ciąg znaków, konieczne jest tylko wywołanie odpowiedniej jego funkcji. Sposób przechowywania pozycji kursora i atrybutów ekrany przestaje być istotny.

Definiowanie procedur

Dyrektywa **PROC** oznacza początek definicji procedury. W najprostszej formie podaje się jej nazwę oraz typ (bliska czy daleka):

Tryb MASM:

```
NazwaFunkcji   PROC   NEAR   ; Można podać typ NEAR lub FAR
```

Tryb IDEAL:

```
PROC   NazwaFunkcji   NEAR   ; Można podać typ NEAR lub FAR
```

Poniższy przykład prezentuje prostą procedurę, która dodaje dwie długie liczby całkowite:

```
; Dodaj liczbę w AX,BX do liczby zawartej w CX,DX
AddLong   PROC  NEAR
          add   bx, dx   ; Dodanie młodszych słów
          adc   ax, cx   ; Dodanie starszych słów
          ret
AddLong   ENDP
```

Jak pokazuje przykład, za dyrektywą **PROC** występuje ciało procedury. Jej koniec definiuje dyrektywa **ENDP**. Aby nie było wątpliwości, do której procedury odnosi się **ENDP**, jako jej argument można podać nazwę kończonej funkcji. W większości wypadków nazwa nie

jest jednak konieczna. (Dyrektywy **PROC** i **ENDP** muszą się uzupełniać. Przez podanie nazwy procedury określamy, która dyrektywa **PROC** odpowiada napotkanej właśnie dyrektywie **ENDP**).
Jeśli na początku programu wystąpiła dyrektywa **.MODEL**, to nie jest konieczne określanie typu procedury. W modelach MEDIUM, LARGE i HUGE są one traktowane jako FAR; zaś w modelach SMALL i COMPACT zaś jako NEAR. Przyjęte domniemanie może być zmienione przez jawne podanie typu procedury jako FAR lub NEAR. Używa się tego często przy deklarowaniu prywatnych funkcji w modelu LARGE. Jeśli cały kod modułu mieści się w 64K pamięci lub używany jest tylko jeden segment kodu, to wszystkie prywatne procedury mogą zostać zdefiniowane jako NEAR. Powoduje to zmniejszenie rozmiaru kodu i zwiększenie szybkości działania programu, gdyż wywołania bliskie zajmują o dwa bajty pamięci mniej i wymagają od 6 do 13 taktów zegara mniej na ich wykonanie. W modelach MEDIUM, LARGE i HUGE funkcje publiczne powinny być natomiast zadeklarowane jako FAR. Korzystające z nich bowiem podprogramy z innych modułów, umieszczone są przecież w różnych segmentach, konieczne więc będą wywołania dalekie.

ZACHOWYWANIE WARTOŚCI REJESTRÓW

Jedna z podstawowych zasad programowania strukturalnego głosi, że każda funkcja powinna mieć tak mało efektów ubocznych, jak to tylko jest możliwe. W językach wysokiego poziomu efekty te ograniczają się tylko do zmiennych globalnych. W języku asemblera efekty te dotyczą nie tylko zmiennych globalnych, ale także rejestrów procesora. Problem ten jest dość ważny, gdyż procesory 80x86 nie mają ich zbyt wielu.
By zapobiec tego typu efektom ubocznym, można wszystkie rejestry, używane przez funkcję, zachować na stosie. Ilustruje to przykład:

```
        EXTRN   FileName:BYTE
        EXTRN   Handle:WORD
Open    PROC    FAR
        push  ax                ; Zachowaj rejestr AX
        push  dx                ; Zachowaj rejestr DX
        .                       ; Ciało funkcji
        .
        .
        pop   dx                ; Odtwórz rejestr DX
        pop   ax                ; Odtwórz rejestr AX
        ret
Open    ENDP
```

Zachowywanie i odtwarzanie rejestrów może być wykonywane automatycznie przez asembler. Dyrektywa **PROC** posiada klauzulę **USES**, w której można podać nazwy wszystkich, używanych w funkcji, rejestrów. Poprzedni przykład wyglądałby następująco:

```
        EXTRN   FileName:BYTE
        EXTRN   Handle:WORD
Open    PROC    FAR   USES ax dx
                                ; Tutaj asembler umieści
                                ; instrukcje PUSH
```

```
        .                           ; Ciało funkcji
        .
        .
                                    ; Tutaj asembler umieści
                                    ; instrukcje POP
        ret
Open    ENDP
```

W obu przykładach zostanie wygenerowany ten sam kod. W drugim z nich programista zostaje natomiast zwolniony z obowiązku zdjęcia ze stosu, odłożonych nań wcześniej rejestrów. Troszczy się o to asembler, zapewniając poprawność kodu na wejściu i na wyjściu funkcji.

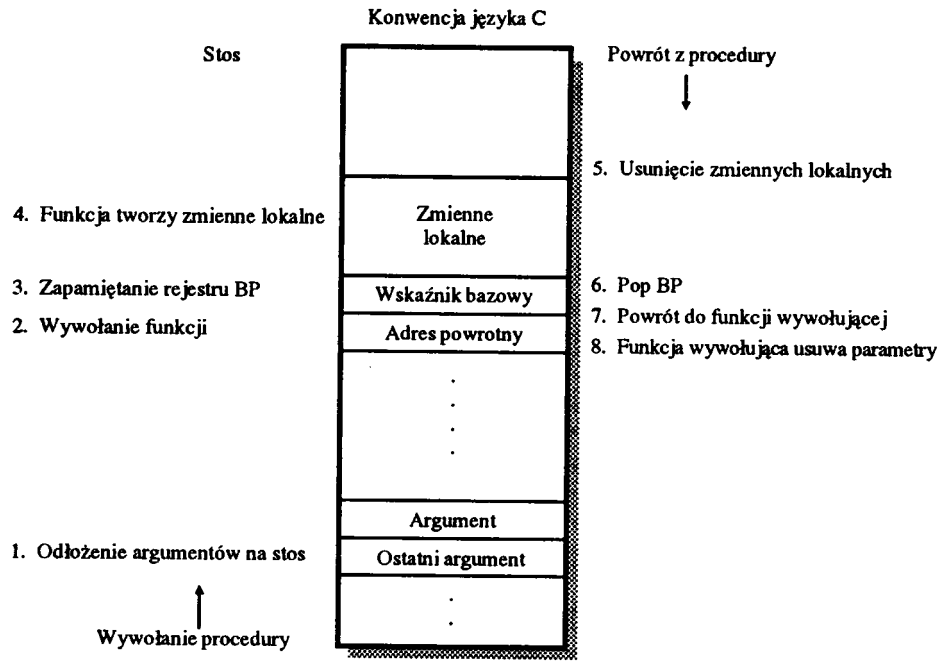
PRZEKAZYWANIE PARAMETRÓW I ZWRACANIE REZULTATÓW

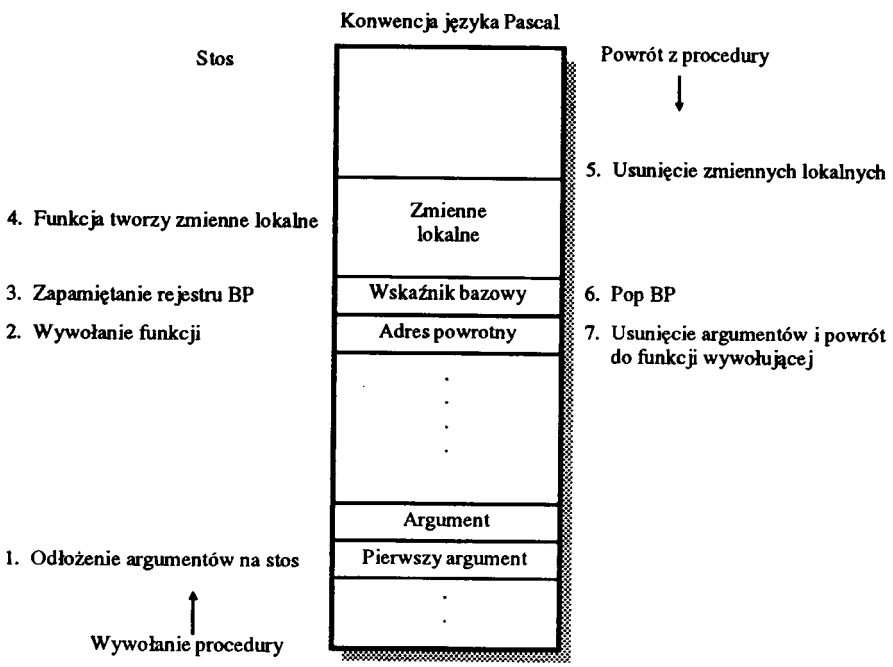
Funkcja w poprzednim przykładzie nie jest zbyt użyteczna, ponieważ otwiera tylko plik o nazwie zawartej w zmiennej **FileName**. By stała się ona bardziej uniwersalna, program wywołujący powinien podawać jej nazwę otwieranego pliku. Poza tym funkcja ta powinna mieć możliwość zwrócenia deskryptora otworzonego pliku.

Do przekazania wskaźnika do nazwy pliku może zostać wykorzystana ramka stosu. Rozwiązanie to jest powszechnie stosowane w językach wysokiego poziomu takich jak Pascal lub C. Niektóre z nich używają również ramek do zwracania rezultatów ich działania. Inne natomiast wykorzystują w tym celu rejestry AX i DX. Rysunek 10-1 pokazuje sposoby wykorzystania ramek stosu w różnych językach.

W poniższym przykładzie przedstawiono funkcję, która używa do przekazania parametrów, sposobu stosowanego w języku C.

```
        .
        .
        .
        mov     ax, 2
        push    ax                  ; Prześlij tryb otwierania
                                    ; pliku
        mov     ax, OFFSET Msg1     ; Prześlij na stos adres
                                    ; łańcucha
        push    ax
        call    Open
        add     sp, 4               ; Usuń parametry ze stosu
        mov     Handle, ax          ; Zapamiętaj zwrócony
                                    ; deskryptor
        .
        .
        .
Open    PROC    FAR
        push    bp
        mov     bp, sp
        mov     dx, [bp+6]      ; Nazwa pliku
        mov     ah, 3Dh         ; Funkcja, która otwiera
```





Rysunek 10-1. ***Ramki stosu używane w językach wysokiego poziomu.***

```
                                ; podany plik
        mov     al, [bp+8]      ; Otworzenie do czytania
                                ; i pisania
        int     21h             ; Wywołanie tej funkcji MS-DOS-a
                                ; powoduje tylko zmianę
                                ; rejestru AX
        jnc     NoErr
        mov     ax, -1          ; Zwrócenie -1 oznacza
                                ; wystąpienie błędu
        pop     bp
        ret
Open    ENDP
```

Pierwsza część przykładu pokazuje, w jaki sposób należy umieszczać argumenty na stosie, wywołać funkcję i przywrócić początkowy stan stosu. Funkcja **Open** ilustruje natomiast sposób odwoływania się do przekazanych parametrów. Najpierw zapamiętuje ona na stosie starą wartość rejestru BP, a następnie wpisuje do niego zawartość wskaźnika stosu. Oznacza to, że pierwszy argument znajduje się sześć bajtów od miejsca, na które wskazuje BP (dwa bajty na starą wartość BP i cztery na adres powrotu). Za pomocą wyrażenia **[bp+6]** można się więc do niego odwołać.

Do utworzenia ramki stosu służy dyrektywa **ARG**. Ponadto pozwala ona używać symbolicznych nazw argumentów. Zamiast pisania wyrażenia w rodzaju **bp+6**, można odwołując się do niego, podać się jego nazwę. Powoduje to, że program staje się bardziej czytelny. Funkcja **Open**, z użyciem dyrektywy **ARG**, wyglądałaby następująco:

```
Open    PROC    FAR
; Następna linia powoduje utworzenie odpowiedniej
; ramki stosu
ARG     FileName:WORD, OpenMode:WORD
        push    bp
        mov     bp, sp
        mov     dx, [FileName]  ; Nazwa pliku
        mov     ah, 3Dh         ; Funkcja, która otwiera
                                ; podany plik
        mov     al, [OpenMode]  ; Otworzenie do czytania
                                ; i pisania
        int     21h             ; Wywołanie tej funkcji
                                ; MS-DOS-a powoduje tylko
                                ; zmianę rejestru AX
        jnc     NoErr
        mov     ax, -1          ; Zwrócenie -1 oznacza
                                ; wystąpienie błędu
        pop     bp
        ret
Open    ENDP
```

Oczywiście działanie tego programu jest identyczne jak poprzedniego. Tutaj jednak nie potrzeba pamiętać położenia odpowiednich argumentów. Zamiast ich adresów użyte zostały ich nazwy symboliczne **FileName** i **OpenMode**.
Ramka stosu jest również używana do przechowania zmiennych lokalnych. Aby je stworzyć, należy odpowiednio zmodyfikować zawartość rejestru wskaźnika stosu. Dostęp do nich odbywa się również poprzez rejestr BP jak w następującym przykładzie:

```
Func1   PROC
        push  bp
        mov   bp, sp
        sub   sp, 4          ; Zarezerwowanie na stosie
                             ; czterech bajtów
        xor   ax, ax
        mov   [bp-2], ax     ; Wyzeruj pierwszą zmienną
        mov   [bp-4], ax     ; Wyzeruj drugą zmienną
        .
        .
        .
```

Turbo Assembler sam potrafi utworzyć zmienne lokalne. Służy do tego dyrektywa **LOCAL**. Jest ona podobna do dyrektywy **ARG**. Gdy posłużymy się nią poprzedni przykład, będzie wyglądać następująco:

```
Func1   PROC
LOCAL   VAR1:WORD, VAR2:WORD
        xor   ax, ax
        mov   [VAR1], ax       ; Wyzeruj zmienną VAR1
        mov   [VAR2], ax       ; Wyzeruj zmienną VAR2
        .
        .
        .
```

Makrodefinicje

W rozdziale 9 zostało pokazane, jak za pomocą dyrektywy **EQU** można utworzyć symbol oznaczający łańcuch. Ten rodzaj substytucji jest prostą formą makrodefinicji. Makrodefinicje mogą zastępować całe fragmenty kodu lub danych. Mogą one mieć, podobnie jak funkcje, argumenty i zmienne lokalne.
Makrodefinicje posiadają wiele zalet funkcji, a także wnoszą wiele własnych istotnych cech. Użycie makrodefinicji jest bowiem szybsze niż odpowiadającej jej funkcji, ponieważ makrodefinicja nie używa rozkazów wywołania i powrotu. Kompilacja warunkowa, zastosowana w makrodefinicjach, pozwala na ich optymalizację bez generowania żadnego dodatkowego kodu.
Każde wywołanie makrodefinicji jest zamieniane przez asembler na odpowiadający mu kod lub dane. Jeśli więc jest ona wywoływana kilka razy, to potrzebuje więcej miejsca niż odpowiadająca jej funkcja. Programista powinien więc, zależnie od wymagań (szybkość działania lub ograniczenia pamięci), wybrać odpowiednią wersję.

DEFINIOWANIE MAKRODEFINICJI

Definicja makro kojarzy jego nazwę z kodem lub danymi, które będą zastępowały każde pojawienie się tej nazwy w programie. Na przykład makro, ktore wywołuje funkcję systemową (numer 4Ch), kończącą program, może wyglądać następująco:

```
DOSEXIT  MACRO            ; Podanie nazwy makra
         mov   ah, 4Ch    ; Numer funkcji w AH
         int   21h        ; Wywołanie MS-DOS-a
         ENDM             ; Koniec makrodefinicji
```

Dyrektywa **ENDM** jest używana do zaznaczenia końca makrodefinicji. Gdy makro zostało już zdefiniowane, to każde wystąpienie jego nazwy w programie będzie zastępowane przez asembler jego ciałem. Sama makrodefinicja nie generuje jeszcze żadnego kodu lub danych. Często stosowane makrodefinicje mogą zostać umieszczone w plikach włączanych i być wykorzystywane w wielu programach. Natomiast tylko te, które zostaną rzeczywiście w programie użyte, wygenerują odpowiadający im kod lub dane.

ARGUMENTY MAKRODEFINICJI

Jedną z makrodefinicji, którą wielu programistów umieszcza w pliku włączanym jest ta, która realizuje wywołanie systemu DOS. By wywołać DOS należy przesłać do rejestrów numer żądanej funkcji oraz jej parametry. Tak więc makro, dokonujące takiej operacji, musi znać wartości, które mają zostać umieszczone w rejestrach. Aby je mu przesłać, należy zdefiniować makro z argumentami. Zostało to pokazane w następującym przykładzie:

```
CALLDOS  MACRO  FuncNumb, DXData  ; Argumenty
                                  ; makrowywołania
         mov    ah, FuncNumb      ; Numer funkcji do AH
         mov    dx, DXData        ; Dane do DX
         int    21h               ; Wywołanie systemu
         ENDM
```

Korzystając z tego makrowywołania, należy podać numer funkcji oraz jej parametr:

```
CALLDOS   41h, <OFFSET FileName>  ; Usuń plik FileName
```

Użycie nawiasów trójkątnych (<>) w drugim argumencie zapobiega próbom obliczenia jego wartości przed przesłaniem do makrowywołania. W tabeli 10-2 podano wszystkie operatory dotyczące makrodefinicji.

Tabela 10-2. Operatory stosowane w makrodefinicjach

Operator	Opis
&	Wymuszenie podstawienia. Używane jest wewnątrz znaku cudzysłowu oraz, gdy argument znajduje się zaraz obok innego symbolu. Wywołanie MAKEMSG 4,<Disk error> makrodefinicji postaci: MAKEMSG MACRO NUM, TEXT MSG&NUM DB '&TEXT' ENDM wygeneruje napis MSG4 DB 'Disk error'
<>	Literał tekstowy. Zapobiega próbom obliczania wartości łańcucha przed przekazaniem do makroinstrukcji.
!	Występujący po nim znak nie jest traktowany jako operator: MAKEMSG 5,<A string with a !> in it>
%	Obliczenie wyrażenia i przekazanie jego wartości do makroinstrukcji: MAKEMSG %3+4,<This is MSG7> W przykładzie tym liczby 3 i 4 są najpierw dodawane, a rezultat (7) jest przekazywany jako pojedynczy argument.
;;	Komentarz. Komentarz ten nie będzie wstawiany w rozwijanym tekście: MAKEMSG MACRO NUM, TEXT MSG&NUM DB '&TEXT' ;; nie zapamiętuj tego komentarza ENDM

ETYKIETY LOKALNE

Każde pojawienie się wywołania makroinstrukcji powoduje zastąpienie jej generowanym przez nią kodem. Pewien problem powstaje wówczas, gdy makroinstrukcja zawiera etykietę. Przy drugim jej wywołaniu asembler napotka taką samą etykietę, a to spowoduje powstanie błędu. Są dwa sposoby utworzenia etykiet lokalnych, i tym samym uniknięcia błędu wielokrotnej definicji.

W pierwszym z nich do stworzenia etykiety lokalnej wykorzystywana jest dyrektywa **LOCAL**. Jest to ta sama dyrektywa, która służyła w procedurach do tworzenia zmiennych lokalnych. Poniższy przykład pokazuje sposób jej użycia:

```
WAIT    MACRO   Delay
LOCAL   Tloop           ; Tloop jest etykietą lokalną
        mov   ah, 0     ; Pobierz licznik zegara
        int   1Ah       ; Wywołaj funkcję BIOS-u, która
                        ; obsługuje zegar
        add   ax, Delay ; Wyznaczenie czasu końca przerwy
        adc   dx, 0     ; Dodanie przeniesienia do
                        ; starszej części
        mov   si, ax    ; Zapamiętaj czas zakończenia
                        ; (młodsze słowo)
        mov   di, dx    ; Zapamiętaj czas zakończenia
```

```
                                ; (starsze słowo)
Tloop:
        mov     ah, 0           ; Kontynuuj sprawdzanie zegara
        int     1Ah
        cmp     dx, di          ; Porównaj starsze części
        jb      Tloop           ; Jeśli są one równe,
        cmp     ax, si          ; porównaj młodsze
        jb      Tloop
```

Innym sposobem utworzenia etykiet lokalnych jest użycie dyrektywy **LOCALS** i zaznaczenie ich lokalnym przedrostkiem. Zwykle jest to przedrostek **@@**, ale może on zostać zmieniony dyrektywą **LOCALS**. Odwołanie tworzenia zmiennych lokalnych realizuje dyrektywa **NOLOCALS**. Po zastosowaniu tych dyrektyw makroinstrukcja WAIT będzie wyglądała następująco:

```
LOCALS      $$                  ; Etykiety lokalne zaczynają
                                ; się przedrostkiem $$
WAIT        MACRO   Delay
            mov     ah, 0       ; Pobierz licznik zegara
            int     1Ah         ; Wywołaj funkcję BIOS-u,
                                ; która obsługuje zegar
            add     ax, Delay   ; Wyznaczenie czasu końca przerwy
            adc     dx, 0       ; Dodanie przeniesienia do
                                ; starszej części
            mov     si, ax      ; Zapamiętaj czas zakończenia
                                ; (młodsze słowo)
            mov     di, dx      ; Zapamiętaj czas zakończenia
                                ; (starsze słowo)
$$Tloop:
            mov     ah, 0       ; Kontynuuj sprawdzanie zegara
            int     1Ah
            cmp     dx, di      ; Porównaj starsze części
            jb      Tloop       ; Jeśli są one równe,
            cmp     ax, si      ; porównaj młodsze
            jb      Tloop
```

PĘTLA REPT

Pętle w makroinstrukcjach mogą służyć do tworzenia powtarzających się fragmentów kodu lub danych. W Turbo Assemblerze istnieją trzy różne typy pętli. Pierwszą z nich jest pętla **REPT** (repeat). Jej argumentem jest liczba, która określa ilość obiegów pętli. Najczęściej dyrektywa ta stosowana jest do tworzenia i inicjowania tablic danych. W poniższym przykładzie dyrektywa **REPT** została użyta do utworzenia tablicy o określonym rozmiarze, który zawiera ciąg bajtów, zainicjowanych wartościami z zakresu od **Start** do **Start+Size**:

```
RANGE   MACRO   Start, Size
Value   =       Start           ; Wartość początkowa
        REPT    Size            ; Powtórzenie Size razy
```

```
        DB      Value           ; Zdefiniowanie bajtu
                                ; zainicjowanego wartością
                                ; Value
Value   =       Value+1         ; Zwiększenie Value
        ENDM                    ; Koniec pętli
    ENDM
; Koniec makrodefinicji
```

Do zakończenia pętli również służy dyrektywa **ENDM**. Musi ona kończyć każdą makrodefinicję i pętlę. Napotkana dyrektywa **ENDM** odnosi się do ostatnio otworzonej pętli lub dyrektywy **MACRO**.

Powyższa makroinstrukcja tworzy listę dyrektyw **DB**, każda z inną wartością. Należy zauważyć, że zamiast dyrektywy **EQU** został użyty znak równości (=). Dzięki niemu symbol może występować po obu stronach instrukcji przypisania. Oto kilka przykładów wywołania tej makroinstrukcji:

```
LowAlpha    LABEL   BYTE        ; Utworzenie etykiety danych
            RANGE   <'a'>, 26   ; Definicja 26 bajtów od
                                ; wartości początkowej 'a'
UpAlpha     LABEL   BYTE        ; Tablica dużych liter
            RANGE   <'A'>, 26
```

PĘTLA IRP

Dwa pozostałe rodzaje pętli pozwalają na podanie listy danych, które zostaną następnie w nich wykorzystane. Dyrektywa **IRP** generuje kod lub dane dla każdego podanego jej argumentu. Dyrektywa **IRP** może zostać użyta do stworzenia kodu, który oblicza wartość wielomianu. Ogólna postać wielomianu jest następująca:

$$C_1x^n+C_2x^{n-1}+...+C_{n+1}x^0,$$

gdzie C_1, C_2, itd są współczynnikami wielomianu. Każdy składnik wyrażenia zawiera zmienną x podnoszoną do różnych potęg. Każdy wielomian można jednak zdefiniować, podając tylko wartości jego współczynników. Należy tylko uporządkować wszystkie składniki w kolejności malejących potęg, a każdej nie występującej potędze przypisać współczynnik równy 0. Na przykład, wielomian:

$$10x^4 + 5x^2 + 12x + 1$$

może być zdefiniowany za pomocą listy:

$$10, 0, 5, 12, 1$$

Za pomocą dyrektywy **IRP** można utworzyć makroinstrukcję, która oblicza wartości wielomianu:

```
; Makrodefinicja, która generuje kod, obliczający wartość
; wielomianu. Należy jej podać wartość zmiennej, dla
; której ma być obliczona wartość wielomianu
; i jego współczynniki.
POLY    MACRO   x, Coef     ; Podaj wartość zmiennej
                            ; i współczynników wielomianu
```

```
        mov     cx, 1           ; CX zawiera x do pewnej potęgi
                                ; Najpierw zawiera on x do zerowej
        xor     bx, bx          ; Suma składników
        IRP     Trm, Coef
        mov     ax, Trm         ; Pobierz współczynnik, który
                                ; stoi przy składniku
        imul    cx              ; Pomnóż przez x do odpowiedniej
                                ; potęgi
        add     bx, ax          ; Dodaj składnik do sumy
        mov     ax, x           ; Prześlij x do AX
        imul    cx              ; Pomnóż przez x by otrzymać
                                ; następną potęgę tej zmiennej
        mov     cx, ax
        ENDM                    ; Koniec pętli
        ENDM                    ; Koniec makrodefinicji
```

Współczynniki muszą zostać podane w kolejności odwrotnej niż się to podaje w zwykłej notacji. Wynika to stąd, że pętla zwiększa potęgę zmiennej x poprzez wielokrotne mnożenie. W obliczeniach makroinstrukcja używa rejestrów AX, BX, CX i DX. Jeśli ich wartości nie powinny zostać zmienione, to przed wykonaniem pętli powinny one zostać przesłane na stos, a po jej zakończeniu pobrane z niego. Również żaden z nich nie może być pierwszym argument tej makroinstrukcji. Wynika to stąd, że każdy z nich jest przez nią używany. Niestety zapominanie o tym ograniczeniu może spowodować powstanie bardzo trudnego do znalezienia błędu. Dlatego też na początku tego przykładu zostało umieszczone stosowne ostrzeżenie. Aby wygenerować kod, który oblicza wartości przykładowego wielomianu, należy wywołać powyższą makroinstrukcję w następujący sposób:

```
POLY    Var1, < 1, 12, 5, 0, 10 >
```

PĘTLA IRPC

Dyrektywa **IRPC** jest bardzo podobna do **IRP**, z tym że jej lista jest listą znaków, a nie listą liczb. Używając tej dyrektywy można napisać makroinstrukcję, która szuka pewnego znaku na ich liście. Może ona zostać następnie wykorzystana do stwierdzenia, czy znak, podany przez użytkownika, znajduje się na liście poleceń.

```
; Odszukanie znaku Char w liście Cmnds. Char musi być
; 8-bitowym rejestrem lub komórką pamięci. Makroinstrukcja
; zwraca w BX numer komendy lub -1.
; Wcześniej musi zostać wykonana dyrektywa LOCALS, która
; ustawia przedrostek na @@
PARSE     MACRO   Char, Cmnds
          xor     bx, bx          ; Wyzeruj BX
          IRPC    x, Cmnds        ; Przejrzenie wszystkich
                                  ; możliwych komend
          cmp     Char, x         ; Czy znaki są identyczne
          je      @@Match         ; Koniec pętli, gdy tak
          inc     bx              ; Zwiększ indeks
```

```
        ENDM
@@Match:
        cmp     bx, SIZESTR     ; Czy BX przekroczył koniec
                                ; łańcucha
        jb      @@Found         ; Jeśli nie znaleziono znak
        mov     bx, -1          ; w przeciwnym razie ustaw
                                ; BX na -1
@@Found:
        ENDM
```

KOMPILACJA WARUNKOWA W MAKROINSTRUKCJACH

Makroinstrukcja **CALLDOS** generuje kod, który ładuje rejestr DX nawet wtedy, gdy nie jest to konieczne. Makroinstrukcja **POLY** generuje kod również i dla współczynników równych zeru. Oczywiście obie one działają poprawnie, ale kod przez nie wygenerowany nie jest optymalny. W wielu wypadkach, stosując dyrektywy kompilacji warunkowej, można znacznie poprawić efektywność generowanego kodu. i tak makroinstrukcja **CALLDOS** mogła by być przekształcona na:

```
CALLDOS     MACRO   FuncNumb, DXData    ; Argumenty
                                        ; makrowywołania
            mov     ah, FuncNumb        ; Numer funkcji do AH
IFNB        <DXData>                    ; Sprawdzenie, czy
                                        ; argument
                                        ; DXData nie jest pusty
            mov     dx, DXData          ; Dane do DX
ENDIF
            int     21h                 ; Wywołanie systemu
            ENDM
```

W przykładzie tym, gdy nie zostanie podany drugi argument, kod, który ładuje rejestr DX, nie zostanie wygenerowany.
A oto optymalna wersja makroinstrukcji **POLY**:

```
; Makrodefinicja, która generuje kod obliczający wartość
; wielomianu. Należy jej podać wartość zmiennej, dla
; której ma być obliczona wartość wielomianu
; i jego współczynniki.
POLY    MACRO   x, Coef     ; Podaj wartość zmiennej i
                            ; współczynników wielomianu
        mov     cx, 1       ; CX zawiera x do pewnej potęgi
                            ; Najpierw zawiera on x do zerowej
        xor     bx, bx      ; Suma składników
        IRP     Trm, Coef
IF Trm                      ; Sprawdzenie czy Trm nie
                            ; jest równe 0
        mov     ax, Trm     ; Pobierz współczynnik, który
                            ; stoi przy składniku
```

```
        imul    cx          ; Pomnóż przez x do
                            ; odpowiedniej potęgi
        add     bx, ax      ; Dodaj składnik do sumy
ENDIF
        mov     ax, x       ; Prześlij x do AX
        imul    cx          ; Pomnóż przez x by otrzymać
                            ; następną potęgę tej zmiennej
        mov     cx, ax
        ENDM                ; Koniec pętli
        ENDM                ; Koniec makrodefinicji
```

Ponieważ rezultatem mnożenia przez 0 jest 0, to nie ma potrzeby wykonywania obliczeń dla zerowych współczynników. Dzięki dyrektywom warunkowym instrukcje mnożenia i dodawania nie są wtedy umieszczane w generowanym kodzie.

Podsumowanie

Dzięki technikom programowania strukturalnego pisanie programów może być bardziej efektywne, a one same są bardziej przejrzyste i zrozumiałe. Podzielenie programu na moduły pozwala mieć przed oczyma cały jego obraz bez wgłębiania się w zbędne szczegóły. Następnym poziomem w strukturze programu są funkcje i makroinstrukcje. By wykorzystać dobrze zaprojektowaną funkcję, wystarczy tylko wiedzieć, co ona robi. Natomiast sposób w jaki jej zadanie jest realizowane to już następny poziom w strukturze programu.
Dyrektywy **PUBLIC**, **EXTRN**, **COMM** oraz **GLOBAL** zapobiegają niepożądanym interakcjom między modułami. Tylko symbole, określone przez programistę, są widoczne w innych modułach. Dzięki temu powstanie efektów ubocznych ograniczone jest tylko do tych właśnie symboli.
Dobrze określony interfejs funkcji z otoczeniem jest niezbędną cechą poprawnego programowania strukturalnego. Przy jego definicji może być wykorzystana dyrektywa **PROC**. Należy jednak pamiętać, że w językach wysokiego poziomu istnieje wiele rodzajów interakcji.
Innym sposobem podzielenia złożonych zadań na etapy jest zastosowanie makroinstrukcji. Różnica między makro a funkcją polega na tym, że w miejscu wywołania makroinstrukcji umieszczany jest kod lub dane przez nią generowane, a w miejscu wywołania funkcji rozkaz procesora, który przekazuje sterowanie do odpowiedniej części programu. Makroinstrukcje są najczęściej stosowane tam, gdzie szczególnie ważna jest szybkość działania programu, a mniej ważna jest ilość przezeń zajętej pamięci.

ARG

ZASTOSOWANIE

Zdefiniowanie argumentów procedury.

SKŁADNIA

```
ARG Nazwa_Argumentu[:[zakres] [PTR] Typ]
[,Nazwa_Argumentu[:[zakres] [PTR] Typ]]...[=Rozmiar]
[RETURNS Nazwa_Argumentu[:[zakres] [PTR] Typ]
[,Nazwa_Argumentu[:[zakres] [PTR] Typ]...]
```

Nazwa_Argumentu	Symboliczna nazwa argumentu funkcji.
zakres	Może być NEAR lub FAR. Oznacza, że argument jest wskaźnikiem
PTR	Dzięki niemu Turbo Assembler dołącza specjalną informację dla Turbo Debuggera.
Typ	Typ argumentu może być określony jako WORD, DWORD, FWORD, PWORD, QWORD, TBYTE lub może być nazwą struktury.
Rozmiar	Oznacza rozmiar poprzedzającego go argumentu.
RETURNS	Podany argument po zakończeniu funkcji nie zostanie usunięty ze stosu.

KOMENTARZ

Dzięki dyrektywie **ARG** wiele szczegółów, związanych z przekazywaniem argumentów poprzez stos, przejmuje Turbo Assembler. Zapewnia ona również zgodność tak deklarowanej funkcji z funkcjami powstałymi w języku określonym w dyrektywie **.MODEL**. Każdy argument, umieszczony na stosie identyfikowany jest poprzez przemieszczenie względem rejestru BP. Dzięki dyrektywie **ARG** nie istnieje już potrzeba używania wyrażeń takich jak BP+8 przy odwoływaniu się do argumentów. Zamiast tego podaje się nazwę argumentu, zdefiniowaną w tej dyrektywie.

W części języków wysokiego poziomu np. W C, argumenty umieszczane są na stosie w kolejności od strony prawej do lewej, podczas gdy w innych np. W Pascalu od strony lewej do prawej. Również i te różnice uwzględnia dyrektywa **ARG**. Argumenty w niej wyspecyfikowane powinny więc odpowiadać kolejności, w jakiej występowałyby w języku wysokiego poziomu.

PATRZ

LOCAL
PROC
USES

PRZYKŁAD

```
; Poniższa funkcja z języka C może być wywołana
; w następujący sposób:
; Func(AnInt, AString, AChar);
```

COMM

```
        .MODEL    SMALL, C
        .CODE
_Func   PROC
ARG     AnInt:WORD, AString:PTR BYTE, AChar:BYTE
        mov   ax, [AnInt]     ; Pobranie pierwszego
                              ; argumentu ([bp+4])
        mov   di, [AString]   ; Drugiego ([bp+6])
        mov   cl, [AChar]     ; Trzeciego ([bp+8])
        .
        .
        .
        ret
_Func   ENDP
```

COMM

ZASTOSOWANIE

Zdefiniowanie zmiennej współdzielonej przez kilka modułów.

SKŁADNIA

```
COMM [zakres] nazwa:typ [:ilość]
      [, [zakres] nazwa:typ [:ilość]]...
```

zakres	Może być NEAR lub FAR, zależnie od tego czy symbol znajduje się w grupie DGROUP czy też nie.
nazwa	Nazwa zmiennej.
Typ	Typ zmiennej może być określony jako WORD, DWORD, FWORD, PWORD, QWORD, TBYTE lub może być nazwą struktury.
ilość	Określa liczbę elementów danych, z których składa się symbol.

KOMENTARZ

Dyrektywa **COMM** powoduje przydzielenie zmiennej pamięci. Oznacza to, że każdy moduł, który używa tej zmiennej, może użyć tej samej definicji, zamiast stosowania dyrektyw **EXTRN** lub **PUBLIC**. Dzięki temu dyrektywa **COMM** doskonale nadaje się do zastosowania w plikach włączanych.

Należy tylko upewnić się, że dyrektywa **COMM** została umieszczona we właściwym segmencie. Assembler bowiem przyjmuje, że do zmiennej przez nią definiowanej można się odwołać, wykorzystując rejestr DS. Nie zawsze tak, co może prowadzić do powstania błędów.

Przed użyciem zmiennych, utworzonych dyrektywą **COMM**, program musi dokonać ich inicjalizacji. Sposób tworzenia zmiennych już zainicjowanych został przedstawiony przy opisie dyrektywy **GLOBAL**.

PATRZ

GLOBAL
EXTRN
PUBLIC

PRZYKŁAD

Plik nagłówkowy, który zawiera definicje zmiennych dokonane dyrektywą **COMM**:

```
; DATA.INC      plik z definicjami zmiennych
COMM Name:BYTE:20      ; 20 bajtowe pole na nazwę pliku
COMM Dscrpt:WORD       ; Deskryptor pliku
COMM TRec:TransRec     ; Rekord do transmisji (Należy
                       ; zapewnić, aby definicja tej
                       ; struktury wystąpiła przed
                       ; włączeniem tego pliku)
```

Moduł ten inicjalizuje zmienną Dscrpt:

```
          .MODEL   SMALL
          .DATA
INCLUDE   DATA.INC              ; Tutaj zostanie włączony
                                ; plik nagłówkowy
FileName  DB   "TRANS.DAT",0
          .CODE
OpenFile  PROC
          mov   dx, OFFSET FileName
          mov   al, 2           ; Plik do czytania/pisania
          mov   ah, 3Dh         ; Funkcja otwierająca
                                ; podany plik
          int   21h
          jnc   OpenOK
          mov   ax, -1          ; Wystąpienie błędu oznaczane
                                ; jest zwróceniem wartości -1
OpenOK:
          mov   Dscrpt, ax      ; Prześlij AX do zmiennej
                                ; zadeklarowanej jako COMM
          ret
OpenFile  ENDP
```

W poniższym przykładzie z pliku, identyfikowanego przez zmienną Dscrpt, odczytywany jest rekord:

```
          .MODEL   SMALL
          .DATA
INCLUDE   DATA.INC
          .CODE
ReadRec   PROC
          mov   bx, Dscrpt          ; Pobierz deskryptor
                                    ; pliku
```

```
        mov     cx, TYPE TransRec ; Pobierz rozmiar rekordu
        mov     dx, OFFSET TRec   ; Adres bufora TRec
                                  ; znajduje się w parze
                                  ; rejestrów DS:DX
        mov     ah, 3Fh           ; Odczyt danych z pliku
        int     21h
        ret
ReadRec ENDP
```

ELSE

ZASTOSOWANIE

Zaznaczenie początku bloku alternatywnego w instrukcji kompilacji warunkowej.

SKŁADNIA

```
IF warunek
      .          Kompiluj te linie, gdy warunek jest prawdziwy
      .
      .
ELSE
      .          Kompiluj te linie, gdy warunek jest fałszywy
      .
      .
ENDIF
```

KOMENTARZ

Blok **ELSE** jest opcjonalny, a może wystąpić po dowolnej z dyrektyw warunkowych IF. Wszystkie instrukcje, które znajdują się pomiędzy **ELSE** a **ENDIF**, zostaną użyte, gdy podany warunek nie jest prawdziwy.

PATRZ

IF
IF1
IF2
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI
IFNB

IFNDEF

PRZYKŁAD

Patrz przykład do **IF**.

ELSEIF, ELSEIF1, ELSEIF2, ELSEIFB, ELSEIFDEF, ELSEIFDIF ELSEIFE, ELSEIFIDN, ELSEIFNB, ELSEIFNDEF

ZASTOSOWANIE

Zaznaczenie początku zagnieżdżonego bloku warunkowego.

SKŁADNIA

`ELSEIF wyrażenie`	*Kompiluj poniższe instrukcje, gdy wyrażenie jest prawdziwe.*
`ELSEIF1`	*Kompiluj poniższe instrukcje przy pierwszym przejściu asemblera.*
`ELSEIF2`	*Kompiluj poniższe instrukcje przy drugim przejściu asemblera.*
`ELSEIFB <arg>`	*Kompiluj poniższe instrukcje, gdy argument jest łańcuchem pustym.*
`ELSEIFDEF symbol`	*Kompiluj poniższe instrukcje, gdy symbol został zdefiniowany.*
`ELSEIFDIF <arg1>,<arg2>`	*Kompiluj poniższe instrukcje, gdy argumenty są różne.*
`ELSEIFDIFI <arg1>,<arg2>`	*Podobnie jak IFDIF, z tym że wielkość liter argumentów nie ma znaczenia.*
`ELSEIFE wyrażenie`	*Kompiluj poniższe instrukcje, gdy wartość wyrażenia jest równa zero.*
`ELSEIFIDN <arg1>,<arg2>`	*Kompiluj poniższe instrukcje, gdy argumenty są identyczne.*
`ELSEIFIDNI <arg1>,<arg2>`	*Podobnie jak IFIDN, z tym że wielkość liter argumentów nie ma znaczenia.*
`ELSEIFNB <arg>`	*Kompiluj poniższe instrukcje, gdy argument nie jest łańcuchem pustym.*
`ELSEIFNDEF symbol`	*Kompiluj poniższe instrukcje, gdy symbol nie został jeszcze zdefiniowany.*

ENDIF

KOMENTARZ

Pomiędzy dyrektywami **IF** i **ENDIF** może znajdować się wiele dyrektyw **ELSEIF**. Jeśli warunek **IF** jest fałszywy, wtedy będzie obliczany każdy warunek po **ELSEIF**, aż do napotkania warunku prawdziwego. Po skompilowaniu odpowiadającego mu bloku, lub gdy żaden z warunków nie zostanie spełniony, asembler przechodzi do dyrektywy **ENDIF**.

PATRZ

IF
IF1
IF2
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI
IFNB
IFNDEF

PRZYKŁAD

Patrz przykład do **IF**.

ENDIF

ZASTOSOWANIE

Zaznaczenie końca bloku warunkowego.

SKŁADNIA

```
IF warunek
          .
          .
          .
ELSE
          .
          .
          .
ENDIF
```

KOMENTARZ

Każdej dyrektywie **IF** musi odpowiadać jedna dyrektywa **ENDIF**. Jeśli warunek, który występuje po **IF** nie jest prawdziwy, to asembler przechodzi do odpowiedniej instrukcji

ELSE lub **ENDIF**. Jeśli natomiast jest on prawdziwy, wówczas wszystkie instrukcje pomiędzy **IF** a **ELSE** zostaną skompilowane, a następnie zostanie wykonana dyrektywa **ENDIF**. Jeśli instrukcja **IF** znajdzie się w bloku, który ma być przez asembler ominięty, najpierw zostanie odszukana odpowiadająca jej dyrektywa **ENDIF**, a dopiero potem **ELSE** lub **ENDIF**, które odnoszą się do właśnie kompilowanej dyrektywy warunkowej. Dzięki temu można uniknąć niejednoznaczności, które dyrektywy **ELSE** lub **ENDIF** odpowiadają danej dyrektywie **IF**.

PATRZ

IF
IF1
IF2
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI
IFNB
IFNDEF
ELSE
ELSEIF
ELSEIF1
ELSEIF2
ELSEIFB
ELSEIFDEF
ELSEIFDIF
ELSEIFDIFI
ELSEIFE
ELSEIFIDN
ELSEIFIDNI
ELSEIFNB
ELSEIFNDEF

PRZYKŁAD

Patrz przykład do **IF**.

ENDM

ZASTOSOWANIE

Zaznaczenie końca pętli lub makrodefinicji.

ENDP

SKŁADNIA

```
nazwa     MACRO [argumenty]
           .
           .
           .
           ENDM          Koniec makrodefinicji
           REPT licznik
           .
           .
           .
           ENDM          Koniec pętli
          IRP zmienna, <lista>
          .
          .
          .
          ENDM           Koniec pętli
          IRPC zmienna, <łańcuch>
          .
          .
          .
          ENDM           Koniec pętli
```

KOMENTARZ

Każda pętla i każda makrodefinicja musi być zakończona dyrektywą **ENDM**. Gdy pętle są zagnieżdżone w makrodefinicjach lub w innych pętlach, to pierwsza dyrektywa **ENDM** odnosi się do ostatniej pętli.

PATRZ

MACRO
REPT
IRP
IRPC

PRZYKŁAD

Patrz przykład do **MACRO**.

ENDP

ZASTOSOWANIE

Zakończenie definicji funkcji, którą rozpoczęlismy dyrektywą **PROC**.

SKŁADNIA

Tryb MASM:

```
nazwa       PROC        Początek definicji funkcji
            .
            .
            .
[nazwa]     ENDP
```

Tryb IDEAL:

```
PROC        nazwa
            .
            .
            .
ENDP        [nazwa]
```

KOMENTARZ

Dla każdej dyrektywy **PROC** musi istnieć odpowiadająca jej dyrektywa **ENDP**, która kończy definicję funkcji. Nazwa funkcji w dyrektywie **ENDP** może zostać pominięta. Asembler w tym wypadku przyjmie nazwę, która występuje w odpowiedniej dyrektywie **PROC**. Bloki PROC/ENDP mogą być zagnieżdżone.

PATRZ

PROC

PRZYKŁAD

Patrz przykład do **PROC**.

EXITM

ZASTOSOWANIE

Wcześniejsze zakończenie makrodefinicji lub pętli.

SKŁADNIA

```
EXITM
```

KOMENTARZ

Dyrektywa **EXITM** jest używana najczęściej w instrukcjach warunkowych do zakończenia pętli lub makrodefinicji po spełnieniu określonego warunku. Może być użyta do pominięcia niepotrzebnej już części makrodefinicji lub do symulacji pętli while.

PATRZ

MACRO

EXITM

REPT
IRP
IRPC

PRZYKŁAD

```
; w makrodefinicji dyrektywa EXITM została wykorzystana
; do pominięcia już niepotrzebnej części makro.
AddNums   MACRO   arg1, arg2          ; Makroinstrukcja,
                                      ; która dodaje dwa
                                      ; argumenty
          mov     ax, WORD PTR arg1   ; Pobierz pierwszy
                                      ; argument
          add     ax, WORD PTR arg2   ; Dodaj do niego drugi
IF (TYPE arg1) LT 4                   ; Czy argumenty są
                                      ; słowami
          EXITM
ENDIF
          mov     dx, WORD PTR HIGH arg1
          adc     dx, WORD PTR HIGH arg2
          ENDM
; Symulacja pętli while
MyMacro   MACRO   SizeData, Start
@@Size1   DB      ?
Numb      =       Start
          REPT    SizeData        ; Ilość obiegów pętli
                                  ; równa maksymalnie
                                  ; SizeData
IF Numb GE 128                    ; Rozmiar danych nie może
                                  ; być większy niż 128
          EXITM
ENDIF
          DB      Numb            ; Stworzenie zmiennej
Numb      =       Numb + 1
Last      =       $               ; Zapamiętanie wskaźnika
                                  ; pozycji
          ENDM
          ORG     @@Size1         ; Cofnięcie wskaźnika
                                  ; pozycji na początek
          DB      Numb - Start    ; Umieszczenie aktualnego
                                  ; rozmiaru
          ORG     Last            ; Przemieszczenie się na
                                  ; koniec bloku danych
          ENDM
```

EXTRN

ZASTOSOWANIE

Utworzenie zmiennej, zdefiniowanej w innym module, ale dostępnej w module, który zawiera dyrektywę **EXTRN**.

SKŁADNIA

```
EXTRN nazwa:typ[:ilość] [, nazwa:typ[:ilość]]...
```

nazwa	Nazwa zmiennej.
typ	Typ zmiennej może być określony jako NEAR, FAR, PROC, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE lub może być nazwą struktury.
ilość	Określa liczbę elementów danych, z których składa się symbol.

KOMENTARZ

Dyrektywy **PUBLIC** i **EXTRN** określają zmienne widoczne w wielu modułach. Dyrektywa **EXTRN** określa symbole zdefiniowane w innych modułach. Jedyną dostarczaną przez nią informacją o zmiennej jest jej typ i rozmiar. Do programisty natomiast należy ustalenie, by we wszystkich modułach dane te sobie odpowiadały.

Dyrektywa **EXTRN** może być umieszczona wewnątrz segmentu jak i poza nim. Jeśli znajduje się ona wewnątrz niego, to asembler przyjmuje, że również definiowana zmienna znajduje się w tym segmencie. Jeśli natomiast dyrektywa ta jest umieszczona poza segmentem, wówczas asembler przyjmie, że wszystkie konieczne zmiany segmentów dokonane zostaną przez programistę.

PATRZ

PUBLIC
COMM
GLOBAL

PRZYKŁAD

W poniższym pliku zadeklarowanych zostało kilka publicznych symboli:

```
          .MODEL   LARGE
          PUBLIC   AnInt, Astring, Print, ClearScrn
          .DATA
AnInt     DW   10
AString   DB   "The string", 0Dh, 0Ah, '$'
CLSstr    DB   "[2J$"                ; Komenda ANSI do
                                     ; czyszczenia ekranu
          .CODE
; Wyświetl łańcuch spod adresu DS:DX
Print     PROC
          mov   ah, 9                ; Funkcja, która
                                     ; wyświetla łańcuch
```

```
                                        ; znaków
          int   21h
          ret
Print     ENDP
; Wyczyszczenie ekranu komendą ANSI
ClearScrn PROC
          mov   ah, 9
          mov   dx, OFFSET CLSstr ; Wskazanie na łańcuch
                                  ; komendy
          int   21h
          ret
ClearScrn ENDP
```

Poniższy plik używa symboli, które są zdefiniowane w pliku powyżej:

```
          .MODEL   LARGE
EXTRN     AnInt:WORD
EXTRN     AString:BYTE:12
EXTRN     Print:PROC
EXTRN     ClearScrn:PROC
          .DATA
ASPtr     DD   AString            ; Adres łańcucha
                                  ; zdefiniowanego w
                                  ; innym module
          .CODE
Start     PROC
          mov   ax, SEG DGROUP    ; Pobierz adres segmentu
                                  ; danych
          mov   ds, ax            ; i umieść go w DS
          call  ClearScrn         ; Wywołanie funkcji
                                  ; zdefiniowanej w innym
                                  ; pliku źródłowym
          mov   cx, AnInt         ; Pobierz liczbę z
                                  ; innego pliku
BigLp:
          mov   dx, ASPtr         ; Pobierz adres łańcucha
          call  Print             ; Wyświetl go
          loop  BigLp             ; Powtórz 10 razy
          mov   ah, 4Ch           ; Funkcja kończąca
                                  ; program
          int   21h
Start     ENDP
          .STACK  1024
END  Start
```

@FileName, ??FileName

ZASTOSOWANIE

Pobranie nazwy bieżącego pliku źródłowego.

SKŁADNIA

```
@FileName
??FileName
```

KOMENTARZ

Każdy moduł znajduje się w swoim własnym pliku. Dyrektywa **@FileName** zwraca jego nazwę. Jest to tylko nazwa główna pliku bez ścieżki dostępu ani rozszerzenia. Jest ona zawsze ośmiobajtowa. Nie wypełnione przez nią znaki zostaną zastąpione spacjami.

PATRZ

NAME *Ustalenie nazwy modułu*

PRZYKŁAD

```
; Program wyświetla nazwę pliku źródłowego
            .MODEL SMALL
            .DATA
FName       DB   @FileName, 0Dh, 0Ah, '$'
            .CODE
Start:
            mov   ax, SEG DGROUP
            mov   ds, ax
            mov   dx, OFFSET FName   ; Pobierz adres
            mov   ah, 9              ; Funkcja, która
                                     ; wyświetla łańcuch
                                     ; znaków
            int   21h
            mov   ah, 4Ch            ; Funkcja, która
                                     ; kończy program
            int   21h
            .STACK  512
END Start
```

GLOBAL

ZASTOSOWANIE

Zdefiniowanie symbolu globalnego.

GLOBAL

SKŁADNIA

```
GLOBAL [zakres] nazwa:typ [:ilość]
      [, [zakres] nazwa:typ [:ilość]]...
```

zakres	Może być NEAR lub FAR, zależnie od tego czy symbol znajduje się w grupie DGROUP czy też nie.
nazwa	Nazwa zmiennej.
Typ	Typ zmiennej może być określony jako BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE lub może być nazwą struktury.
ilość	Określa liczbę elementów danych, z których składa się symbol.

KOMENTARZ

Dyrektywa **GLOBAL** działa jak kombinacja dyrektyw **PUBLIC** i **EXTRN**. W module, który definiuje symbol, odpowiada ona dyrektywie **PUBLIC**, udostępniając go innym modułom. W nich zaś działa ona jako **EXTRN**.

Dyrektywa **GLOBAL** jest często używana w plikach nagłówkowych. Odwrotnie niż **COMM**, może ona wystąpić w dowolnym miejscu programu przed użyciem definiowanej przez nią zmiennej. Ponieważ pliki nagłówkowe są włączane na początku każdego modułu, to całość działa poprawnie.

PATRZ

COMM
EXTRN
PUBLIC

PRZYKŁAD

```
; DATA.INC      plik z definicjami zmiennych
GLOBAL Name:BYTE:20     ; 20 bajtowe pole na nazwę pliku
GLOBAL Dscrpt:WORD      ; Deskryptor pliku
GLOBAL TRec:TransRec    ; Rekord do transmisji (Należy
                        ; zapewnić aby definicja tej
                        ; struktury wystąpiła przed
                        ; włączeniem tego pliku)
```

Moduł ten inicjalizuje zmienną **Dscrpt**:

```
INCLUDE   STRCDEFS.INC          ; Plik, który zawiera
                                ; definicje struktur
INCLUDE   DATA.INC              ; Tutaj zostanie włączony
                                ; plik nagłówkowy
          .MODEL   SMALL
          .DATA
FileName  DB   "TRANS.DAT",0
          .CODE
OpenFile  PROC
          mov   dx, OFFSET FileName
          mov   al, 2           ; Plik do czytania/pisania
```

```
            mov     ah, 3Dh         ; Funkcja, która otwiera
                                    ; podany plik
            int     21h
            jnc     OpenOK
            mov     ax, -1          ; Wystąpienie błędu
                                    ; oznaczane jest
                                    ; zwróceniem wartości -1
OpenOK:
            mov     Dscrpt, ax      ; Prześlij AX do zmiennej
                                    ; zadeklarowanej jako COMM
            ret
OpenFile    ENDP
```

W poniższym przykładzie z pliku identyfikowanego przez zmienną **Dscrpt** odczytywany jest rekord:

```
INCLUDE     STRCDEFS.INC
INCLUDE     DATA.INC
            .MODEL    SMALL
            .CODE
ReadRec     PROC
            mov     bx, Dscrpt          ; Pobierz deskryptor
                                        ; pliku
            mov     cx, TYPE TransRec   ; Pobierz rozmiar
                                        ; rekordu
            mov     dx, OFFSET TRec     ; Adres bufora TRec
                                        ; znajduje się w parze
                                        ; rejestrów DS:DX
            mov     ah, 3Fh             ; Odczyt danych z pliku
            int     21h
            ret
ReadRec     ENDP
```

IF, IF1, IF2, IFB, IFDEF IFDIF, IFDIFI, IFE, IFIDN IFIDNI, IFNB, IFNDEF

ZASTOSOWANIE

Kompilacja odpowiednich bloków programu na podstawie wartości pewnych warunków.

SKŁADNIA

`IF wyrażenie` *Kompiluj poniższe instrukcje, gdy wyrażenie jest prawdziwe.*

IF1	*Kompiluj poniższe instrukcje przy pierwszym przejściu asemblera.*
IF2	*Kompiluj poniższe instrukcje przy drugim przejściu asemblera.*
IFB <arg>	*Kompiluj poniższe instrukcje, gdy argument jest łańcuchem pustym.*
IFDEF symbol	*Kompiluj poniższe instrukcje, gdy symbol został zdefiniowany.*
IFDIF <arg1>,<arg2>	*Kompiluj poniższe instrukcje, gdy argumenty są różne.*
IFDIFI <arg1>,<arg2>	*Podobnie jak IFDIF, z tym że wielkość liter argumentów nie ma znaczenia.*
IFE wyrażenie	*Kompiluj poniższe instrukcje, gdy wartość wyrażenia jest równa zero.*
IFIDN <arg1>,<arg2>	*Kompiluj poniższe instrukcje, gdy argumenty są identyczne.*
IFIDNI <arg1>,<arg2>	*Podobnie jak IFIDN, z tym że wielkość liter argumentów nie ma znaczenia.*
IFNB <arg>	*Kompiluj poniższe instrukcje, gdy argument nie jest łańcuchem pustym.*
IFNDEF symbol	*Kompiluj poniższe instrukcje, gdy symbol nie został jeszcze zdefiniowany.*

KOMENTARZ

Dyrektywy warunkowe używane są do tworzenia różnego rodzaju kodu zależnie od pewnych okoliczności. Dzięki nim z jednego pliku źródłowego można uzyskać kilka wersji tego samego programu. Natomiast zastosowanie kompilacji warunkowej w makrodefinicjach pozwala na zoptymalizowanie kodu generowanego przez nie.

Na blok warunkowy składają się wszystkie instrukcje, które występują pomiędzy odpowiadającymi sobie dyrektywami **IF** i **ENDIF**. W szczególności mogą to być dyrektywy **ELSE** lub **ELSEIF**, dzielące go na kilka mniejszych bloków. W strukturze tej mogą występować również zagnieżdżone instrukcje **IF**.

PATRZ

ELSE
ELSEIF
ENDIF

PRZYKŁAD

```
PrnErr    MACRO   Row, Col, Msg
IFB Row                          ; Jeśli nie podano numeru
                                 ; wiersza,
          mov   dh, 24           ; przyjmij wiersz ostatni
ELSEIF Row LE 24                 ; Czy podano prawidłowy
                                 ; numer wiersza
          mov   dh, Row
ELSE
IF1                              ; Wyświetl tylko przy
                                 ; pierwszym
```

```
          DISPLAY "Invalid Row"    ; przebiegu kompilacji
ENDIF
ENDIF
IFB Col                            ; Jeśli nie podano numeru
                                   ; kolumny,
          xor   dl, dl             ; przyjmij kolumnę
                                   ; pierwszą
ELSEIF (Col + (SIZESTR Msg) LE 79  ; Czy podany komunikat
                                   ; zmieści się w linii
                                   ; ekranu
          mov   dl, Col
ELSE
IF1                                ; Wyświetl tylko przy
                                   ; pierwszym
          DISPLAY "Invalid Column"; przebiegu kompilacji
ENDIF
ENDIF
          xor   bh, bh             ; Ustaw 0 stronę ekranu
          mov   ah, 2              ; Funkcja, która
                                   ; przemieszcza kursor
          int   10h                ; Wywołania BIOS-a
          mov   dx, OFFSET @@TMsg
          mov   ah, 9              ; Funkcja, która
                                   ; wyświetla łańcuch
                                   ; znaków
          int   21h
          jmp   short @@Skip       ; Ominięcie bajtów
                                   ; komunikatu
@@TMsg    DB    Msg                ; Tutaj jest umieszczany
                                   ; komunikat
          DB    '$'                ; Znak końca łańcucha
@@Skip:
          ENDM
Func      PROC
          .
          .
          .
IFDEF     PrnErr                   ; Makroinstrukcja PrnErr
                                   ; zostanie wywołana tylko
                                   ; wtedy, gdy została
                                   ; zdefiniowana
          PrnErr   10, 10, <Error #1>
ELSE                               ; w przeciwnym razie
                                   ; zostanie wywołana
                                   ; funkcja
          mov   ah, ErrNumb1
```

```
        push  ax
        call  FPrnErr
        add   sp, 2
ENDIF
```

INCLUDE

ZASTOSOWANIE

Włączenie zawartości pliku nagłówkowego do właśnie kompilowanego pliku.

SKŁADNIA

```
INCLUDE NazwaPliku
```

KOMENTARZ

W plikach nagłówkowych najczęściej znajdują się definicje często używanych struktur, makroinstrukcji lub stałych. Umieszczenie tych informacji w plikach nagłówkowych zapobiega upychaniu ich w plikach źródłowych. Oznacza to więc, że zmiana, dokonana w takim pliku, nie wymaga przejrzenia wszystkich zbiorów źródłowych.

Argumentem tej dyrektywy jest nazwa pliku nagłówkowego. Jeśli nie zostanie podane rozszerzenie, to asembler przyjmie je jako .INC. Jeśli nie zostanie podana ścieżka dostępu do pliku, wówczas będzie on poszukiwany najpierw w katalogach podanych w opcji /I asemblera, a następnie w katalogu bieżącym.

Zawartość pliku nagłówkowego jest umieszczana w miejscu wystąpienia dyrektywy **INCLUDE**. Tak więc może ona zostać włączona w dowolnej części programu.

PATRZ

INCLUDELIB

PRZYKŁAD

Patrz przykład do **COMM**.

INCLUDELIB

ZASTOSOWANIE

Przekazanie informacji do konsolidatora o dołączeniu do programu podanej biblioteki.

SKŁADNIA

Tryb MASM:

```
INCLUDELIB NazwaPliku
```

Tryb IDEAL:

```
INCLUDELIB "NazwaPliku"
```

KOMENTARZ

Nazwa pliku jest nazwą ładowanej biblioteki. Standardowym rozszerzeniem tego typu plików jest .LIB. Jeśli nie zostanie podana ścieżka dostępu do pliku, to będzie on poszukiwany najpierw w katalogach podanych w opcji **/I** asemblera, a następnie w katalogu bieżącym.
Zastosowanie dyrektywy **INCLUDELIB** jest alternatywą do podania jej nazwy bezpośrednio w komendzie dla konsolidatora. Jeśli jednak dyrektywa ta została użyta, wówczas nazw bibliotek podanych w niej nie powinno się umieszczać w rozkazie dla programu łączącego. Spowoduje to bowiem powstanie błędu wielokrotnej definicji symboli.

PATRZ

INCLUDE

PRZYKŁAD

```
INCLUDELIB EMU.LIB    ; Będzie wykorzystywana biblioteka
                      ; EMU.LIB, która zawiera emulator
                      ; zmiennoprzecinkowy
```

IRP

ZASTOSOWANIE

Wygenerowanie bloku instrukcji dla każdej z wartości na liście.

SKŁADNIA

```
IRP   zmienna, <arg [,arg]...>
      instrukcje
ENDM
```

zmienna	Symbol, który będzie zastąpiony każdą z podanych wartości
arg	Dane, które w każdym obiegu pętli są podstawiane za zmienną
instrukcje	Blok instrukcji, który zostanie powtórzony dla każdej danej z listy

KOMENTARZ

Pętla **IRP** jest podobna do pętli **REPT**. Różnica polega na tym, że w dyrektywie **IRP** można określić dla każdego obiegu pętli inną daną. Dzięki temu każda iteracja może generować różny kod. Dane mogą być symbolami, łańcuchami albo liczbami.

PATRZ

IRPC
REPT

IRPC

PRZYKŁAD

```
        .MODEL   SMALL
        .DATA
SaveM   DW   4 dup(?)
        .CODE
        .
        .
        .
Tmp     =   0                   ; Licznik
IRP reg, <ax, bx, cx, dx>       ; Pętla dla każdego rejestru
        mov   SaveM+Tmp, reg    ; Zapamiętaj wartość rejestru
Tmp     =   Tmp + 2             ; Uaktualnij adres
ENDP
        .
        .
        .
```

IRPC

ZASTOSOWANIE

Wygenerowanie bloku instrukcji dla każdego znaku podanego łańcucha.

SKŁADNIA

```
IRPC zmienna, łańcuch
     instrukcje
ENDM
```

zmienna	Symbol, który będzie zastąpiony każdą z podanych wartości
łańcuch	Łańcuch, zwierający znaki, które w każdym obiegu pętli są podstawiane za zmienną
instrukcje	Blok instrukcji, który zostanie powtórzony dla każdego znaku łańcucha

KOMENTARZ

Dyrektywa **IRPC** jest bardzo podobna do dyrektywy **IRP**. Różnią się one tylko tym, że IRPC zamiast listy zmiennych, używa znaków łańcucha. Dla każdego z nich będzie powtórzony blok instrukcji. Znaki te zastąpią każde pojawienie się zmiennej w ciele pętli.

PATRZ

REPT
IRP

PRZYKŁAD

```
; Utworzenie listy zmiennych o nazwach Licz_a, Licz_e ...
IRPC   letter, <aeiou>
Licz_&letter   DW   0
ENDM
```

LOCAL

ZASTOSOWANIE

Zdefiniowanie zmiennych lokalnych procedury lub makroinstrukcji.

SKŁADNIA

Dla makroinstrukcji:

```
LOCAL Symbol [,Symbol]
```

Dla procedur:

```
LOCAL nazwa:typ [:ilość] [,nazwa:typ [:ilość]]...
```

Symbol	Nazwa lokalnego symbolu makroinstrukcji.
nazwa	Nazwa zmiennej lokalnej procedury.
typ	Typ symbolu może być określony jako BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE lub może być nazwą struktury.
ilość	Określa liczbę elementów danych, z których składa się symbol.

KOMENTARZ

Dyrektywa **LOCAL** w makroinstrukcjach sprawia, że symbol staje się dla nich lokalny. Przed użyciem musi on jednak zostać najpierw zdefiniowany.
Natomiast w procedurach, dyrektywa **LOCAL** ponadto, że czyni symbol lokalnym, przydziela mu jeszcze pamięć w ramce stosu. Asembler, po dodaniu rozmiarów wszystkich zmiennych lokalnych, umieszcza w kodzie programu instrukcję, odpowiednio modyfikującą wskaźnik stosu.
Do każdej zmiennej lokalnej można odwołać się wykorzystując rejestr bazowy BP z odpowiednim przemieszczeniem. Na przykład, pierwsza zmienna ma adres [bp-2]. Natomiast użycie nazwy symbolu lokalnego, powoduje, że asembler automatycznie zastąpi go odpowiednim wyrażeniem.

PATRZ

PROC
MACRO
ARG

LOCALS, NOLOCALS

PRZYKŁAD

```
; Funkcja, która używa trzech zmiennych lokalnych.
MyFunc  PROC
LOCAL   AnInt:WORD, AnArray:WORD:14, AStr:PTR WORD
        xor   ax, ax
        mov   AnInt, ax     ; Wyzeruj zmienną
        mov   cx, 14
        lea   di, AnArray   ; Pobierz adres tablicy
        rep   stosw         ; Wyzeruj ją
        mov   cx, -1
        mov   si, AStr      ; Wskazanie na łańcuch
        rep   scansb        ; Wyszukaj zerowy bajt
        not   cx            ; CX jest długością łańcucha
        .
        .
        .
MyFunc  PROC
; Symbole AnInt, AnArray, AStr są nie zdefiniowane
; poza tą funkcją.
```

LOCALS, NOLOCALS

ZASTOSOWANIE

Udostępnienie, zablokowanie

SKŁADNIA

```
LOCALS [przedrostek]
NOLOCALS
```

przedrostek Dwuznakowa sekwencja używana jako przedrostek symboli lokalnych. Jeśli nie zostanie on jawnie określony, to asembler domniemywa wtedy przedrostek @@.

KOMENTARZ

Jeśli symbole lokalne są dostępne, wówczas symbole, poprzedzone przedrostkiem, stają się lokalne dla procedur, makroinstrukcji, unii i struktur, które je zawierają. Przedrostek ten musi być poprawnym początkiem nazwy symbolu. Tak więc na przykład, "12" nie może być użyty jako poprawny przedrostek.

Dyrektywa **NOLOCALS** wyłącza stosowanie symboli lokalnych. Jeśli w dyrektywie **LOCALS** podany został przedrostek, to stanie się on przedrostkiem domniemanym przy następnym jej użyciu.

PATRZ

LOCAL
MACRO
PROC

PRZYKŁAD

```
LOCALS              ; Włączenie stosowania symboli lokalnych
AFunc     PROC
          .
          .
          .
@@Start:            ; Symbol lokalny funkcji AFunc
          .
          .
          .
AFunc     ENDP
BFunc     PROC
          .
          .
          .
@@Start:            ; Symbol ten jest różny od pozostałych
                    ; etykiet @@Start
          .
          .
BFunc     ENDP
```

MACRO

ZASTOSOWANIE

Zaznaczenie początku makrodefinicji.

SKŁADNIA

Tryb MASM:

```
nazwa MACRO [arg [,arg]...]
     instrukcje
ENDM
```

Tryb IDEAL:

```
MACRO nazwa [arg [,arg]...]
     instrukcje
ENDM
```

PROC

nazwa	Nazwa makrodefinicji.
arg	Argumenty makrodefinicji.
instrukcje	Blok instrukcji, który zastąpi każde wywołanie makrodefinicji.

KOMENTARZ

Każde pojawienie się nazwy makroinstrukcji w programie powoduje zastąpienie jej przez blok instrukcji podany w definicji. Instrukcje te mogą być dowolnymi poprawnymi instrukcjami Turbo Assemblera, generującymi kod, dane, dyrektywami lub instrukcjami kompilacji warunkowej. Każda dyrektywa **MACRO** musi posiadać odpowiadającą jej dyrektywę **ENDM**.

Do makroinstrukcji mogą zostać przekazane argumenty. Wtedy każde wystąpienie nazwy argumentu będzie zastąpione odpowiednią daną.

PATRZ

ENDM
PROC

PRZYKŁAD

```
; Definicja makroinstrukcji, która dodaje dwa argument
i umieszczającej wynik operacji w rejestrze ax. Wywołana
ona może być np. następująco:
;       AddM    < 14, dx >  ; Dodanie liczby 14 do
                            ; rejestru DX
AddM    MACRO   A, B
        mov     ax, a       ; Pobierz pierwszy argument do AX
        add     ax, B       ; Dodaj do niego drugi argument
        ENDM
```

PROC

ZASTOSOWANIE

Zaznaczenie początku definicji procedury.

SKŁADNIA

Tryb MASM:

```
nazwa PROC [zakres] [uses reg [, reg]]
```

Tryb IDEAL:

```
PROC nazwa [zakres] [uses reg [, reg]]
```

nazwa	Nazwa funkcji.
zakres	Może być określony jako NEAR lub FAR zależnie od sposobu wywoływania funkcji.

reg Rejestry, które powinny zostać odłożone na stos przed wykonaniem ciała funkcji.

KOMENTARZ

Kod, który występuje po dyrektywie **PROC**, stanowi ciało funkcji. Argument zakres określa typ użytej na jej zakończenie instrukcji powrotu. Poza tym definiuje on również rozmiar nazwy procedury. Każde odwołanie do tej nazwy będzie uwzględniać ten właśnie rozmiar. Jeśli zostanie użyta klauzula **USES**, to Turbo Assembler na początku procedury automatycznie wstawi kod, odkładający na stos podane rejestry, a na jej końcu kod, który odtwarza ich zawartość. Dzięki temu pozostaną one przez funkcję nie zmienione.
Instrukcje, zawarte pomiędzy dyrektywami **PROC** oraz **ENDP**, mogą zawierać odwołania do symboli lokalnych. Symbole te muszą się zaczynać odpowiednim przedrostkiem. Ponadto przed definicją funkcji w programie musi wystąpić słowo **LOCALS**.

PATRZ

ARG
LOCAL
LOCALS
ENDP

PRZYKŁAD

```
        .MODEL   SMALL
; Funkcja używa rejestrów SI i DI. Jej rozmiar
; wyznaczony jest przez dyrektywę .MODEL.
Func1   PROC USES SI, DI
        .
        .
        .
        ret                     ; Przekształcany na RETN lub
                                ; RETF zależnie od wybranego
                                ; modelu pamięci
Func1   ENDP
; Poniższa funkcja nie zachowuje wartości żadnego z
; rejestrów. Musi być wywoływana rozkazem dalekiego
; wywołania.
Func2   PROC FAR
        .
        .
        .
        ret                     ; Przekształcany na RETF
Func2   ENDP
Main    PROC                    ; Typ tej procedury zależny jest
                                ; od wybranego modelu pamięci
        .
        call  Func1             ; Wywołanie bliskie
        call  Func2             ; Wywołanie dalekie
```

```
        ret
Main    ENDP
```

PUBLIC

ZASTOSOWANIE

Uwidocznienie symbolu we wszystkich modułach.

SKŁADNIA

```
PUBLIC symbol [, symbol]
```

KOMENTARZ

Nazwy symboli publicznych przekazywane są przez Turbo Assembler do programu łączącego. Dzięki temu mogą być zrealizowane odwołania międzymodułowe. Symbole publiczne definiowane są dyrektywą **PUBLIC**. Symbol może być zmienną, etykietą lub nazwą funkcji.

Program łączący nie wnika w typy poszczególnych symboli. Do programisty należy sprawdzenie, czy ten sam symbol posiada w każdym z modułów ten sam typ. Niezgodność typów może bowiem doprowadzić do powstania błędów trudnych do wykrycia.

PATRZ

EXTRN
COMM
GLOBAL

PRZYKŁAD

Patrz przykład do **EXTRN**.

PURGE

ZASTOSOWANIE

Usunięcie makrodefinicji.

SKŁADNIA

```
PURGE nazwa_makra [, nazwa_makra]
```

KOMENTARZ

Często używane makroinstrukcje mogą być wspólnie umieszczone w pliku nagłówkowym. Jedną z niedogodności tego rozwiązania jest możliwość konfliktu nazw jednej z makroin-

strukcji i symbolu używanego w programie. Zmiana, dokonana w pliku nagłówkowym, może spowodować to, że przestanie on być poprawny dla innych programów. W tych wypadkach przydatna okazuje się dyrektywa **PURGE**. Usuwa ona nazwy podanych makroinstrukcji z tablicy symboli asemblera. Nazwy te mogą być więc wykorzystane do innych celów.

PATRZ

MACRO

PRZYKŁAD

```
INCLUDE MACROS.INC   ; Włączenie zbioru, który zawiera
                     ; makroinstrukcję Print
PURGE Print          ; Usunięcie nazwy Print z tablicy
                     ; symboli
Print   PROC         ; Poprawne zdefiniowanie nowego symbolu
                     ; o nazwie Print
        .
        .
        .
```

REPT

ZASTOSOWANIE

Powtórzenie bloku instrukcji.

SKŁADNIA

```
REPT wyrażenie
     instrukcje
ENDM
```

wyrażenie	Określa liczbę obiegów pętli.
instrukcje	Powtarzany blok instrukcji.

KOMENTARZ

Blok instrukcji zostanie powtórzony, określoną przez wartość wyrażenia, liczbę razy. Zastąpienie sekwencji, powtarzających się rozkazów, dyrektywą **REPT** sprawia, że program staje się bardziej zwięzły i łatwiejszy do czytania. W pętli mogą być używane zmienne i instrukcje kompilacji warunkowej.

PATRZ

IRP
IRPC
ENDM

REPT

PRZYKŁAD

```
; Definicja 10 bajtów, które zawierają liczby
; o wartościach od 1 do 10
Value       =     1            ; Wartość początkowa
            REPT  10           ; Dziesięć obiegów pętli
            DB    Value        ; Zdefiniowanie bajtu
                               ; i przypisanie mu wartości
                               ; Value
Value       =     Value + 1    ; Kolejna wartość
            ENDM               ; Koniec pętli
; Utworzenie 4 instrukcji shl
            REPT   4
            shl    ax, 1
            ENDM
```

Rozdział 11 Obsługa błędów i tworzenie raportów

Turbo Assembler może generować kod dla każdego z procesorów rodziny 80x86. Może również tworzyć kod dla odpowiadających im koprocesorów. Programista musi więc poinformować kompilator, jakiego zestawu instrukcji będzie używał. Do tego celu istnieją specjalne dyrektywy, opisane w pierwszej części rozdziału. Przedstawione tam zostały również dyrektywy, które służą do zapewnienia zgodności z asemblerem MASM firmy Microsoft.
Turbo Assembler może w trakcie kompilacji wykryć wiele błędów. Część z nich, przed podjęciem ponownej kompilacji, musi zostać poprawiona np. usunięcie nie zdefiniowanych symboli. Pozostałe, jak np. użycie zbyt długiej linii, generują tylko ostrzeżenie, a proces kompilacji może zostać dokończony. W części drugiej przedstawione zostały dyrektywy, które obsługują te błędy i ostrzeżenia.
Raporty kompilacji to pliki, w których umieszczany jest kod źródłowy oraz odpowiadający mu efekt działania asemblera. Przydają się one przede wszystkim w czasie uruchamiania programu. W ostatniej części zamieszczono opis dyrektyw, które pozwalają na sterowanie informacją umieszczaną w raportach.

Cechy asemblera

Turbo Assembler składa się z kilku asemblerów. Najczęściej używanym jest asembler zgodny z kompilatorem MASM dla procesorów 8088/8086. Używając odpowiednich dyrektyw można wybrać inny rodzaj asemblera. Turbo Assembler może bowiem pracować na wielu różnych procesorach, pozwala korzystać z dyrektyw kompilatora MASM oraz dostarcza swoje własne polecenia, może też generować instrukcje dla koprocesorów.

Procesory

Rodzina 80x86 firmy Intel składa się z siedmiu różnych mikroprocesorów, które mają pięć różnych zestawów instrukcji. Pierwsze z tych procesorów to 8088 i 8086. Posiadają one ten sam zestaw rozkazów. Standardowo asembler używa instrukcji tych właśnie układów. Jeśli program zawiera rozkaz innego procesora, to kompilator zasygnalizuje błąd. Oczywiście odpowiednią dyrektywą, umieszczaną na początku programu można wybrać inny zestaw instrukcji. Dyrektywy te zostały przedstawione w tabeli 11-1. Polecenia, które zaczynają się od kropki (.) mogą być używane tylko w trybie **MASM**. Natomiast te, które zaczynają się literą P, mogą wystąpić zarówno w trybie **MASM** jak i **IDEAL**.

Tabela 11-1. Dyrektywy wyboru zestawu instrukcji procesora.

Dyrektywa	Zestaw instrukcji
.186, P186	Dostępne wszystkie instrukcje procesorów 80186/80188.

Dyrektywa	Zestaw instrukcji
.286, .286C, P286N	Dostępne nieuprzywilejowane instrukcje procesora 80286.
.286P, P286	Dostępne wszystkie instrukcje procesora 80286.
.287, P287	Dostępne wszystkie instrukcje koprocesora 80287.
.386, .386C, P386N	Dostępne nieuprzywilejowane instrukcje procesora 80386.
.386P, P386	Dostępne wszystkie instrukcje procesora 80386.
.387, P387	Dostępne wszystkie instrukcje koprocesora 80387.
.8086, P8086	Dostępne wszystkie instrukcje procesorów 8086/8088.
.8087, P8087	Dostępne wszystkie instrukcje koprocesora 8087.

Zgodność z asemblerem MASM

Turbo Assembler jest w pełni zgodny z asemblerem MASM. Programy napisane dla Turbo Assemblera, mogą być swobodnie przeniesione na kompilator MASM. Ale istnieje również możliwość wykorzystania cech specyficznych tylko dla Turbo Assemblera. Wtedy jednak nie zostanie zachowana zgodność z kompilatorem MASM.
W asemblerze MASM istnieje kilka rzeczy, które w Turbo Assemblerze są już niedozwolone. Na przykład, MASM pozwala na stworzenie poleceń, które nie mają sensu:

```
ABC     EQU     [BP+2]
        PUBLIC  AB      ; ABC nie wskazuje żadnego miejsca
                        ; w pamięci
```

Natomiast Turbo Assembler, napotkawszy tego rodzaju kod, generuje ostrzeżenie. Asembler MASM posiada swój własny format liczb zmiennoprzecinkowych. Włącza się go dyrektywą **.MSFLOAT**. Turbo Assembler natomiast zawsze używa standardu IEEE zapisu liczb rzeczywistych. Używają go również koprocesory. Dyrektywa **.MSFLOAT** nie jest przez niego akceptowana.

TRYB MASM

Standardowym trybem pracy Turbo Assemblera jest tryb MASM. Oznacza to, że można skompilować pliki przeznaczone dla asemblera MASM. Należy jednak pamiętać, że w tym trybie dostępnych jest kilka dyrektyw, charakterystycznych tylko dla Turbo Assemblera. Dlatego też gdy Turbo Assembler potrafi skompilować dowolny plik napisany dla MASM, asembler MASM może nie potrafić przetłumaczyć niektórych plików, powstałych w Turbo Assemblerze.
Turbo Assembler jest zgodny z wersjami kompilatora MASM od 1 do 4. By wykorzystać cechy charakterystyczne dla wersji MASM 5.1, należy użyć najpierw dyrektywy **MASM51**. Stają się wtedy dostępne dyrektywy **SUBSTR**, **CATSTR**, **SIZESTR** oraz **INSTR**, a także znak backslash (\), który oznacza, że następna linia jest kontynuacją bieżącej.
Zmienia się również składnia dyrektywy **LOCAL**. Może być ona teraz następująca:

```
LOCAL nazwa [[ilość] [:[zakres] PTR] typ]
```

W trybie **QUIRKS**, który pozwalającym na wykorzystanie kilku sztuczek asemblera MASM:

- Etykiety lokalne mogą mieć przedrostek @F lub @B, gdy ustawiono przedrostek @@
- w procedurach zmienne mogą być redefiniowane
- w języku C wszystkie procedury są publiczne, a ich nazwy poprzedzone są podkreśleniem

W trakcie użytkowania w sposobie kompilacji programów przez asembler MASM znaleziono kilka niejednoznaczności. W wielu wypadkach programiści wykorzystali je w swoich programach. Wydając dyrektywę **QUIRKS** można je również wykorzystać w programach pisanych w Turbo Assemblerze. Efekty jej działania są następujące:

- Przy przesyłaniu do pamięci rejestrów segmentowych nie jest sprawdzana poprawność tej operacji. Kod:

  ```
  SEGVAL    DB    ?
        mov    SEGVAL, ds
  ```

  jest więc poprawny. W tym wypadku bajt za zmienną **SEGVAL** zostanie zamazany.

- Ignorowanie operatora **FAR PTR** w instrukcjach **JMP**, jeśli etykieta znajduje się w tym samym segmencie co ten rozkaz. (W Turbo Assemblerze 2.0 cecha ta była w trybie standardowym.) Kod:

  ```
          jmp    FAR PTR Next
      Next:
  ```

  był kompilowany jako:

  ```
  9Bh                ; 8-bitowy skok
  03h                ; Przemieszczenie do etykiety Next
  90h, 90h, 90h      ; Instrukcje NOP, które wypełniają
                     ; miejsce przeznaczone na skok daleki
  ```

- Gubienie informacji o typie w instrukcjach EQU lub =. Kod:

  ```
      X    DW    0
      Y    =     OFFSET X
      mov ax, Y
  ```

  będzie skompilowany jako MOV ax, [X], zamiast MOV ax, OFFSET X.

- Dyrektywa **ALIGN** może rozmieszczać dane na mniejszych granicach niż rozmieszczony jest zawierający ją segment. Może być ona ignorowana, gdyż konsolidator może umieszczać segmenty na różnych granicach.

- Rozkazy z rozszerzaniem bajtu z zachowaniem znaku są stosowane tylko w instrukcjach arytmetycznych. Wszystkie rozkazy, których argumentem jest dana natychmiastowa, mogą przesłać wartość jednobajtową do rejestru 16-bitowego. Bajt ten zostanie rozszerzony z zachowaniem znaku. W trybie **QUIRKS** rozszerzanie to dotyczy tylko instrukcji arytmetycznych.

TRYB IDEAL

Turbo Assembler posiada swój własny zbiór dyrektyw i trochę inną składnię niż w trybie MASM. Cechy te ujawniają się w trybie **IDEAL**, do którego przechodzi się poleceniem IDEAL. Z powrotem na tryb MASM można przełączyć się dyrektywą **MASM**.
W książce przy opisach poszczególnych dyrektyw podano ich składnię dla obu trybów. Ponadto tryb IDEAL cechuje się:

- Szybszą kompilacją o 30%
- Te same nazwy pól struktur mogą występować w kilku strukturach
- Operatory **HIGH** i **LOW** mogą pobierać części danych o różnych rozmiarach
- Stałe zadeklarowane dyrektywą **EQU** są traktowane jako łańcuchy
- Przemieszczenie danej jest zawsze wyznaczane względem początku grupy, w której się ona znajduje
- Nawiasy kwadratowe muszą otaczać całe wyrażenie adresowe
- Operator SIZE zwraca rozmiar wszystkich elementów, które wchodzą w skład symbolu
- Dyrektywy nie mogą zaczynać się znakiem kropki (.).

Implikacje, które wynikają z użycia trybu IDEAL i dotyczą poszczególnych dyrektyw są omówione w ich opisach znajdujących się za podsumowaniem, na końcu tego rozdziału.

Pseudoinstrukcje

Turbo Assembler pozwala używać instrukcji, które nie są częścią zbioru rozkazów procesora. Istnieją trzy typy takich pseudoinstrukcji. Pierwszy to emulacja rozkazów koprocesora. Każda instrukcja zmiennoprzecinkowa, która występuje za dyrektywą **EMUL**, ale przed **NOEMUL**, wygeneruje emulujący ją kod, zamiast fizycznego rozkazu dla koprocesora. Dyrektywa **EMUL** odpowiada opcji **/e** kompilatora. Ponadto do konsolidowania muszą zostać dołączone podprogramy emulacji.
Drugi typ tworzą pseudoinstrukcje, które dotyczą skoków warunkowych. Po użyciu dyrektywy **JUMPS**, można stosować skoki warunkowe o zakresie większym niż -128 do 127 bajtów od miejsca skoku. Ponieważ dla procesorów 8088/86 nie ma pojedynczego rozkazu realizującego taki skok, to asembler wygeneruje dwie instrukcje skoków. Pierwszym z nich jest rozkaz skoku warunkowego, który testuje przeciwny niż wyjściowy warunek, przenoszący sterowanie za następną instrukcję. A tą instrukcją jest właśnie drugi rozkaz skoku, tym razem bezwarunkowy, bliski, przekazujący sterowanie do żądanego miejsca programu. Na przykład:

```
jc    nlabel     ; Skocz, jeśli znacznik przeniesienia
                 ; ustawiony (nlabel jest oddalona
                 ; o więcej niż 128 bajtów od
                 ; miejsca skoku)
```

staje się:

```
jnc   @@a        ; Ominięcie następnej instrukcji,
```

```
                        ; gdy znacznik przeniesienia nie
                        ; jest ustawiony
     jmp    nlabel      ; Jeśli znacznik przeniesienia jest
                        ; ustawiony, wykonaj bliski skok
@@a:
```

Ostatni typ pseudoinstrukcji stanowią instrukcje, które przesyłają daną natychmiastową na stos. Zwykle dana musi być najpierw przesłana do rejestru, a dopiero potem na stos. Natomiast Turbo Assembler pozwala na bezpośrednie przesłanie stałej na stos, generując w tym celu odpowiedni kod. Na przykład, pseudoinstrukcja:

```
push  21h                ; Prześlij liczbę 21h na stos
```

powoduje wygenerowanie kodu:

```
push  ax                 ; Zarezerwowanie miejsca na stosie
push  bp                 ; Zachowanie rejestru BP
mov   bp, sp             ; Pobranie wskaźnika stosu
mov   [bp+2], 21h        ; Przesłanie danej na stos w miejsce
                         ; zajmowane przez rejestr AX
pop   bp                 ; Odtworzenie rejestru BP
```

Należy zauważyć, że powstało o wiele więcej kodu, niż w wypadku przesłania danej do rejestru AX i następnie odłożenia tego rejestru na stos. Wygenerowanie dodatkowego kodu wynika z tego, że wartość żadnego z rejestrów nie może zostać zmieniona.

Śledzenie przebiegu kompilacji

Pojedynczy plik źródłowy może wygenerować wiele wersji programu, w zależności od wartości symboli, używanych w instrukcjach kompilacji warunkowej (Patrz rozdział 10). Utworzenie nowej wersji może jednak okazać się problemem, jeśli dokonane zmiany nie są tymi oczekiwanymi. Jednym ze sposobów sprawdzenia, czy wszystko przebiega zgodnie z założeniami, jest wyświetlanie komunikatów o aktualnie kompilowanych blokach instrukcji. Turbo Assembler posiada kilka dyrektyw przeznaczonych do tego celu.

Wyświetlanie komunikatów

Dyrektywy **%OUT** oraz **DISPLAY** wyświetlają na ekranie komunikaty w czasie kompilacji. W zasadzie są one identyczne, z tym że łańcuch w dyrektywie **DISPLAY** musi być ujęty w cudzysłów. Mogą one zostać wykorzystane do określenia, który blok instrukcji został skierowany do kompilacji:

```
INCLUDE   config.inc          ; Dołączenie symboli
                              ; konfigurujących program
IF @CodeSize                  ; Testowanie symbolu
                              ; predefiniowanego
%OUT Model small
ELSE
DISPLAY "Korzystamy z modelu pamięci large"
ENDIF
```

Błędy i ostrzeżenia

Turbo Assembler informuje o błędach oraz niejednoznacznych sformułowaniach, wyświetlając w czasie kompilacji odpowiednie komunikaty. Jeśli w programie występują błędy, wówczas plik wynikowy nie zostanie utworzony.
Do błędów już wykrywanych przez kompilator można dołączyć swoje własne. Na przykład, jeśli chcemy zabezpieczyć się przed przekazaniem makroinstrukcji, to pustego argumentu możemy użyć kodu:

```
MyMacro   MACRO arg1, arg2
.ERRB arg1                    ; Wygeneruj błąd. gdy arg1
                              ; jest pusty
Lab&arg1  DB   arg2           ; Stwórz symbol, którego
                              ; nazwa zawiera arg1
          ENDM
```

Ponadto do wyświetlenia informacji o zaistniałym błędzie można użyć dyrektyw **%OUT** lub **DISPLAY**. Poprzedni przykład można wtedy zapisać:

```
MyMacro   MACRO arg1, arg2
IFB arg1
.ERR
%OUT Brak argumentów dla MyMacro
ENDIF
Lab&arg1  DB   arg2
          ENDM
```

Istnieje kilka dyrektyw, które generują błąd w zależności od spełnienia różnych warunków. Przedstawione są one w tabeli 11-2. Dyrektywy, zaczynające się od kropki (.), mogą być używane tylko w trybie MASM. Pozostałe akceptowane są w obu trybach.

Tabela 11-2. Dyrektywy generujące błędy.

Dyrektywa	Warunek
.ERR, ERR	Bezwarunkowe wygenerowanie błędu.
ERRIF wyr	Błąd jest generowany, jeśli wyrażenie wyr jest prawdziwe.
.ERR1, ERRIF1	Błąd jest generowany w pierwszym przebiegu asemblera.
.ERR2, ERRIF2	Błąd jest generowany w drugim przebiegu asemblera.
.ERRB arg, ERRB arg	Błąd jest generowany, jeśli argument arg jest pusty.
.ERRDEF symbol, ERRIFDEF symbol	Błąd jest generowany, jeśli podany symbol jest zdefiniowany.
.ERRDIF arg1, arg2, ERRIFDIF arg1, arg2	Błąd jest generowany, jeśli arg1 różni się od arg2.
.ERRDIFI arg1, arg2, ERRIFDIFI arg1, arg2	Podobnie do .ERRDIF, z tym że ignorowana jest wielkość liter.
.ERRE wyr, ERRIFE wyr	Błąd jest generowany, jeśli wynik wyrażenia wyr jest równy 0.

Dyrektywa	Warunek
.ERRIDN arg1, arg2, ERRIFIDN arg1, arg2	Błąd jest generowany, jeśli arg1 jest identyczne z arg2.
.ERRIDNI arg1, arg2, ERRIFIDNI arg1, arg2	Podobnie do .ERRIDN, z tym że ignorowana jest wielkość liter.
.ERRNB arg, ERRIFNB arg	Błąd jest generowany, jeśli argument arg nie jest pusty.
.ERRNDEF symbol, ERRIFNDEF symbol	Błąd jest generowany, jeśli podany symbol nie jest zdefiniowany.
.ERRNZ wyr, ERRIFNZ wyr	Błąd jest generowany, jeśli wynik wyrażenia wyr jest różny od 0.

Komentarze

We wszystkich dotychczas przedstawionych przykładach pojawiały się komentarze. Tekst jest traktowany jako komentarz, gdy znajduje się za średnikiem (;). Natomiast dyrektywą **COMMENT** można utworzyć komentarz, który składałby się z kilku linii. Jej argumentem jest znak, oznaczający koniec komentarza. Asembler zignoruje wtedy cały tekst, aż do napotkania podanego znaku. Zastosowanie tej dyrektywy zostało przedstawione w przykładzie:

```
COMMENT ^ To jest komentarz.
Ten cały tekst zostanie przez asembler zignorowany. Kompila-
cja rozpocznie się po odnalezieniu drugiego znaku (^).
^ Koniec komentarza. Linia ta będzie również zignorowana.
```

Raporty

Poza tworzeniem plików wynikowych, asembler może jeszcze generować raporty kompilacji, które zawierają oprócz instrukcji źródłowych, informacje o sposobie zinterpretowania ich przez kompilator. Aby Turbo Assembler utworzył tego rodzaju plik, należy w poleceniu wywołującym go, podać nazwę pliku, w którym raport zostanie umieszczony:

```
c:\> tasm prnstr,,prnstr
```

Pierwsza nazwa określa kompilowany plik. Przecinki rozdzielają nazwy różnych plików, które mogą przez asembler zostać utworzone. W tym wypadku nie podano nazwy pliku, który zawiera mapę kompilacji, nie zostanie on więc wygenerowany. Ostatnia nazwa określa plik, zawierający raport. Ponieważ nie podano jego rozszerzenia, to przyjęte ono zostanie jako .LST. Zastosowanie powyższej komendy do pliku:

```
%TABSIZE    4
PAGE        50, 65
%BIN        10
%LINUM      0
%PCNT       3
%TEXT       45
TITLE       Napisz Hello!
```

```
; Definicja stałych
LF    EQU  0Ah
CR    EQU  0Dh
; Makroinstrukcja, która wywołuje funkcję
; systemu operacyjnego DOS
CallOS     MACRO    cmnd
           mov      ah, cmnd
           int      21h
           ENDM
           .MODEL   SMALL
           .DATA
Msg        DB        "Hello, World", LF, CR, '$'
           PAGE      ; Wymuszenie końca strony
           .CODE
Start:
           mov       ax, SEG DGROUP ; Ustal adres segmentu
                                    ; danych
           mov       ds, ax
           mov       dx, OFFSET Msg ; Pobierz adres
                                    ; komunikatu
           CallOS    <9>            ; Funkcja, która
                                    ; wyświetla łańcuch
           CallOS    <4Ch>          ; Funkcja, która
                                    ; kończy program
           .STACK
END        Start
```

...powoduje wygenerowanie następującego raportu kompilacji:

```
Turbo Assembler   Version 3.2   04/20/94 12:16:32
Page 1
PRNSTR.ASM
Print Hello string
; Definicja stałych
       =000A   LF   EQU   0Ah
       =000D   CR   EQU   0Dh
; Makroinstrukcja, która wywołuje funkcję
; systemu operacyjnego DOS
       CallOS  MACRO   cmnd
       mov     ah, cmnd
       int     21h
       ENDM
 *00            .MODEL SMALL
 *00            .DATA
 *00  48 65 6C  + Msg  DB  "Hello,  World",+
      6C 6F 2C  + LF, CR, '$'
      20 57 6F  +
```

```
      72 6C 64  +
      0A 0D 24
Turbo Assembler  Version 3.2  04/20/94 12:16:32
Page 2
PRNSTR.ASM
Print Hello string
 *0F                .CODE
 *00        Start:
 *00    B8 0000s  mov  ax, SEG DGROUP
                  ; Ustal adres segmentu danych
 *03    8E D8     mov  ds, ax
 *05    BA 0000r  mov  dx, OFFSET Msg
; Pobierz adres komunikatu
                  CallOS <9>   +
; Funkcja, która wyświetla łańcuch
1 *08   B4 09     mov  ah, 9
1 *0A   CD 21     int  21h
                  CallOS <4Ch> +
; Funkcja, która kończy program
1 *0C   B4 4C     mov  ah, 4Ch
1 *0E   CD 21     int  21h
 *00               .STACK
            END   Start
Turbo Assembler  Version 3.2  04/20/94 12:16:32
Page 3
Symbol Table
Print Hello string
Symbol Name       Type  Value
??DATE            Text   "04/20/94"
??FILENAME        Text   "prnstr  "
??TIME            Text   "12:16:32"
??VERSION         Number 0314
@32BIT            Text   0
@CODE             Text   _TEXT
@CODESIZE         Text   0
@CPU              Text   0101H
@CURSEG           Text   _TEXT
@DATA             Text   DGROUP
@DATASIZE         Text   0
@FILENAME         Text   PRNSTR
@INTERFACE        Text   00H
@MODEL            Text   2
@STACK            Text   DGROUP
@WORDSIZE         Text   2
CR                Number 000D
LF                Number 000A
```

```
MSG                 Byte    DGROUP:0000
START               Near    _TEXT:0000
Macro Name
CALLOS
Groups & Segments   Bit   Size Align  Combine Class
DGROUP              Group
 STACK              16    0400 Para Stack  STACK
 _DATA              16    000F Word Public DATA
_TEXT               16    0010 Word Public CODE
```

W raporcie kompilacji umieszczanych jest wiele informacji, które dotyczą właśnie skompilowanego zbioru. Każda jego strona zawiera linię, która określa, że jest to plik stworzony przez Turbo Assembler oraz podaje datę i czas kompilacji. Następna linia określa numer strony. Dalej występuje nazwa pliku źródłowego lub nazwa sekcji oraz tytuł dla danego pliku.

Każda linia pliku źródłowego pojawia się w raporcie kompilacji. Na jej początku dodawana jest odpowiadająca jej wartość wskaźnika pozycji oraz kod przez nią wygenerowany. Komendy, które nie powodują utworzenia kodu, zawierają tylko wartość wskaźnika. Dyrektywy EQU lub = pokazują wartość definiowanego symbolu poprzedzoną znakiem równości (=). Przy rozwijaniu makroinstrukcji przed wartością wskaźnika pozycji umieszczony zostaje dodatkowo numer poziomu zagnieżdżenia.

Położenie grupy DGROUP nie może zostać określone, aż do momentu uruchomienia programu. Zatem asembler miejsce na jej adres w pliku wynikowym wypełnia zerami oraz umieszcza w nim rekord, który zawiera informację dla konsolidatora, że miejsce to należy w procesie ładowania programu uzupełnić poprawną wartością. Takie miejsca w raporcie oznaczane są literą s. W trakcie kompilacji nie może być również wyznaczony adres zmiennej Msg. W tym wypadku natomiast jego poprawna wartość może być już wyznaczona przez konsolidator. Doda on bowiem przemieszczenie zmiennej Msg w module do następnego wolnego przemieszczenia w segmencie danych. Asembler umieszcza więc w pliku wynikowym przemieszczenie zmiennej w module, jednocześnie informując program łączący, że wymaga ono uaktualnienia. Takie miejsca w raporcie są zaznaczane literą r.

Ostatnia część raportu zawiera tablicę symboli programu. Zawiera ona nazwę, typ i wartość każdego zdefiniowanego symbolu. Lista ta jest uporządkowana w kolejności alfabetycznej. Można zauważyć, że wiele symboli jest przez asembler tworzonych automatycznie. Najpierw występują symbole definiujące dane, następnie symbole, które oznaczają makroinstrukcje, a na końcu nazwy segmentów i grup. Z każdą nazwą grupy występuje lista segmentów, które wchodzą w jej skład. Lista ta zawiera nazwę segmentu, jego typ (16- czy 32-bitowy), rozmiar w bajtach, typ rozmieszczenia, sposób połączenia z innymi segmentami oraz jego klasę.

Formatowanie raportu

Turbo Assembler formatuje raport kompilacji tak, by był on jasny i łatwy w czytaniu. Jednak w celu osiągnięcia lepszej przejrzystości tak tworzonego pliku, można określić kilka jego cech. W poprzednim przykładzie pierwszą dyrektywą była **%TABSIZE**. Określa ona liczbę kolumn, zajmowanych przez pojedynczy znak tabulacji. Wartością standardową jest 8. Jeśli

ustawiony w pliku rozmiar znaku tabulacji nie odpowiada rozmiarowi znaku tabulacji na urządzeniu wyjściowym, to powstały wydruk może okazać się nieczytelny.
Drugą dyrektywą jest **PAGE**. Określa ona rozmiar strony raportu. Pierwszy argument oznacza liczbę wierszy, a drugi liczbę kolumn na stronie. Jeśli linia jest dłuższa od określonego rozmiaru kolumny, to zostanie ona obcięta albo przeniesiona do następnego wiersza, zależnie od ustawienia znacznika **%TRUNC**. Dyrektywa **PAGE** bez argumentów powoduje umieszczanie następnych linii raportu na nowej stronie.
Następnych pięć dyrektyw używanych jest do określania rozmiarów różnych pól raportu. Są one użyteczne przy ustawianiu rozmiaru linii tak by mieściła się ona na stronie. Przedstawione są one w tabeli 11-3.

Tabela 11-3. Dyrektywy określające rozmiar pól w raporcie.

Dyrektywa	Wartość standardowa	Pole
%BIN	20	Pole kodu wynikowego
%DEPTH	1	Poziom zagnieżdżenia makroinstrukcji
%LINUM	4	Numer linii
%PCNT	4 lub 8	Wskaźnik pozycji
%TEXT	NA	Kod źródłowy

Ustalanie zawartości raportu

Raport kompilacji może stać się nawet bardzo duży. Im jest on większy, tym trudniej odnaleźć interesującą nas część. Może on zostać skrócony poprzez wyłączenie generowania raportu dla pewnych części kodu źródłowego. Przełączenia dokonują dyrektywy **.LIST** i **.XLIST**. Pozwalają one umieścić w raporcie kompilacji tylko żądane fragmenty programu. Dyrektywy te są najczęściej stosowane w plikach nagłówkowych, zapobiegając umieszczaniu ich zawartości w raporcie.
Asembler może również umieszczać w raporcie tylko informacje o określonych rodzajach danych. Na przykład, dyrektywa **.LALL** powoduje umieszczanie w raporcie rozwiniętych wersji makroinstrukcji, a dyrektywa **.SALL** zapobiega ich rozwijaniu. Polecenia, znajdujące się nie kompilowanych gałęziach instrukcji warunkowych, mogą zostać usunięte z listingu dyrektywą **.LFCOND**. Dyrektywa **%INCL** powoduje umieszczenie w raporcie zawartości plików nagłówkowych. Przeciwne działanie ma dyrektywa **%NOINCL**.

Lista symboli

Tablica symboli, umieszczana na końcu raportu kompilacji, może okazać się przydatna w procesie uruchamiania i testowania programu. Stanowi ona bowiem mapę danych, używanych w programie. Dyrektywą **.CREF** można wygenerować ponadto listę odwołań do symboli programu.
Tablica symboli nie zostanie jednak umieszczona w pliku z raportem, gdy w zbiorze źródłowym pojawi się dyrektywa **%NOSYMS**. Może ona wystąpić w dowolnym jego miejscu. Dyrektywą o odwrotnym działaniu jest **%SYMS**. Przyjmowana jest ona standardowo.

Lista odwołań zawiera wszystkie symbole programu wraz z numerami linii, w których one wystąpiły. Jeśli dyrektywa **.CREF** wystąpiła w pliku źródłowym, to informacje te będą włączone do tablicy symboli. Dla poprzedniego przykładu lista odwołań wygląda następująco:

```
Symbol Name        Type    Value        Cref (defined at #)
??DATE             Text    "04/20/94"
??FILENAME         Text    "prnstr  "
??TIME             Text    "14:46:27"
??VERSION          Number  0314
@32BIT             Text    0            #13
@CODE              Text    _TEXT        #13       #13     #21
@CODESIZE          Text    0            #13
@CPU               Text    0101H
@CURSEG            Text    _TEXT        #14       #21     #39
@DATA              Text    DGROUP       #13
@DATASIZE          Text    0            #13
@FILENAME          Text    PRNSTR
@INTERFACE         Text    00H          #13
@MODEL             Text    2            #13
@STACK             Text    DGROUP       #13
@WORDSIZE          Text    2            #14       #21     #39
CR                 Number  000D         #4        16
LF                 Number  000A         #3        16
MSG                Byte    DGROUP:0000  #16       27
START              Near    _TEXT:0000   #22       40
Macro Name                              Cref   (defined at #)
CALLOS                                  #8         30      34
Groups & Segments Bit Size Align Combine Class  Cref (defi-
ned at #)
DGROUP     Group                       #13       13      23
STACK      16 0400 Para  Stack  STACK  #39
_DATA      16 000F Word  Public DATA   #13       #14
_TEXT      16 0010 Word  Public CODE   #13       3     #21 21
```

Numery linii, poprzedzone znakiem #, oznaczają linie, w których wystąpiła definicja danego symbolu. Na przykład, symbol **_DATA** jest definiowany w liniach 13 i 14. W pozostałych liniach nastąpiło odwołanie do symbolu. Do symboli, przy których nie występują takie linie, nie wystąpiło żadne odwołanie w tekście programu. Tablicę symboli można ograniczyć tylko do używanch symboli dyrektywą **%CREFREF**. Natomiast po wydaniu polecenia **%CREFUREF** w tablicy symboli znajdą się tylko te symbole, do których nie nastąpiło żadne odwołanie.

Zapamiętywanie i odtwarzanie ustawień formatu raportu

Można zapamiętać stan zmiennych sterujących formatem raportu takich jak np. **%INCL**, **%NOINCL**, **%CTRL**, **%NOCTRL**, by na pewien czas je zmienić. Na przykład, mamy plik nagłówkowy, którego zawartość nie będzie umieszczana w raporcie. Nie wiemy jednak,

czy pozostała część programu ma być listowana czy też nie. W takiej sytuacji można zastosować poniższą sekwencję poleceń:

```
%PUSHLCTL       ; Zachowaj stan zmiennych sterujących
                ; generowaniem raportu
%NOLIST         ; Wyłącz generowanie raportu
        .
        .
        .
%POPLCTL        ; Odtwórz zapamiętane ustawienia
```

Dyrektywy **%PUSHLCTL** i **%POPLCTL** zapisują i pobierają dane z 16-poziomowego stosu.

Podsumowanie

Turbo Assembler posiada wiele różnych możliwości. Może on kompilować instrukcje dla wielu procesorów rodziny 80x86 i odpowiadających im koprocesorów. Wyboru odpowiedniego zestawu instrukcji można dokonać dyrektywami **.286** i **.8087**. Poza tym Turbo Assembler potrafi kompilować programy napisane w różnych wersjach asemblera MASM, jak również posiada swój własny tryb IDEAL.

Śledzenie procesu kompilacji ułatwia tworzenie różnych wersji programu. Dyrektywy **%OUT** oraz **DISPLAY** służą do wyświetlania różnych komunikatów w czasie procesu kompilacji. Dzięki dyrektywom typu **.ERR** można stworzyć własny sposób obsługi niektórych błędów.

Raporty kompilacji to pliki, w których asembler umieszcza wyniki swojego działania. Ich analiza jest pierwszym krokiem przy szukaniu błędów w programie. Dyrektywy takie jak **PAGE** i **%TABSIZE**, pozwalają sterować formatem tego pliku. Inne jak **.LIST** i **.LALL** pozwalają umieszczać w raporcie tylko określone fragmenty programu.

.186

ZASTOSOWANIE

Udostępnienie instrukcji specyficznych dla procesora 80186.

SKŁADNIA

```
.186
```

KOMENTARZ

Wszystkie instrukcje za tą dyrektywą muszą należeć do zbioru rozkazów procesora 80186. Asembler tylko sprawdza, czy instrukcje należą do poprawnego zestawu, natomiast programista musi sam zadbać, by program był uruchomiony na odpowiednim procesorze i koprocesorze.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

Dla procesora 80186 i nowszych istnieje już instrukcja, która przesyła daną natychmiastową na stos:

```
.186          ; Udostępnienie instrukcji
              ; procesora 80186
push  12      ; Przesłanie liczby 12 na stos
```

.286, .286C

ZASTOSOWANIE

Udostępnienie instrukcji trybu rzeczywistego procesora 80286.

SKŁADNIA

```
.286
```

```
.286C
```

KOMENTARZ

Wszystkie instrukcje za tą dyrektywą muszą należeć do zbioru rozkazów procesora 80286 w trybie rzeczywistym. Asembler tylko sprawdza, czy instrukcje w zbiorze źródłowym należą do poprawnego zestawu, natomiast programista musi sam zadbać, by program był uruchomiony na odpowiednim procesorze i koprocesorze.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

```
            .286                          ; Będzie użyta
                                          ; instrukcja BOUND
ArrayBnds   DW    0, 99
Array       DB    100 DUP(?)
            .
            .
            .
            bound   si, DWORD PTR ArrayBnds ; Czy SI
                                            ; znajduje się
                                            ; w podanym
                                            ; zakresie
            mov     al, Array[si]           ; Tak, można
                                            ; odwołanie
                                            ; jest
                                            ; prawidłowe
```

.286P

ZASTOSOWANIE

Udostępnienie instrukcji trybu chronionego procesora 80286.

.287

SKŁADNIA

`.286P` *Udostępnienie wszystkich instrukcji procesora 80286*

KOMENTARZ

Instrukcje za tą dyrektywą mogą być rozkazami procesora 80286 w trybie chronionym. Oczywiście dyrektywa ta nie przełącza procesora w ten tryb, również nie sprawdza, czy w ogóle jest to procesor 80286. Zapewnić to powinien programista.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

```
.286P          ; Udostępnienie instrukcji trybu
               ; chronionego
lgdt  ax       ; Instrukcja w trybie chronionym,
               ; która ładuje
               ; rejestr globalnej tablicy
               ; deskryptorów
```

.287

ZASTOSOWANIE

Udostępnienie rozkazów koprocesora 80287.

SKŁADNIA

`.287` *Udostępnienie rozkazów koprocesora 80287*

KOMENTARZ

Instrukcje za tą dyrektywą mogą zawierać rozkazy dla koprocesora 80287. Jeśli została użyta dyrektywa **EMUL** lub opcja /e asemblera, to instrukcje te zostaną zastąpione wywołaniami procedur z biblioteki podprogramów matematycznych. W przeciwnym razie zostanie wygenerowana odpowiednia instrukcja **ESC**.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

```
.287            ; Udostępnienie instrukcji
                ; zmiennoprzecinkowych
fsetpm          ; Instrukcja koprocesora 80287
```

.386, .386C

ZASTOSOWANIE

Udostępnienie instrukcji trybu rzeczywistego procesora 80386.

SKŁADNIA

```
.386
.386C
```

KOMENTARZ

Wszystkie instrukcje za tą dyrektywą muszą należeć do zbioru rozkazów procesora 80386 w trybie rzeczywistym. Asembler tylko sprawdza, czy instrukcje w zbiorze źródłowym należą do poprawnego zestawu, natomiast programista musi już zadbać, by program był uruchamiany na odpowiednim procesorze i koprocesorze.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087

.386P

PN087

PRZYKŁAD

```
; By móc wykorzystać rejestry 32-bitowe, należy udostępnić
instrukcje procesora 80386
      .386            ; Udostępnienie instrukcji
                      ; procesora 80386
      add   eax, 14   ; Można użyć rejestr 32-bitowego
```

.386P

ZASTOSOWANIE

Udostępnienie instrukcji trybu chronionego procesora 80386.

SKŁADNIA

```
.386P
```

KOMENTARZ

Instrukcje za tą dyrektywą mogą być rozkazami procesora 80386 w trybie chronionym. Oczywiście dyrektywa ta nie przełącza procesora w ten tryb, również nie sprawdza, czy w ogóle jest to procesor 80386. Zapewnić to powinien programista.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

```
      .386P                 ; Udostępnienie instrukcji
                            ; trybu chronionego
      mov   ax, TSS_Sel     ; Skopiowanie selektora
                            ; zadania do rejestru AX
      ltr   ax              ; i przesłanie go do rejestru
                            ; zadania (instrukcja trybu
                            ; chronionego)
```

.387

ZASTOSOWANIE

Udostępnienie rozkazów koprocesora 80387.

SKŁADNIA

```
.387
```

KOMENTARZ

Instrukcje za tą dyrektywą mogą zawierać rozkazy dla koprocesora 80387. Jeśli została użyta dyrektywa **EMUL** lub opcja **/e** asemblera, to instrukcje te zostaną zastąpione wywołaniami procedur z biblioteki podprogramów matematycznych. W przeciwnym razie zostanie wygenerowana odpowiednia instrukcja **ESC**.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

```
.387      ; Udostępnienie instrukcji
          ; zmiennoprzecinkowych
fsin      ; Instrukcja koprocesora 80387
```

.8086

ZASTOSOWANIE

Udostępnienie instrukcji tylko procesorów 8088/86 i 8087.

SKŁADNIA

`.8086` *Udostępnienie instrukcji procesorów 8088/8086*

.8087

KOMENTARZ

Wszystkie instrukcje za tą dyrektywą muszą należeć do zbioru rozkazów procesorów 8088/86. Przyjmowane są również rozkazy dla koprocesora 8087. Dyrektywa ta jest przyjmowana standardowo.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

Zwykle nie używa się tej dyrektywy, gdyż jest ona przyjmowana przez asembler domyślnie. Czasami jednak tylko krótka sekcja programu powinna zawierać instrukcje specyficzne dla innych procesorów. Można wtedy, używając tej dyrektywy, przywrócić żądany tryb kompilacji.

```
.386                ; Udostępnienie instrukcji
                    ; procesora 80386
movsx   eax, bx     ; Skopiowanie rejestru BX
                    ; z zachowaniem znaku
.
.
.
.8086                ; Ponownie tryb, w którym przyjmowane
                     ; są tylko instrukcje procesora 8086
```

.8087

ZASTOSOWANIE

Udostępnienie rozkazów koprocesora 8087.

SKŁADNIA

`.8087` *Udostępnienie rozkazów koprocesora 8087.*

KOMENTARZ

Instrukcje za tą dyrektywą mogą zawierać rozkazy dla koprocesora 8087. Jeśli została użyta dyrektywa **EMUL** lub opcja **/e** asemblera, to instrukcje te zostaną zastąpione wywołaniami procedur z biblioteki podprogramów matematycznych. W przeciwnym razie zostanie wygenerowana odpowiednia instrukcja **ESC**.

PATRZ

P186
P286
P286N
P287
P386
P386N
P387
P8086
P8087
PN087

PRZYKŁAD

Zwykle nie używa się tej dyrektywy, gdyż standardowo asembler przyjmuje rozkazy dla koprocesora 8087. Czasami jednak pewne krótkie sekcje programu powinny zawierać instrukcje, specyficzne dla innych koprocesorów. Można wtedy, używając tej dyrektywy, przywrócić żądany tryb kompilacji.

```
.387        ; Udostępnienie instrukcji
            ; zmiennoprzecinkowych
fldpi       ; Instrukcja koprocesora 80387
.8087       ; Ponowne przełączenie na używanie tylko
            ; rozkazów koprocesora 8087
```

%BIN

ZASTOSOWANIE

Ustalenie w raporcie kompilacji szerokości pola, które zawiera kod wynikowy.

SKŁADNIA

`%BIN szerokość`

`szerokość` *określa liczbę znaków pola kodu wynikowego*

KOMENTARZ

Standardową wartością szerokości pola kodu wynikowego jest 20 znaków. Jeśli raport będzie wyświetlany na ekranie, to pole to zajmie już jego czwartą część. Na inne pola nie

pozostanie więc już wiele miejsca. Dyrektywą **%BIN** można zmniejszyć ilość miejsca, które jest przeznaczone na to pole.

PATRZ

%DEPTH
%LINUM
%PCNT
%TEXT

PRZYKŁAD

```
%PAGESIZE  55, 75      ; Utworzenie strony o szerokości
                       ; 75 kolumn, na której mieści się
                       ; 55 wierszy
%BIN   10              ; Zmniejszenie rozmiaru pola kodu
                       ; wynikowego do 10 znaków
%DEPTH  2              ; Szerokość pola poziomu
                       ; zagnieżdżenia wynosi 2 znaki
%LINUM  3              ; Szerokość pola numeru wiersza
                       ; wynosi 3 znaki
%PCNT   3              ; Szerokość pola wskaźnika pozycji
                       ; wynosi 3 znaki
%TEXT   50             ; Tekst źródłowy jest wyświetlany
                       ; w 50 kolumnach
.
.
.
```

%CONDS

ZASTOSOWANIE

Polecenie umieszczania w raporcie kompilacji linii, które znajdują się w nie kompilowanych gałęziach dyrektyw warunkowych.

SKŁADNIA

```
%CONDS
```

KOMENTARZ

Turbo Assembler umieszcza w raporcie kompilacji linie, które znajdują się w nie kompilowanych gałęziach dyrektyw warunkowych. Wydanie dyrektywy **%NOCONDS** blokuje tą możliwość. Przywrócona może ona zostać poleceniem **%CONDS**.

PATRZ

%MAC
%NOMAC
.LALL
.SALL
.XALL

PRZYKŁAD

```
%NOCONDS                ; Wyłączenie listowania fałszywych
                        ; gałęzi
IFDEF SYM1              ; Linie te w raporcie nie zostaną
                        ; umieszczone, gdy SYM1 nie jest
                        ; zdefiniowany
      .
      .
ENDIF
%CONDS                  ; Włączenie listowania fałszywych
                        ; gałęzi
IFIDN SYM1, <SAMPLE>    ; Linie te zostaną zawsze umieszczone
                        ; w raporcie
      .
      .
      .
ENDIF
```

.CREF, %CREF

ZASTOSOWANIE

Umieszczenie w raporcie tablicy symboli informacji o odwołaniach do poszczególnych symboli

SKŁADNIA

.CREF	*Udostępnienie informacji o odwołaniach do symboli, tryb MASM*
%CREF	*Udostępnienie informacji o odwołaniach do symboli, tryb MASM i IDEAL*
.XCREF	*Zaniechanie umieszczania informacji o odwołaniach do symboli, tryb MASM*
%NOCREF	*Zaniechanie umieszczania informacji o odwołaniach do symboli, tryb MASM i IDEAL*

%CREFALL, %CREFREF, %CREFUREF

KOMENTARZ

W czasie procesu testowania i uruchamiania programu może się przydać informacja o odwołaniach do symboli w programie. Zostanie ona umieszczona w raporcie kompilacji po wydaniu dyrektywy **.CREF** lub **%CREF**.
Natomiast po wydaniu dyrektyw **.XCREF** lub **%NOCREF** informacja ta nie zostanie do raportu dołączona. Przyjmowane są one przez asembler domyślnie.

PATRZ

%CREFALL
%CREFREF
%CREFUREF

PRZYKŁAD

```
.CREF; Udostępnienie informacji o odwołaniach do symboli
        .MODEL  SMALL
        .DATA                   ; Numer tej linii pojawi się
                                ; przy symbolach _DATA i DGROUP
Var1    DW   1234               ; Numer tej linii pojawi się
                                ; przy symbolu Var1
        .CODE                   ; Numer tej linii pojawi się
                                ; przy symbolu _TEXT
Start:                          ; Numer tej linii pojawi się
                                ; przy symbolu Start
        mov   Var1, ax          ; Numer tej linii pojawi się
                                ; przy symbolu Var1
```

%CREFALL, %CREFREF, %CREFUREF

ZASTOSOWANIE

Wybranie symboli, o których informacja zostania umieszczona w tablicy symboli.

SKŁADNIA

%CREFALL	*Informacja o wszystkich symbolach*
%CREFREF	*Informacja o symbolach, do których nastąpiło odwołanie*
%CREFUREF	*Informacja o symbolach, do których nie nastąpiło odwołanie*

KOMENTARZ

Zwykle Turbo Assembler umieszcza w raporcie kompilacji informację o odwołaniach do wszystkich symboli programu. Dyrektywy **%CREFREF** i **%CREFUREF** ograniczają ją do symboli, do których wystąpiły bądź nie wystąpiły odwołania.
Dyrektywa **%CREFALL** poleca umieścić informację o wszystkich symbolach programu. Jest ona przyjmowana standardowo.

PATRZ

.CREF
%CREF
.XCREF
%NOCREF

PRZYKŁAD

```
.CREF                       ; Udostępnienie informacji
                            ; o odwołaniach do symboli
.CREFREF                    ; Informacja tylko o symbolach,
                            ; do których nastąpiło odwołanie
          .MODEL   SMALL
          .DATA             ; Numer tej linii pojawi się
                            ; przy symbolach _DATA i DGROUP
Var1      DW   1234         ; Numer tej linii pojawi się
                            ; przy symbolu Var1
          .CODE             ; Numer tej linii pojawi się
                            ; przy symbolu _TEXT
Start:                      ; Numer tej linii nie pojawi
                            ; w raporcie, ponieważ w
                            ; programie nie ma odwołania
                            ; do symbolu Start
          mov   Var1, ax    ; Numer tej linii pojawi się
przy symbolu Var1
```

%CTLS

ZASTOSOWANIE

Umieszczenie w raporcie kompilacji informacji o dyrektywach, które sterują jego postacią.

SKŁADNIA

%CTLS	*Umieszczenie informacji o dyrektywach, które sterują postacią raportu.*
%NOCTLS	*Niewłączenie do raportu informacji o dyrektywach, które sterują jego postacią.*

KOMENTARZ

Domyślnie Turbo Assembler nie umieszcza w raporcie kompilacji informacji o dyrektywach typu **PAGE**, **%DEPTH**, **%BIN** itp. Natomiast informacja ta pojawi się w nim po wydaniu polecenia **%CTLS**.

PATRZ

.LIST

DISPLAY

%LIST
%NOLIST
.XLIST

PRZYKŁAD

```
PAGE    60, 120 ; Dyrektywa ta nie pojawi się w raporcie
&CTLS           ; Umieszczanie w raporcie informacji
                ; o dyrektywach, które sterują
                ; jego postacią.
%BIN    5       ; Dyrektywa ta oraz pozostałe zostaną
                ; umieszczone w raporcie
        .
        .
        .
%NOCTLS         ; Koniec umieszczania w raporcie informacji
                ; o dyrektywach sterujących jego postacią.
%BIN    4       ; Dyrektywa ta nie pojawi się w raporcie
```

DISPLAY

ZASTOSOWANIE

Wyświetlenie komunikatu w trakcie kompilacji programu.

SKŁADNIA

```
DISPLAY "tekst"
tekst
```

Wyświetlany komunikat.

KOMENTARZ

Dzięki tej dyrektywie może być śledzony proces kompilacji programu. Można więc między innymi sprawdzić, które bloki instrukcji warunkowych zostały skierowane do kompilacji.

PATRZ

%OUT

PRZYKŁAD

```
INCLUDE   CONFIG.INC        ; Włączenie definicji stałych
IF @MODEL EQ 1              ; Wyświetlenie informacji o
                            ; rodzaju modelu, w którym
DISPLAY "TINY MODEL"        ; program jest kompilowany
ELSEIF @MODEL EQ 2
DISPLAY "SMALL MODEL"
ELSEIF @MODEL EQ 3
```

```
DISPLAY "MEDIUM MODEL"
ELSEIF @MODEL EQ 4
DISPLAY "COMPACT MODEL"
ELSEIF @MODEL EQ 5
DISPLAY "LARGE MODEL"
ELSEIF @MODEL EQ 6
DISPLAY "HUGE MODEL"
ENDIF
```

%DEPTH

ZASTOSOWANIE

Ustawienie szerokości pola zagnieżdżenia makroinstrukcji.

SKŁADNIA

```
%DEPTH szerokość
szerokość          Liczba znaków określająca szerokość pola.
```

KOMENTARZ

Standardowo szerokością tego pola jest jeden znak. Jednakże w programach, w których występują makroinstrukcje silnie zagnieżdżone, należy zwiększyć rozmiar tego pola do dwóch znaków.

PATRZ

%PCNT
%BIN
%LINUM
%TEXT

PRZYKŁAD

Patrz przykład do **%BIN**.

EMUL

ZASTOSOWANIE

Emulowanie rozkazów koprocesora.

SKŁADNIA

```
EMUL               Rozkazy koprocesora są emulowane.
```

NOEMUL *Użycie rzeczywistych rozkazów koprocesora.*

KOMENTARZ

Standardowo Turbo Assembler zamienia instrukcje zmiennoprzecinkowe na odpowiednie rozkazy **ESC**. Natomiast po zinterpretowaniu dyrektywy **EMUL** zostaną one zastąpione wywołaniami odpowiednich funkcji z biblioteki matematycznej. Taki sam efekt ma podanie asemblerowi opcji **/e**. Natomiast po wykonaniu polecenia **NOEMUL**, będą one zamieniane na rzeczywiste rozkazy dla koprocesora.
Jeśli w programie została użyta emulacja instrukcji koprocesora, to w czasie konsolidacji musi zostać dołączona odpowiednia biblioteka procedur matematycznych.

PATRZ

JUMPS
NOJUMPS

PRZYKŁAD

```
; Jeśli nie został zainstalowany koprocesor, wówczas
; użyta zostanie emulacja jego instrukcji
        cmp   MathInst, 0   ; Czy w systemie jest
                            ; zainstalowany koprocesor
        je    noMath
        NOEMUL              ; Generowanie rzeczywistych
                            ; rozkazów koprocesora
        fsave  fbuf         ; Rozkaz dla koprocesora
noMath:
        EMUL                ; Emulacja instrukcji
                            ; koprocesora
        fsave  fbuf         ; Wywołanie funkcji
                            ; bibliotecznej
done:
```

.ERR, .ERR1, .ERR2, .ERRB, .ERRDEF
.ERRDIF, .ERRDIFI, .ERRE, .ERRIDN, .ERRIDNI
.ERRNB, .ERRNDEF, .ERRNZ, ERR, ERRIF
ERRIF1, ERRIF2, ERRIFDEF, ERRIFDIF, ERRIFDIFI
ERRIFE, ERRIFIDN, ERRIFIDNI, ERRIFNB, ERRIFNDEF

ZASTOSOWANIE

Wygenerowanie błędu.

SKŁADNIA

Tryb MASM:

`.ERR`	*Bezwarunkowe wygenerowanie błędu.*
`.ERR1`	*Błąd jest generowany przy pierwszym przebiegu asemblera.*
`.ERR2`	*Błąd jest generowany przy drugim przebiegu asemblera.*
`.ERRB arg`	*Błąd jest generowany, jeśli argument arg jest pusty.*
`.ERRDEF symbol`	*Błąd jest generowany, jeśli podany symbol jest zdefiniowany.*
`.ERRDIF arg1, arg2`	*Błąd jest generowany, jeśli arg1 różni się od arg2.*
`.ERRDIFI arg1, arg2`	*Podobnie do .ERRDIF, z tym że ignorowana jest wielkość liter.*
`.ERRE wyr`	*Błąd jest generowany, jeśli wynik wyrażenia wyr jest równy 0.*
`.ERRIDN arg1, arg2`	*Błąd jest generowany, jeśli arg1 jest identyczny jak arg2.*
`.ERRIDNI arg1, arg2`	*Podobnie do .ERRIDN, z tym że ignorowana jest wielkość liter.*
`.ERRNB arg`	*Błąd jest generowany, jeśli argument arg nie jest pusty.*
`.ERRNDEF symbol`	*Błąd jest generowany, jeśli podany symbol nie jest zdefiniowany.*
`.ERRNZ wyr`	*Błąd jest generowany, jeśli wynik wyrażenia wyr jest różny od 0.*

Tryb MASM i IDEAL:

`ERR`	*Bezwarunkowe wygenerowanie błędu.*
`ERRIF wyr`	*Błąd jest generowany, jeśli wyrażenie wyr jest prawdziwe.*
`ERRIF1`	*Błąd jest generowany w pierwszym przebiegu asemblera.*
`ERRIF2`	*Błąd jest generowany w drugim przebiegu asemblera.*
`ERRB arg`	*Błąd jest generowany, jeśli argument arg jest pusty.*
`ERRIFDEF symbol`	*Błąd jest generowany, jeśli podany symbol jest zdefiniowany.*
`ERRIFDIF arg1, arg2`	*Błąd jest generowany, jeśli arg1 różni się od arg2.*
`ERRIFDIFI arg1, arg2`	*Podobnie do ERRIFDIF, z tym że ignorowana jest wielkość liter.*
`ERRIFE wyr`	*Błąd jest generowany, jeśli wynik wyrażenia wyr jest równy 0.*
`ERRIFIDN arg1, arg2`	*Błąd jest generowany, jeśli arg1 jest identyczny jak arg2.*
`ERRIFIDNI arg1, arg2`	*Podobnie do ERRIFIDN, z tym że ignorowana jest wielkość liter.*
`ERRIFNB arg`	*Błąd jest generowany, jeśli argument arg nie jest pusty.*
`ERRIFNDEF symbol`	*Błąd jest generowany, jeśli podany symbol nie jest zdefiniowany.*

KOMENTARZ

Dyrektywy te pozwalają rozszerzyć listę błędów Turbo Assemblera o błędy zdefiniowane przez programistę. Najczęściej są one stosowane w makroinstrukcjach do sprawdzania zakresu podanych argumentów lub w instrukcjach warunkowych przy testowaniu używanych w nich symboli.

Błędy wygenerowane przez te dyrektywy traktowane są przez kompilator jak zwykłe błędy. Powodują wypisanie odpowiedniego komunikatu i zwiększenie licznika błędów. Komuni-

IDEAL

kat ten zawiera numer linii, w której wystąpił błąd oraz informację o tym, że był on spowodowany dyrektywą. Ponadto programista może, korzystając z dyrektyw **%OUT** lub **DISPLAY**, wyświetlić swoją własną wiadomość. Plik wynikowy dla modułu, w którym wystąpiły błędy, nie zostanie utworzony.

PATRZ

%OUT
DISPLAY

PRZYKŁAD

```
MyMacro   MACRO row, col
.ERRNZ row < 0 or row > 24     ; Błąd, gdy argument row
                               ; znajduje się poza zakresem
.ERRNZ col < 0 or row > 79     ; Błąd, gdy argument col
                               ; znajduje się poza zakresem
          .
          .
          .
          ENDM
```

IDEAL

ZASTOSOWANIE

Przełączenie Turbo Assemblera w tryb IDEAL.

SKŁADNIA

IDEAL

KOMENTARZ

Turbo Assembler jest zgodny z asemblerem MASM firmy Microsoft. Dyrektywa **IDEAL** powoduje natomiast przełączenie go w tryb IDEAL. Programy, napisane w tym trybie, nie są już zgodne z programami napisanymi w asemblerze MASM, ale ich kompilacja jest o 30% szybsza.

PATRZ

MASM

PRZYKŁAD

```
IDEAL
        MODEL   SMALL        ; Dyrektywa wyboru modelu
                             ; pamięci w trybie IDEAL
        DATASEG              ; Uproszczona dyrektywa
```

```
                        ; segmentowa w trybie IDEAL,
                        ; która definiuje segment danych
.
.
CODESEG                 ; Uproszczona dyrektywa
                        ; segmentowa w trybie IDEAL,
                        ; która definiuje segment kodu
PROC _main              ; Składnia w trybie IDEAL
.
.
.
ENDP _main              ; Składnia w trybie IDEAL
.
.
.
```

%INCL

ZASTOSOWANIE

Umieszczanie zawartości plików nagłówkowych w raporcie kompilacji.

SKŁADNIA

`%INCL`	*Umieszczanie zawartości plików nagłówkowych w raporcie kompilacji.*
`%NOINCL`	*Niewłączanie zawartości plików nagłówkowych do raportu kompilacji.*

KOMENTARZ

Standardowo Turbo Assembler umieszcza zawartość plików nagłówkowych w raporcie kompilacji. Prowadzić to może jednak do powstawania zbyt dużych zbiorów, w których na dodatek pewna część informacji jest identyczna w wielu raportach. Dlatego też warto stosować dyrektywę **%NOINCL**, dzięki której zawartość plików nagłówkowych nie jest już w raporcie kompilacji umieszczana.

PATRZ

.LALL
%MACS
%NOMACS
.SALL
.XALL
.LIST
%LIST
%NOLIST
.XLIST

JUMPS

PRZYKŁAD

```
%NOINCL                ; Zawartość tego pliku nie pojawi się
                       ; w raporcie kompilacji
INCLUDE defs.inc
```

JUMPS

ZASTOSOWANIE

Ustalenie sposobu reagowanie przez asembler na skoki warunkowe, które przekraczają dopuszczalny zakres.

SKŁADNIA

JUMPS	*Wygenerowanie kodu, który emuluje skok warunkowy, przekraczający dopuszczalny zakres.*
NOJUMPS	*Wygenerowanie błędu, gdy skok warunkowy przekracza dopuszczalny zakres.*

KOMENTARZ

Etykiety w skokach warunkowych mogą być odległe od instrukcji skoku o nie więcej niż -128 do +127 bajtów. Turbo Assembler może jednak wygenerować kod, który pozwala przekazać sterowanie w dalszy obszar programu. W tym celu testowany jest warunek przeciwny do podanego i gdy jest on spełniony, to omijana jest instrukcja bliskiego skoku. W przeciwnym razie instrukcja jest wykonywana i realizuje ona żądany skok.
Standardowo Turbo Assembler po napotkaniu niepoprawnego skoku generuje błąd. Natomiast po zinterpretowaniu dyrektywy **JUMPS**, skoki te zostaną odpowiednio przekształcone.

PRZYKŁAD

```
            JUMPS                ; Zezwolenie na długie
                                 ; skoki warunkowe
            jnc   distant        ; Asembler rozkaz ten
                                 ; przekształci następująco:
                                 ; jc   @@a
                                 ; jmp  distant
                                 ; @@a:
            .                    ; Więcej niż 128 bajtów
            .
            .
distant:
            .
            .
            .
```

.LALL

ZASTOSOWANIE

Wybranie sposobu umieszczania w raporcie rozwinięć makroinstrukcji.

SKŁADNIA

Tryb MASM:

.LALL	*Umieszczanie rozwinięć makroinstrukcji.*
.SALL	*Niewłączanie do raportu rozwinięć makroinstrukcji.*
.XALL	*Rozwinięcia są włączane tylko wtedy, gdy jest generowany kod wynikowy.*

Tryb IDEAL:

%MACS	*Umieszczanie rozwinięć makroinstrukcji.*
%NOMACS	*Niewłączanie do raportu rozwinięć makroinstrukcji.*

KOMENTARZ

Standardowo Turbo Assembler umieszcza w raporcie każdą linię makroinstrukcji przy każdym jej wystąpieniu w zbiorze źródłowym. Można skrócić raport, zakazując umieszczania w nim tych rozwinięć dyrektywami **.SALL** lub **%NOMACS**.
Wypisywanie makrodefinicji może zostać przywrócone dyrektywą **.LALL** lub **%MACS**. Po zinterpretowaniu polecenia **.XALL** w raporcie zostaną umieszczone tylko te makroinstrukcje, które spowodowały wygenerowanie kodu wynikowego. Tak więc instrukcje warunkowe oraz dyrektywy, które znajdują się w makroinstrukcji nie zostaną włączone.

PATRZ

%INCL
%NOINCL

PRZYKŁAD

```
; Makrodefinicja zostanie umieszczona w raporcie
MyMacro   MACRO Arg1, Arg2
          push  ax
          mov   ax, Arg1
          add   Arg2, ax
          pop   ax
          ENDM
%NOINCL                          ; Niewłączanie do raportu
                                 ; rozwinięć makroinstrukcji
          .DATA
Result    DW    1234
          .CODE
          MyMacro <Result, 44>   ; Rozwinięcie tej
                                 ; makroinstrukcji nie
```

```
                                        ; zostanie umieszczone
                                        ; w raporcie
%INCL                                   ; Przywrócenie włączania
                                        ; do raportu rozwinięć
                                        ; makroinstrukcji
          MyMacro <cx, dx>              ; Rozwinięcie tej
                                        ; makroinstrukcji zostanie
                                        ; umieszczone w raporcie
```

.LFCOND

ZASTOSOWANIE

Umieszczanie w raporcie kompilacji nie skompilowanych gałęzi instrukcji kompilacji warunkowej.

SKŁADNIA

.LFCONDW trybie MASM umieszczanie w raporcie kompilacji również nie skompilowanych gałęzi instrukcji kompilacji warunkowej.

PRZYKŁAD

Patrz przykład do **%CONDS**.

%LINUM

ZASTOSOWANIE

Ustalenie szerokości pola, która zawiera numer linii.

SKŁADNIA

```
%LINUM szerokość
```

szerokość — *Liczba znaków, która określa rozmiar pola zawierającego numer linii.*

KOMENTARZ

Wszystkie wiersze raportu kompilacji są ponumerowane. Szerokość pola, które przechowuje numer linii domyślnie jest ustawiona na cztery znaki. Jeśli jednak raport ma więcej niż 9999 linii, szerokość tego pola powinna zostać zwiększona. Z drugiej strony może zaistnieć potrzeba zmniejszenia rozmiaru tego pola dla zostawienia wystarczającej ilości miejsca dla pozostałych pól.

PATRZ

%BIN
%DEPTH
%PCNT
%TEXT

PRZYKŁAD

Patrz przykład do **%BIN**.

.LIST, %LIST

ZASTOSOWANIE

Włączenie lub wyłączenie generowania raportu.

SKŁADNIA

Tryb MASM:

`.LIST`	*Włączenie generowania raportu.*
`.XLIST`	*Wyłączenie generowania raportu.*

Tryb MASM i IDEAL:

`%LIST`	*Włączenie generowania raportu.*
`%NOLIST`	*Wyłączenie generowania raportu.*

KOMENTARZ

Dzięki tym dyrektywom można umieścić w raporcie tylko interesujące fragmenty kodu. Raport staje się wtedy czytelniejszy.

PATRZ

%MACS
%NOMACS
%INCL
%NOINCL

PRZYKŁAD

```
.XLIST                          ; Poniższe linie w raporcie
                                ; nie zostaną umieszczone
          .MODEL   SMALL
INCLUDE   STRUCS.INC
          .DATA
          .
          .
```

```
            .
            .CODE
.LIST                          ; Po ominięciu mało
                               ; interesującej części
                               ; programy, generowanie raportu
                               ; zostaje włączone
            .
            .
.XLIST                         ; w raporcie znajdzie się
                               ; tylko kod programu
                               ; Wyłączenie generowania
                               ; raportu
            .STACK
END
```

%MACS

Patrz **.LALL**.

MASM

ZASTOSOWANIE

Przełączenie Turbo Assemblera w tryb MASM.

SKŁADNIA

MASM

KOMENTARZ

Standardowo Turbo Assembler pracuje w trybie MASM. W trybie tym może on kompilować programy powstałe dla asemblera MASM firmy Microsoft. Poza tym Turbo Assembler posiada swój własny tryb IDEAL, który różni się od MASM inną składnią i zestawem dyrektyw.

PATRZ

IDEAL

PRZYKŁAD

```
IDEAL                  ; Przełączenie asemblera w tryb
                       ; IDEAL
```

```
        MODEL SMALL
        DATASEG
        .
        .
        .
MASM                    ; Ponowne ustawienie trybu MASM
        .CODE           ; Obowiązuje więc już składnia tego
                        ; właśnie trybu
Start   PROC
```

MASM51

ZASTOSOWANIE

Udostępnienie dyrektyw asemblera MASM 5.1.

SKŁADNIA

MASM51	*Udostępnienie dyrektyw asemblera MASM 5.1.*
NOMASM51	*Zakończenie używania dyrektyw asemblera MASM 5.1.*

KOMENTARZ

Turbo Assembler jest zgodny z wersjami od 1 do 4 asemblera MASM. By wykorzystać polecenia charakterystyczne dla wersji MASM 5.1, należy użyć dyrektywy MASM51. Cechy tej wersji to:

- Dyrektywy **SUBSTR**, **CATSTR**, **SIZESTR** i **INSTR**
- Linie mogą być kontynuowane po znaku backslash (\)
- Zmieniona składnia dyrektywy **LOCAL**:

```
LOCAL Nazwa [ilość] [:[zakres] PTR [typ]]
```

PATRZ

QUIRKS

PRZYKŁAD

```
MASM51                          ; Udostępnienie dyrektyw
                                ; asemblera MASM 5.1
Str4 EQU <Oto ciąg znaków>
Str2 EQU < połączony z innym>
Str1 CATSTR Str4, Str2          ; Użyta dyrektywa asemblera
                                ; MASM 5.1. Bez polecenia
                                ; MASM51 w linii tej
                                ; powstałby błąd składniowy
```

MULTERRS

ZASTOSOWANIE

Zgłaszanie tylko jednego błędu, nawet dla linii, w których powstało ich wiele.

SKŁADNIA

MULTERRS	*Zgłoszenie wszystkich błędów, które pojawiły się w linii.*
NOMULTERRS	*Zgłoszenie tylko jednego błędu dla każdej niepoprawnej linii.*

KOMENTARZ

Standardowo wyświetlany jest tylko jeden błąd dla każdej niepoprawnej linii. Założenie jest bowiem takie, że programista, poprawiając zasygnalizowany w danej linii błąd, zauważy również pozostałe. Jednakże czasami pojawiają się błędy bardzo trudne do wykrycia. Na przykład, w pojedynczej instrukcji mogą zostać użyte dwa argumenty, które odwołują się do pamięci, a jeden z nich może używać nie zdefiniowanego symbolu. W tym wypadku nie jest trudno dostrzec jeden błąd, drugi może zostać jednak pominięty. Natomiast po zinterpretowaniu dyrektywy **MULTERRS** zostaną wypisane wszystkie błędy znalezione przez asembler.

Wyświetlanie wszystkich błędów może jednak prowadzić do pojawienia się wielu komunikatów dotyczących tego samego problemu. Może się to zdarzyć, gdy w definicji zostanie użyty nie zdefiniowany symbol. Najpierw bowiem asembler zasygnalizuje użycie nie zdefiniowanego symbolu, jak również błędy, które pojawią się przy każdym odwołaniu się do symbolu, który powinien zostać utworzony.

PATRZ

WARN
NOWARN

PRZYKŁAD

```
MULTERRS                        ; Zezwolenie na wyświetlanie
                                ; kilku błędów dotyczących
                                ; tej samej linii
          mov   7, abc          ; Niepoprawne argumenty i abc
                                ; jest nie zdefiniowany
```

NAME

ZASTOSOWANIE

Ustalenie nazwy modułu w pliku wynikowym.

SKŁADNIA

```
NAME nazwa_modułu
nazwa_modułu        Nazwa modułu w pliku wynikowym.
```

KOMENTARZ

W każdym pliku wynikowym istnieje pole, które przechowuje nazwę modułu. Nazwa ta jest wykorzystywana przez konsolidator w czasie wypisywania błędów, które pojawiły się w danym zbiorze wynikowym. Standardowo Turbo Assembler nazwę tę przyjmuje jako nazwę pliku źródłowego modułu. Można ją jednak zmienić, wykorzystując dyrektywę **NAME**.

Użycie dyrektywy **NAME** jest jednak konieczne tylko wtedy, gdy pewna aplikacja wymaga, by nazwy modułów były różne. W takim wypadku należy, przed użyciem dyrektywy, przełączyć asembler w tryb IDEAL. W trybie MASM dyrektywa nie powoduje żadnego działania.

PRZYKŁAD

```
IDEAL               ; Dyrektywa NAME istnieje tylko
                    ; w trybie IDEAL
NAME Module1        ; Ustawienie nazwy modułu
.
.
.
```

%NEWPAGE

ZASTOSOWANIE

Utworzenie nowej strony raportu.

SKŁADNIA

```
%NEWPAGE            Utworzenie nowej strony raportu.
```

KOMENTARZ

Dyrektywa **%NEWPAGE** jest używana przy umieszczaniu oddzielnej części programu na nowej stronie raportu. Pozwala to na utrzymanie przejrzystości wydruku i ułatwia jego czytanie.

PATRZ

%BIN
%DEPTH
%LINUM
%PCNT
%TEXT

%NOCONDS

PRZYKŁAD

Patrz przykład do **%PAGESIZE**.

%NOCONDS

Patrz opis dyrektywy **%CONDS**.

%NOCREF

Patrz opis dyrektywy **%CREF**.

%NOCTLS

Patrz opis dyrektywy **%CTLS**.

NOEMUL

Patrz opis dyrektywy **EMUL**.

%NOINCL

Patrz opis dyrektywy **%INCL**.

NOJUMPS

Patrz opis dyrektywy **JUMPS**.

%NOLIST

Patrz opis dyrektywy **%LIST**.

%NOMACS

Patrz opis dyrektywy **.LALL**.

NOMASM51

Patrz opis dyrektywy **MASM51**.

NOMULTERRS

Patrz opis dyrektywy **MULTERRS**.

%NOSYMS

Patrz opis dyrektywy **%SYMS**.

%NOTRUNC

Patrz opis dyrektywy **%TRUNC**.

NOWARN

NOWARN

Patrz opis dyrektywy **WARN**.

%OUT

ZASTOSOWANIE

Wyświetlenie komunikatu w trakcie kompilacji programu.

SKŁADNIA

```
%OUT tekst
tekst                    Wyświetlany komunikat.
```

KOMENTARZ

Dyrektywa **%OUT** powoduje wyświetlenie na ekranie komunikatu. Podany tekst nie jest w żaden sposób interpretowany. Dzięki tej dyrektywie może być śledzony proces kompilacji programu. Można więc między innymi sprawdzić, które bloki instrukcji warunkowych zostały skierowane do kompilacji.

Obie dyrektywy **%OUT** oraz **DISPLAY** wyświetlają na ekranie komunikat. Właściwie są one identyczne, z tym że tekst w poleceniu **DISPLAY** musi być objęty w cudzysłów.

PATRZ

DISPLAY

PRZYKŁAD

```
INCLUDE   CONFIG.INC      ; Włączenie definicji stałych
IF @MODEL EQ 1            ; Wyświetlenie informacji
                          ; o rodzaju modelu, w którym
DISPLAY "TINY MODEL"      ; program jest kompilowany
ELSEIF @MODEL EQ 2
DISPLAY "SMALL MODEL"
ELSEIF @MODEL EQ 3
DISPLAY "MEDIUM MODEL"
ELSEIF @MODEL EQ 4
DISPLAY "COMPACT MODEL"
ELSEIF @MODEL EQ 5
DISPLAY "LARGE MODEL"
ELSEIF @MODEL EQ 6
DISPLAY "HUGE MODEL"
ENDIF
```

P186

Wersja dyrektywy **.186** w trybie **IDEAL**.

P286

Wersja dyrektywy **.286P** w trybie **IDEAL**.

P286N

Wersja dyrektywy **.286** w trybie **IDEAL**.

P287

Wersja dyrektywy **.287** w trybie **IDEAL**.

P386

Wersja dyrektywy **.386P** w trybie **IDEAL**.

P386N

Wersja dyrektywy **.386** w trybie **IDEAL**.

P387

Wersja dyrektywy **.387** w trybie **IDEAL**.

P8086

Wersja dyrektywy **.8086** w trybie **IDEAL**.

P8087

Wersja dyrektywy **.8087** w trybie **IDEAL**.

PNO87

ZASTOSOWANIE

Zakaz używania instrukcji koprocesora 8087.

SKŁADNIA

PNO87 *Zakaz używania instrukcji koprocesora 8087.*

KOMENTARZ

Użycie dowolnej instrukcji koprocesora za tą dyrektywą spowoduje powstanie błędu. Dyrektywa **PNO87** może być zastosowana w programie w celu upewnienia się, że nie wystąpiły w nim żadne rozkazy koprocesora.

PATRZ

.186
.286
.286P
.287
.386
.386P
.387
.8086
.8087

PRZYKŁAD

W programie tym zastosowano instrukcje kompilacji warunkowej. Dzięki temu można stworzyć dwie wersje programu. Jedna używa do obliczeń koprocesora arytmetycznego, a druga działa tylko na liczbach całkowitych. Dyrektywa **PNO87** została użyta w celu upewnienia się, że w drugiej wersji nie zastosowano żadnego rozkazu koprocesora.

```
IFDEF   Float                       ; Sprawdzenie, czy
                                    ; symbol Float jest
                                    ; zdefiniowany
                                    ; (Powinien zostać
                                    ; zdefiniowany przez
                                    ; programistę przed
                                    ; kompilacją modułu)
        P8087                       ; Zapewnienie, że
                                    ; instrukcje
                                    ; koprocesora są
                                    ; dostępne
ELSE
        PNO87                       ; Zakaz używania
                                    ; instrukcji koprocesora
ENDIF
; Słowo na szczycie ramki stosu oznacza typ dodawanych
; danych (1 = rzeczywiste, 2 = całkowite)
AddEm   PROC  NEAR
        push  bp                    ; Zachowaj rejestr BP
        mov   bp, sp                ; BP wskazuje na ramkę
                                    ; stosu.
        mov   ax, WORD [bp-4]       ; Pobierz słowo, które
                                    ; określa typ danych
IFDEF   Float                       ; Jeśli dozwolone
                                    ; operacje
                                    ; zmiennoprzecinkowe
        cmp   ax, 1
            ; Czy są dane
                                    ; rzeczywiste
        jne   NotFloat
        fld   QWORD [bp-6]          ; Pobierz pierwszą
                                    ; liczbę
        fadd  QWORD [bp-14]         ; Dodaj do niej drugą
        fsd   QWORD [bp-6]          ; Zapisz wynik
                                    ; z powrotem na stos
NotFloat:
ENDIF
        cmp   ax, 2
            ; Czy są dane całkowite
        jne   NotInt
```

```
        mov     ax, WORD [bp-8]     ; Pobierz drugą liczbę
        add     WORD [bp-6], ax     ; Dodaj ją do pierwszej
NotInt:
        pop     bp
        ret
AddEm   ENDP
```

PAGE

ZASTOSOWANIE

Sterowanie podziałem raportu na strony.

SKŁADNIA

`PAGE`	*Utworzenie nowej strony.*
`PAGE [wiersze][,kolumny]`	*Zdefiniowanie liczby wierszy i kolumn na stronie.*
`PAGE +`	*Utworzenie nowej sekcji.*

KOMENTARZ

Raport kompilacji może mieć od 59 do 255 kolumn szerokości. Im jest on szerszy, tym więcej informacji może zostać w nim umieszczone, a jego postać staje się łatwiejsza do czytania. Przede wszystkim jednak rozmiar raportu powinien być dopasowany do urządzenia, na którym będzie on drukowany.

Na każdej stronie umieszczana jest nazwa pliku, jego tytuł i podtytuł, numer strony itd. Główną jej część zajmuje jednak treść programu źródłowego. Należy określić ilość wierszy, które przypadają na stronę tak, by jak najlepiej dopasować ją do urządzenia, na którym raport ma być drukowany.

Dyrektywa **PAGE** służy również do wymuszania drukowania raportu od nowej strony. Pomaga to dzielić wydruk na rozdziały i sekcje.

PATRZ

%BIN
%DEPTH
%LINUM
%PCNT
%TEXT
%NEWPAGE
%PAGESIZE

PRZYKŁAD

```
PAGE    50, 70          ; Utworzenie strony, która mieści 50
                        ; wierszy o szerokości 70 znaków
.MODEL    SMALL
```

```
.          ; Definicje struktur i stałych
.
.
PAGE       ; Zacznij wydruk od nowej strony
.DATA      ; Definicje danych
.
.
.
PAGE       ; Zacznij wydruk od nowej strony
.CODE      ; Kod programu będzie zaczynał się od nowej strony
.
.
.
```

%PAGESIZE

ZASTOSOWANIE

Ustalenie rozmiaru strony raportu.

SKŁADNIA

```
%PAGESIZE [wiersze][,kolumny]
```

Zdefiniowanie liczby wierszy i kolumn na stronie.

KOMENTARZ

Raport kompilacji może mieć od 59 do 255 kolumn szerokości. Im jest on szerszy, tym więcej informacji może zostać w nim umieszczone, a jego postać staje się łatwiejsza do czytania. Przede wszystkim jednak rozmiar raportu powinien być dopasowany do urządzenia, na którym będzie on drukowany.

Na każdej stronie umieszczana jest nazwa pliku, jego tytuł i podtytuł, numer strony itd. Główną jej część zajmuje jednak treść programu źródłowego. Należy określić ilość wierszy, które przypadają na stronę tak, by jak najlepiej dopasować ją do urządzenia, na którym raport ma być drukowany.

PATRZ

%BIN
%DEPTH
%LINUM
%PCNT
%TEXT
PAGE
%NEWPAGE

%PCNT

PRZYKŁAD

```
%PAGESIZE 50, 70 ; Utworzenie strony, która mieści 50
                 ; wierszy o szerokości 70 znaków
.MODEL   SMALL
.                ; Definicje struktur i stałych
.
.
%NEWPAGE         ; Zacznij wydruk od nowej strony
.DATA            ; Definicje danych
.
.
.
%NEWPAGE         ; Zacznij wydruk od nowej strony
.CODE            ; Kod programu będzie zaczynał się
                 ; od nowej strony
.
.
.
```

%PCNT

ZASTOSOWANIE

Ustalenie szerokości pola adresu w raporcie kompilacji.

SKŁADNIA

```
%PCNT szerokość
szerokość
```

Liczba znaków, która określa rozmiar pola adresu.

KOMENTARZ

W polu adresu umieszczane jest przemieszczenie i numer segmentu, który odpowiada każdej linii pliku. Standardowo jego rozmiar wynosi cztery znaki dla segmentów 16-bitowych i osiem dla segmentów 32-bitowych. Wystarcza to więc na umieszczenie w polu tylko samego przemieszczenia. Jeśli wymagany jest również numer segmentu, to należy liczbę kolumn zwiększyć o pięć. Jeśli natomiast szerokość pola zostanie zmniejszona, wówczas asembler umieści w nim maksymalną liczbę najmniej znaczących cyfr adresu.

PATRZ

%BIN
%DEPTH
%LINUM
%TEXT

PRZYKŁAD

Patrz przykład do **%BIN**.

%POPLCTL

ZASTOSOWANIE

Zachowywanie i odtwarzanie stanu zmiennych, które sterują postacią raportu.

SKŁADNIA

%POPLCTL	*Pobranie ze stosu LCTL zmiennych, które sterują postacią raportu.*
%PUSHLCTL	*Zapisanie na stos LCTL zmiennych, które sterują postacią raportu.*

KOMENTARZ

Dyrektywy te są używane do zapamiętania i odtworzenia pewnych zmiennych sterujących postacią raportu. Operacja ta dotyczy bowiem tylko zmiennych dwustanowych, takich jak **%MACS**, **%LIST**, **%INCL**, itp. Nie można natomiast zapisać zmiennych, którym przypisywana jest wartość np. **%BIN** czy **%PAGESIZE**.
Dyrektywy te są używane często w makrodefinicjach i plikach nagłówkowych do lokalnej zmiany niektórych parametrów wydruku. Stos LCTL może pomieścić 16 kompletów zmiennych.

PATRZ

%MACS
%LIST
%INCL
MULTERRS
JUMPS
%SYM
%CREF

PRZYKŁAD

```
; Fragment pliku nagłówkowego.
%PUSHLCTL       ; Zachowaj parametry raportu ustawione
                ; przez program, który korzysta z tego pliku
%NOLIST         ; Zawartość pliku nie będzie umieszczona
                ; w raporcie
.
.
.
%POPLCTL        ; Odtwórz zapamiętany stan parametrów
                ; raportu.
```

%PUSHLCTL

Patrz opis dyrektywy **%POPLCTL**.

QUIRKS

ZASTOSOWANIE

Zapewnienie pełnej zgodności z asemblerem MASM poprzez udostępnienie jego kilku rzadko stosowanych cech.

SKŁADNIA

QUIRKS *Zapewnienie pełnej zgodności z asemblerem MASM.*

KOMENTARZ

Programiści potrafią zawsze znaleźć i wykorzystać sztuczki, na które asembler pozwala, a których istnienie wcale nie jest zamierzone. By Turbo Assembler poprawnie zinterpretował programy, w których zostały one użyte, należy użyć dyrektywy **QUIRKS**.

PATRZ

MASM
MASM51

PRZYKŁAD

```
MASM51                          ; Zgodność z asemblerem
                                ; MASM 5.1
QUIRKS                          ; Możliwość wystąpienia
                                ; sztuczek
          .MODEL  SMALL, C
SEGADDR   DB   ?                ; Należy zauważyć, że
                                ; zostało zarezerwowane
                                ; zbyt mało miejsca na
                                ; rejestr segmentowy
DUMMY     DW   ?
          .DATA
DMBADDR   =  OFFSET X
          .CODE
Funct1    PROC                  ; Definicja funkcji
                                ; publicznej o nazwie
                                ; _Funct1
          mov   SEGADDR, es     ; w trybie QUIRKS
                                ; instrukcja ta jest
```

```
                                ; poprawna
        jmp   FAR PTR lab1      ; Skok ten zostanie
                                ; przekształcony na
                                ; skok bliski
        mov   ax, DMBADDR       ; Instrukcja ta powoduje
                                ; pobranie wartości
                                ; zmiennej DUMMY, a nie
                                ; jej adresu
        .
        .
lab1    LABEL FAR
```

.SALL

Patrz opis dyrektywy **.LALL**.

.SFCOND

ZASTOSOWANIE

Wyłącznie umieszczania w raporcie nie skompilowanych gałęzi instrukcji kompilacji warunkowej.

SKŁADNIA

.SFCOND	*W trybie MASM wyłączenie umieszczania w raporcie nie skompilowanych gałęzi instrukcji kompilacji warunkowej.*

PRZYKŁAD

Patrz przykład do **.TFCOND**.

SUBTTL, %SUBTTL

ZASTOSOWANIE

Ustalenie podtytułu raportu.

%SYMS

SKŁADNIA

Tryb MASM:

```
SUBTTL tekst
```

Tryb MASM i IDEAL:

```
%SUBTTL "tekst"
```

KOMENTARZ

W raporcie jego tytuł i podtytuł są drukowane na każdej stronie. Dyrektywy **SUBTTL** i **%SUBTTL** pozwalają zdefiniować podtytuł, który zostanie użyty na następnej stronie.

PATRZ

TITLE
%TITLE

PRZYKŁAD

```
%PAGESIZE 50, 70
%SUBTTL   "Przykładowy podtytuł"
.
.
.
```

%SYMS

ZASTOSOWANIE

Ustalenie, czy tablica symboli ma być umieszczona w raporcie kompilacji.

SKŁADNIA

`%SYMS`	*Umieszczenie w raporcie tablicy symboli.*
`%NOSYMS`	*Tablica symboli nie zostanie w raporcie umieszczona.*

KOMENTARZ

Standardowo Turbo Assembler umieszcza w raporcie kompilacji informację o tablicy symboli. Można tego zaniechać, stosując dyrektywę **%NOSYMS**. Polecenie **%SYMS** przywraca poprzedni stan.

PATRZ

%CREF
%NOCREF

PRZYKŁAD

```
%NOSYMS          ; Tablica symboli nie pojawi się w
                 ; raporcie kompilacji
        .
        .
        .
```

%TABSIZE

ZASTOSOWANIE

Ustalenie liczby kolumn, która odpowiada znakowi tabulacji.

SKŁADNIA

```
%TABSIZE kolumny
```

kolumny — *Liczba kolumn, która odpowiada znakowi tabulacji.*

KOMENTARZ

Turbo Assembler używa znaku tabulacji do wypełniania pustych części pól. Jeśli na urządzeniu, na którym raport ma być drukowany, rozmiar tabulacji jest inny, to wydruk może stać się nieczytelny. Standardowo rozmiar tabulacji wynosi osiem znaków.

PATRZ

%BIN
%DEPTH
%LINUM
%PAGE
%PCNT
%TEXT

PRZYKŁAD

Patrz przykład do **%BIN**.

%TEXT

ZASTOSOWANIE

Ustalenie szerokości pola tekstu źródłowego.

.TFCOND

SKŁADNIA

`%TEXT szerokość`

`szerokość` *Liczba znaków określająca rozmiar pola źródłowego.*

KOMENTARZ

Ostatnim polem w linii raportu jest pole, które zawiera tekst programu źródłowego. Linia wypełnia pozostałą część strony. Jeśli tekst nie mieści się w jednej linii, wówczas zostanie przeniesiony do następnej albo obcięty, zależnie od stanu polecenia **%TRUNC**.
Dyrektywa **%TEXT** pozwala na ustalenie szerokości tego pola, niezależnie od dyrektywy **PAGE**. W poleceniu **PAGE** bowiem rozmiar pola zależy od wymiarów pozostałych pól. Dzięki dyrektywie **%TEXT** można bezpośrednio ustawić żądaną szerokość pola, które przechowuje tekst źródłowy.

PATRZ

PAGE
%PAGESIZE
%TRUNC
%NOTRUNC
%BIN
%DEPTH
%PCNT
%LINUM

.TFCOND

ZASTOSOWANIE

Sterowanie umieszczaniem w raporcie kompilacji nie skompilowanych gałęzi instrukcji kompilacji warunkowej.

SKŁADNIA

`.TFCOND` *W trybie MASM włączenie lub wyłącznie umieszczania w raporcie kompilacji nie skompilowanych gałęzi instrukcji kompilacji warunkowej.*

KOMENTARZ

Standardowo Turbo Assembler umieszcza w raporcie również i fałszywa gałęzie instrukcji warunkowych. W trybie MASM dyrektywa **.TFCOND** służy do zmiany tego stanu na przeciwny.

PATRZ

%MAC
%NOMAC

.LALL
.SALL
.XALL
%CONDS
%NOCONDS
.SFCOND

PRZYKŁAD

```
.SFCOND                 ; Wyłączenie listowania nie
                        ; skompilowanych gałęzi
                        ; w raporcie kompilacji
IFDEF SYM1              ; Poniższe linie w raporcie nie
                        ; pojawią się, gdy symbol SYM1
                        ; nie jest zdefiniowany
      .
      .
ENDIF
.TFCOND                 ; Umieszczanie nie skompilowanych
                        ; gałęzi w raporcie kompilacji
IFIDN SYM1, <SAMPLE>    ; Niezależnie od wartości warunku
                        ; poniższe linie zostaną umieszczone
                        ; w raporcie kompilacji
      .
      .
ENDIF
```

TITLE, %TITLE

ZASTOSOWANIE

Ustalenie tytułu raportu.

SKŁADNIA

Tryb MASM:

```
TITLE tekst
```

Tryb MASM i IDEAL:

```
%TITLE "tekst"
```

KOMENTARZ

W raporcie asembler umieszcza jego tytuł na każdej stronie. Dyrektywy **TITLE** i **%TITLE** pozwalają zdefiniować właśnie ten tytuł. Tytuł będzie jednakowy w całym pliku.

%TRUNC

PATRZ

SUBTTL
%SUBTTL

PRZYKŁAD

```
%TITLE    "Oto przykładowy program"
%SUBTTL   "Część pierwsza"
      .
      .
      .
```

%TRUNC

ZASTOSOWANIE

Ustalenie sposobu reagowania na wiersze, które nie mieszczą się w jednej linii raportu.

SKŁADNIA

%TRUNC	*Obcięcie zbyt długich wierszy.*
%NOTRUNC	*Przeniesienie zbyt długiego wiersza do następnej linii.*

KOMENTARZ

Dzięki tym dyrektywom można wybrać sposób reagowania na wiersze, które nie mieszczą się w całości w jednej linii raportu. Standardowo są one obcinane. Natomiast po wydaniu polecenia **%NOTRUNC** długie wiersze zostaną podzielone. Na końcu pierwszej części wiersza zostanie dodany znak plus (+), a reszta tekstu zostanie przeniesiona do nowej linii.

PATRZ

PAGE
%TEXT

PRZYKŁAD

```
%TRUNC              ; Obcinanie zbyt długich wierszy
          .DATA
Long      DB        "W raporcie linia ta zostanie ucięta"
%NOTRUNC
Another   DB        "Wiersz ten zostanie umieszczony
                     w dwóch liniach"
```

WARN

ZASTOSOWANIE

Wybór klasy ostrzeżeń, na które ma reagować kompilator.

SKŁADNIA

`WARN [klasa]`	*Reagowanie na podaną klasę ostrzeżeń.*
`NOWARN [klasa]`	*Zaniechanie informowania o wystąpieniu ostrzeżenia podanej klasy.*

KOMENTARZ

Użycie dyrektywy **WARN** lub **NOWARN** bez argumentu dotyczy wszystkich klas ostrzeżeń. Klasy te mogą być również wybrane opcją **/W** asemblera. Opis klas znajduje się w tabeli 11-4.

Tabela 11-4. Klasy ostrzeżeń.

Klasa	Ostrzeżenia
ASS	Założenie, że segment jest 16-bitowy
BRK	Potrzebne nawiasy
ICG	Nieefektywny kod
LCO	Przepełnienie wskaźnika pozycji
OPI	Otwarta instrukcja IF
OPP	Otwarta procedura
OPS	Otwarty segment
OVF	Nadmiar arytmetyczny
PDC	Konstrukcja zależna od numeru przebiegu
PRO	Użycie w trybie chronionym przedrostka CS

PRZYKŁAD

```
WARN    OPI         ; Reagowanie na ostrzeżenia, dotyczące
                    ; otwartych instrukcji IF
IF      Enabled
        .
        .
        .
END                 ; Zakończenie modułu przed zakończeniem
                    ; dyrektywy IF powoduje wygenerowanie
                    ; ostrzeżenia
```

.XALL

PATRZ opis dyrektywy **.LALL**.

.XCREF

Patrz opis dyrektywy **.CREF**.

.XLIST

Patrz opis dyrektywy **.LIST**.

IV Techniki programowania

» Moduły asemblerowe w językach wysokiego poziomu

» Wykorzystanie zasobów systemu

» Dostęp i sterowanie sprzętem

» Karta graficzna: tryb tekstowy i graficzny

Rozdział 12 Moduły asemblerowe w językach wysokiego poziomu

Pisanie bazy danych czy edytora tekstów w języku asemblera nie jest najlepszym rozwiązaniem. Oczywiście można to zadanie wykonać, lecz jego złożoność wymaga z użycia języków wysokiego poziomu. Dlatego też większość czasu programiści spędzają pracując właśnie w językach wysokiego poziomu jak C czy Pascal.
Język asemblera musi być zastosowany wszędzie tam, gdzie konieczne jest bezpośrednie operowanie na sprzęcie lub też, gdy kod wynikowy musi być bardzo efektywny. Tak więc, by połączyć łatwość programowania w językach wysokiego poziomu z efektywnością asemblera, potrzebne są sposoby na wywoływanie funkcji i współdzielenie danych pomiędzy modułami.
Większość kompilatorów i asemblerów, które pracują pod kontrolą systemu MS-DOS generuje pliki wynikowe o takim samym formacie. Dzięki temu konsolidator potrafi połączyć w jedną całość zbiory powstałe z różnych plików źródłowych. Niestety, format ten nie precyzuje dokładnie kilku zasad koniecznych do poprawnej współpracy modułów. Tak więc to programista musi zatroszczyć się o zastosowanie odpowiedniej konwencji wywoływania funkcji, przekazywania im parametrów i definiowania zmiennych globalnych.
Każdy język posiada swoje własne metody komunikacji między modułami. W tym rozdziale przedstawiono kilka z nich, używanych przez popularne języki programowania. Dokładniejsza informacja jest zawarta w dokumentacji odpowiedniego kompilatora. Ponadto przeanalizowanie kodu generowanego przez kompilator może dostarczyć wielu cennych wskazówek, które dotyczą wymiany informacji pomiędzy podprogramami. Sposób ten został przedstawiony w dalszej części rozdziału.

Wywołania między modułami w językach wysokiego poziomu a modułami asemblera

Istnieje kilka sposobów na przekazywanie sterowania pomiędzy modułami powstałymi w językach wysokiego poziomu a modułami asemblera. Jednym z nich są przerwania programowe, których używa się do wywołania funkcji BIOS-u i MS-DOS-a. W większości języków wysokiego poziomu istnieją podprogramy, pozwalające na realizowanie takich odwołań. Sposób ten może być więc wykorzystany w programach asemblerowych do komunikacji z innymi modułami. W tym rodzaju komunikacji, do przekazania argumentów wywoływanej funkcji, używane są rejestry procesora. Sposób ten ma również i tę zaletę, że moduły asemblerowe mogą zostać załadowane do pamięci i mogą być dostępne w ten sposób dla wielu innych programów. Dobrym przykładem tego rodzaju modułu jest sterownik myszy, który używa przerwania **33h** jako punktu wywołania swoich procedur.

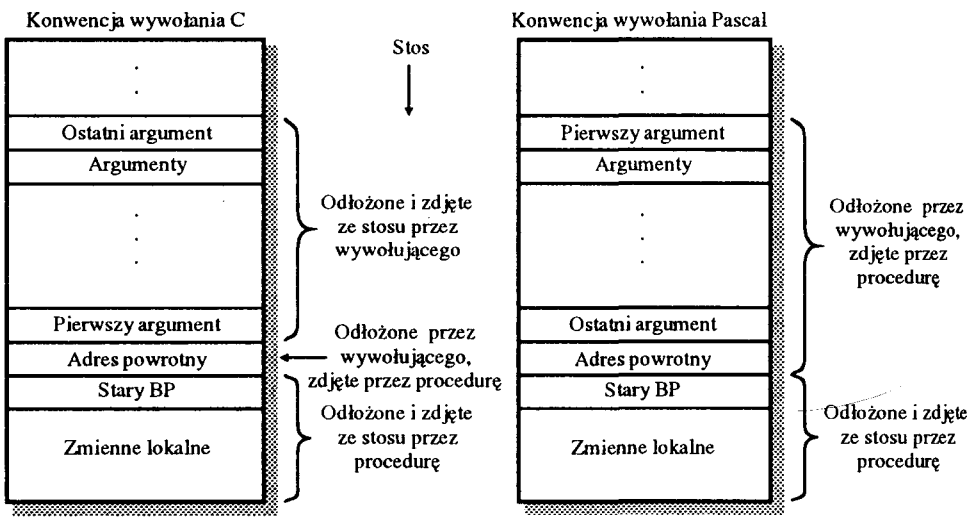
Innym sposobem wymiany informacji pomiędzy modułami języków wysokiego poziomu a modułami asemblera są wywołania podprogramów. W językach BASIC i FORTRAN służą do tego celu instrukcje **CALL** (*CALL PODPROGRAM*). W języku C wszystkie funkcje są podprogramami, tak więc podaje się tylko ich nazwę (Podprogram();). W Pascalu istnieją ponadto procedury. Ich wywołanie składa się tylko z nazwy (Podprogram;). W języku asemblera do przekazania sterowania do innego fragmentu programu służy instrukcja **CALL** (*call Podprogram*).
Oczywiście oprócz instrukcji, która realizuje wywołanie, w każdym języku istnieją polecenia, określające typ wywołania funkcji (NEAR czy FAR), rodzaj jej argumentów i typ zwracanej wartości. Należy zapewnić, by zarówno program wywołujący, jak i podprogram wywoływany, używały tej samej konwencji.

Dyrektywa MODEL

Dyrektywa **MODEL** pozwala na wybranie określonej konwencji wywoływania podprogramów. Określenie modelu pamięci implikuje typ wywołań (NEAR lub FAR) (Patrz Rozdział 8 "Modele pamięci"). Można również podać specyfikator języka, ustalając typ komunikacji między funkcjami. Większość języków stosuje konwencję dla języka C lub Pascal.

Ramki stosu

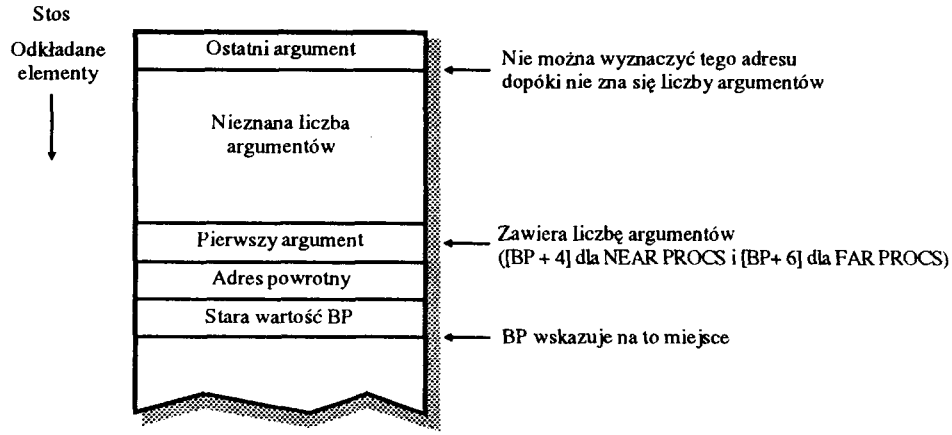
Rysunek 12-1 przedstawia ramki stosu charakterystyczne dla różnych konwencji wywoływania. Różnią się one kolejnością umieszczania na stosie argumentów oraz sposobem kończenia podprogramu. Ramki stosu są dokładniej opisane w rozdziale 10.



Rysunek 12-1. ***Ramki stosu w konwencjach wywoływania języków C i Pascal.***

KONWENCJA WYWOŁYWANIA JĘZYKA C

W języku C argumenty funkcji są umieszczane na stosie w odwrotnym porządku. Oznacza to, że pierwszy argument funkcji znajduje się na szczycie stosu. Dzięki takiemu rozwiązaniu funkcje np. printf mogą mieć zmienną liczbę parametrów. Wtedy pierwszy z nich zawiera informację jak wiele argumentów zostało przekazane. Jeśli zaś byłyby one umieszczone w normalnej kolejności, wówczas funkcja musiałaby mieć dodatkowy argument, który poinformuje ją ilości przesłanych danych. Ilustruje to rysunek 12-2.



Rysunek 12-2. ***Zmienna liczba argumentów.***

W języku C funkcja wywołująca usuwa ze stosu przesłane nań parametry wywołania. Wynika to z istnienia zmiennej ilości argumentów funkcji. Program, który funkcję wywołał wie dokładnie, ile zostało jej przekazanych parametrów, podczas gdy ona musi polegać tylko na argumencie, który zawiera ich liczbę.

KONWENCJA WYWOŁYWANIA JĘZYKA PASCAL

W Pascalu funkcje nie mogą mieć zmiennej liczby parametrów. Argumenty są więc umieszczane na stosie w kolejności ich pojawienia się w wywołaniu. Ponadto znany jest rozmiar stosu zajmowany przez nie. Podprogram może więc usunąć z niego swoje argumenty. Używana jest do tego celu instrukcja **RET n**. Sposób ten jest na tyle efektywny, że wiele kompilatorów języka C pozwala używać tej konwencji dla wybranych przez programistę funkcji. Jest to również ważne dla programisty, który pisze w języku asemblera, ponieważ oznacza to, że, nawet pisząc funkcje dla programów w C, można używać konwencji języka Pascal.

DYREKTYWA ARG

Podając w dyrektywie **MODEL** typ języka, określa się sposób umieszczania i usuwania argumentów funkcji ze stosu. Dzięki poleceniu **ARG**, które nadaje nazwy jej parametrom, można już przestać troszczyć się kolejność argumentów na stosie. Zajmie się tym asembler.

Należy tylko pamiętać, że argumenty w dyrektywie **ARG** powinny wystąpić w tej samej kolejności, w jakiej występują w języku wysokiego poziomu.

OPEROWANIE NA STOSIE ZA POMOCĄ TURBO ASSEMBLERA

W starszych wersjach Turbo Assemblera programista musiał sam, w prawidłowej kolejności, umieszczać argumenty funkcji na stosie. Jednakże od wersji 2.0 Turbo Assembler potrafi zrobić to automatycznie. W tych rozszerzonych wywołaniach należy tylko podać rodzaj użytej konwencji i wymienić argumenty, przekazywane do funkcji:

```
call   MyFunc  C, ax, 0, Data4
```

Pierwszy argument oznacza, że Turbo Assembler ma wygenerować kod, który przesyła na stos następne trzy parametry w sposób używany w języku C. Pierwszym argumentem może być również słowo **PASCAL**. Wtedy parametry wywołania zostaną umieszczone na stosie w porządku stosowanym w języku Pascal.

ZWRACANIE WARTOŚCI

Po zakończeniu swojego działania funkcja może zwrócić programowi wywołującemu pewne dane. Może to zostać zrobione na trzy sposoby. Pierwszy polega na zmodyfikowaniu argumentów już znajdujących się na stosie i oddaniu sterowania do programu wywołującego. W czasie usuwania parametrów ze stosu może on pobrać tę zmodyfikowaną wartość. W języku C argumenty pozostają na stosie, tak więc funkcja może dowolnie je zmieniać. Natomiast w konwencji języka Pascal muszą być podjęte specjalne kroki. Dyrektywa **RETURNS** określa argumenty, które mają pozostać na stosie. Sposób ten, rzadko używany, wymaga ponadto, by parametry te znajdowały się w określonym porządku.
Podobną i częściej stosowaną techniką jest przekazanie adresu zmiennej, która ma być zmodyfikowana, zamiast przekazywania jej samej. Ten sposób nazywany jest przekazywaniem przez referencję, w odróżnieniu do przekazywania przez wartość. Funkcja **time**, która znajduje się w standardowej bibliotece języka C, jest dobrym przykładem, w którym zastosowano przekazywanie przez referencję. Funkcja ta umieszcza w zmiennej typu long int liczbę sekund, które upłynęły od dnia 1 stycznia 1980 roku. Zmienna ta jest przekazywana do funkcji przez referencję:

```
#include <time.h>
/* Funkcja ta wywołuje funkcję time w celu pobrania aktual-
nego czasu */
Func()
{
   time_t   seconds;     /* deklaracja zmiennej */
                         /* przechowującej aktualny czas */
                         /* typ time_t jest zdefiniowany */
                         /* z pliku time.h, */
                         /* zwykle jest to typ long */
                         /* int (32 bity) */
   time( &seconds );     /* przekazanie adresu zmiennej */
                         /* seconds do funkcji time */
}
```

W języku C komentarz jest oznaczany parą znaków slash-gwiazdka (/*...*/). Średniki mają natomiast zupełnie inne znaczenie.
Podprogram **time** pobiera aktualny czas z zegara czasu rzeczywistego i umieszcza go w przekazanej zmiennej:

```
        .MODEL   SMALL, C
        PUBLIC   _time
_time   PROC
ARG     secs:PTR
        sub   ah, ah
       ; AH = 0 numer funkcji, która
                              ; odczytuje zegar
        int   1Ah             ; Wywołanie BIOS-u. Zwraca w
                              ; rejestrach CX i DX liczbę
                              ; taktów zegara, które upłynęły
                              ; od dnia 1 stycznia 1980 roku
        push  dx              ; Przekształcenie liczby taktów
                              ; na ilość sekund (65536 taktów
                              ; na jedną godzinę)
        mov   ax, cx          ; Pobierz liczbę godzin
        mov   cx, 3600        ; Liczba sekund w godzinie
        imul  cx              ; Przekształcona część godzinna
        mov   bx, [secs]      ; Pobierz adres argumentu
        mov   [bx], ax        ; Zapisz rezultat do przekazanej
                              ; zmiennej
        mov   [bx+2], dx
        pop   ax              ; Pobranie 1/65536 części godziny
        imul  cx              ; Przekształcenie na sekundy.
                              ; DX zawiera pełną liczbę sekund,
                              ; AX liczbę jej 1/65536 części
        add   [bx], dx        ; Dodanie sekund do
                              ; przekształconych już godzin
        adc   [bx+2], 0
        ret
_time   ENDP
```

Do pobrania aktualnego czasu używana jest funkcja BIOS-u (**AH=0** przerwanie **1Ah**). Jest on mierzony w taktach (w przybliżeniu 1/18 sekundy) od północy 1 stycznia 1980 roku. Funkcja najpierw dokonuje więc jego konwersji na liczbę sekund. Po jej wykonaniu pobierany jest adres argumentu (znajdujący się w **secs**) i kopiowany do BX. Na koniec wyliczona wartość jest przesyłana pod adres, znajdujący się w BX.
Trzecim sposobem zwracania danych jest umieszczenie ich w rejestrach procesora. Oczywiście w tym wypadku, zarówno funkcja wywołująca, jak i wywoływana, muszą używać tych samych rejestrów. W większości systemów 16-bitowe wartości całkowite zwracane są w rejestrze AX, a 32-bitowe w parze DX i AX. W ten sposób zwracane mogą być również wskaźniki, jednak użyte do tego rejestry mogą być różne na różnych kompilatorach.

Wiele języków może zwracać całe struktury danych. Na przykład, poniższa funkcja napisana w pseudokodzie, wypełnia strukturę **MyStru**, wywołując funkcję **StructFunc**:

```
Begin   function
     Define structure MyStru
        define integer IVar
        define character pointer Buffer
        define long integer Sample
     End of structure
     Set MyStru equal to the result of StructFunc
End of function
```

Niestety nie ma tylu rejestrów, by cała struktura mogła być w nich zwrócona. Funkcja może więc umieścić ją w swoim lokalnym obszarze pamięci, a następnie zwrócić do niej wskaźnik. Program, który wywołuje kopiuje wtedy dane z lokalnego obszaru funkcji do swej własnej struktury, co jest równoważne z pobraniem rezultatu funkcji. Asemblerowa wersja takiej funkcji może wyglądać następująco:

```
_Func   PROC
        call  StructFunc   ; Wywołanie funkcji
        push  ds           ; Skopiowanie rejestru DS do ES
        pop   es
        mov   si, ax       ; Wskaźnik do struktury zostanie
                           ; zwrócony w AX
        lea   di, [bp-8]   ; Pobranie adresu struktury
        mov   cx, 8        ; Rozmiar struktury
        rep   movsb        ; Skopiowanie danych
        ret
_Func   ENDP
```

Nazwy funkcji

W poprzednim przykładzie można zauważyć, że nazwy funkcji, używane w języku asemblera, nie są tymi samymi nazwami, które występują w języku C. Wynika to stąd, że kompilator języka C przed nazwą każdego podprogramu umieszcza znak podkreślenia (_). Dzięki temu funkcja, napisana przez programistę, nie będzie miała takiej samej nazwy jak podprogramy, które występują w wewnętrznej bibliotece kompilatora. W trybie QUIRKS oraz gdy w dyrektywie MODEL zostanie wyspecyfikowany język C, Turbo Assembler automatycznie doda ten przedrostek przed każdą nazwą funkcji. Jeśli kompilator wymaga wystąpienia podkreślenia, a ono nie wystąpiło, konsolidator wygeneruje błąd braku definicji symbolu.

Współdzielenie danych globalnych

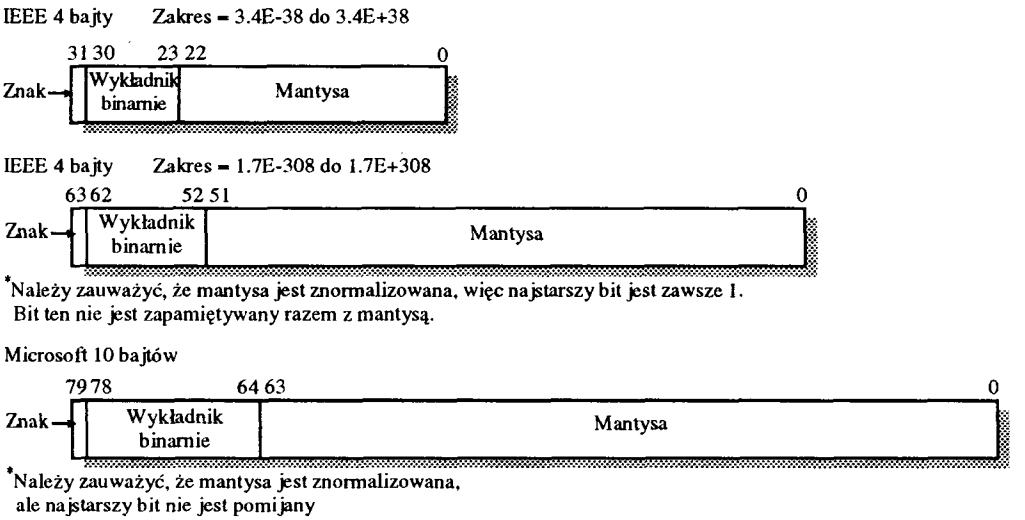
Zmienne globalne mogą być wykorzystywane zarówno w modułach asemblerowych, jak i modułach języków wysokiego poziomu. Moduł, który definiuje dane globalne, powinien użyć wtedy odpowiednika asemblerowej dyrektywy **PUBLIC**. Moduł, który używa tych danych, musi natomiast zastosować ekwiwalent dyrektywy **EXTRN**. Dyrektywy te informują konsolidator, że podany symbol będzie współdzielony przez kilka modułów. Nato-

miast do programisty należy zapewnienie zgodności typów danych oraz ich dostępności z segmentów modułu.

Zgodność typów danych

Konsolidator nie sprawdza zgodności typów danych globalnych. Jeśli w jednym z modułów dana została zadeklarowana jako 16-bitowa liczba całkowita, a w drugim jako dana 32-bitowa, to druga funkcja może spowodować zamazanie innych danych. Tego typu błędy są bardzo trudne do wykrycia.

W dokumentacji każdego kompilatora można znaleźć informację o formatach istniejących w nim typów danych. W większości z nich znaki są przechowywane na jednym bajcie, liczby całkowite w słowie, a długie liczby całkowite na dwóch słowach. Wskaźniki zależnie od modelu pamięci mogą być reprezentowane na słowach lub podwójnych słowach. Bardziej złożony problem stanowią liczby rzeczywiste. W procesorach 80x86 nie ma bowiem wbudowanego typu zmiennoprzecinkowego. W wielu językach zastosowano format, który odpowiada formatowi liczb w koprocesorze 80x87. Niektóre zaś, jak np. te pochodzące z firmy Microsoft, definiują swój własny sposób reprezentacji liczb rzeczywistych. Na rysunku 12-3 przedstawiono formaty dwóch różnych typów zmiennoprzecinkowych.



Rysunek 12-3. ***Formaty zmiennoprzecinkowe.***

Używając dyrektywy **STRUCT**, można w języku asemblera definiować struktury, które odpowiadają strukturom w językach wysokiego poziomu. Pewnym problemem, z którego należy zdawać sobie sprawę jest rodzaj rozmieszczania danych w pamięci. W wielu językach wysokiego poziomu struktury są rozmieszczane od parzystych adresów. Inne mogą jednak stosować odmienne typy. Dokładna informacja na ten temat powinna znajdować się w dokumentacji kompilatora. Stosując odpowiednie dyrektywy **ALIGN**, można więc dopasować się do jego wymagań.

Wybór segmentu

Aby odwołać się do danej, należy znać nazwę segmentu, w którym się ona znajduje. Dobrze jest również znać aktualne wartości rejestrów segmentowych. Większość nowoczesnych kompilatorów stosuje nazwy segmentów polecane przez firmę Microsoft. Tabela 12-1 pokazuje nazwy segmentów oraz ich zastosowanie. Nazwy te odpowiadają nazwom segmentów, tworzonych segmentowymi dyrektywami uproszczonymi.

Tabela 12-1. Powszechnie stosowane nazwy segmentów.

Segment	Grupa	Zastosowanie
_TEXT	*	W małych modelach pamięci w segmencie tym umieszczany jest cały kod programu.
name_TEXT		Zawiera kod programu. Jego nazwa jest różna dla różnych plików źródłowych.
FAR_DATA		Zainicjowane zmienne dalekie. Segment ten jest zadeklarowany jako PRIVATE. Każdy moduł może więc posiadać oddzielny segment FAR_DATA.
FAR_BSS		Nie zainicjowane zmienne dalekie. Również i ten segment jest zadeklarowany jako PRIVATE.
_DATA	DGROUP	Zainicjowane dane typu NEAR.
CONST	DGROUP	Stałe. Najczęściej przechowuje adresy segmentowe dalekich danych.
_BSS	DGROUP	Nie zainicjowane dane typu NEAR.
STACK	DGROUP	Stos.

* w modelu TINY segment _TEXT wchodzi w skład grupy DGROUP.

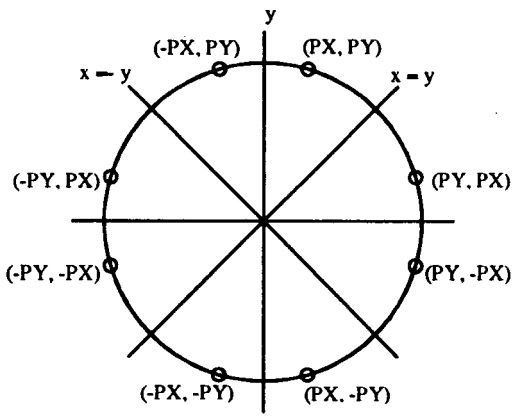
W tego typu rozmieszczeniu segmentów, rejestry segmentowe, wskazują pewne określone miejsca pamięci. Rejestry DS i SS wskazują grupę DGROUP, rejestr CS bieżący segment _TEXT. Rejestr ES pozostaje nie wykorzystany. Może on być użyty przy odwoływaniu się do danych dalekich (Więcej informacji o segmentach znajduje się w rozdziale 8).

Informacje dostarczane przez kompilator

Najwięcej informacji, które dotyczą współpracy procedur asemblerowych z programami napisanymi w innym języku, może dostarczyć dokumentacja dotycząca jego kompilatora. Pomóc może również on sam. W większości kompilatorów istnieje polecenie, dzięki któremu zamiast pliku wynikowego można uzyskać, odpowiadający mu plik asemblerowy. Dzięki temu można obejrzeć rozplanowanie poszczególnych instrukcji.
Pierwszym krokiem jest napisanie procedury w języku wysokiego poziomu odpowiadającej zadaniu, które ma być wykonane przez podprogram asemblerowy. Jeśli nie jest to możliwe, wówczas można stworzyć procedurę, która wykonywałaby zadanie podobne. Należy również użyć w niej nazw zmiennych, które rzeczywiście będą później wykorzystane. Procedura powinna mieć takie same argumenty oraz zwracać tę samą wartość jak oryginalna

funkcja asemblerowa. Jeśli potrzebne będą struktury czy zmienne globalne, należy je również zdefiniować.
Gdy już ten odpowiednik asemblerowej funkcji będzie możliwie bliski oryginałowi, to za pomocą kompilatora można stworzyć asemblerową postać napisanego modułu. Teraz istnieje już szablon funkcji, napisany w języku asemblera. Pozostało już tylko zamienić kod, wygenerowany przez kompilator żądanym kodem.
Sposób ten zademonstrowano w następnym przykładzie. Potrzebna jest funkcja, która rysuje ne ekranie okrąg tak szybko, jak to tylko możliwe. Zastosowany zostanie więc *algorytm Bresenhama*. Jest on bardzo szybki, a ponadto nie wymaga żadnych zmiennoprzecinkowych obliczeń. Sposób działania tego algorytmu przedstawiono na rysunkach 12-4 i 12-5. Z rysunku 12-4 widać, że do narysowania całego okręgu potrzebne jest tylko wyliczenie 1/8 jego punktów. Reszta może powstać przez proste transformacje. Rysunek 12-5 pokazuje 1/8 część koła i sposób jej rysowania. Zauważmy, że poruszając się wzdłuż jego krawędzi okręgu z lewej do prawej, wartość współrzędnej y dla każdego punktu jest taka sama lub o jeden mniejsza niż wartość w poprzednim punkcie. Bresenham odkrył bardzo prostą formułę, która pozwala ustalić miejsce zmiany wartości y. W programie o zmianie wartości y decyduje wartość zmiennej **delta**:

```
/* rysowanie kół */
#include <stdio.h>
#include <graph.h>
/* argumentami są współrzędne środka okręgu x,y, */
/* jego promień oraz kolor */
void
circle ( int cenx, int ceny, int radius, int colr )
{
   int x, y, delta;
   _setcolor( colr );
   y = radius;
   delta = 3 - 2 * radius;
```

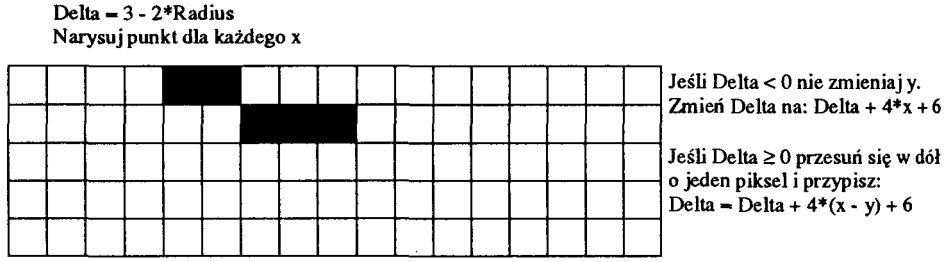


Rysunek 12-4. ***Rysowanie okręgu w ośmiu segmentach.***

```
   for( x=0; x<y; x++)
   {
/* narysuj punkt we wszystkich 8 częściach okręgu */
      _setpixel( x+cenx, y+ceny );
      _setpixel( x+cenx, ceny-y );
      _setpixel( cenx-x, ceny-y );
      _setpixel( cenx-x, y+ceny );
      _setpixel( y+cenx, x+ceny );
      _setpixel( y+cenx, ceny-x );
      _setpixel( cenx-y, ceny-x );
      _setpixel( cenx-y, x+ceny );
/* czy powinna zostać zmieniona wartość y */
      if( delta < 0 )
          delta += 4 * x + 6;
      else
      {
         delta += 4 * ( x - y ) + 10;
         y-=1;
      }
   }
/* narysuj ostatni punkt */
   if( y )
   {
      _setpixel( y+cenx, y+ceny );
      _setpixel( y+cenx, ceny-y );
      _setpixel( cenx-y, ceny-y );
      _setpixel( cenx-y, y+ceny );
   }
}
```

Po podaniu opcji "/Fa" kompilator języka C firmy Microsoft wygeneruje asemblerową wersję tej funkcji. Rezultat ten został przedstawiony poniżej (należy jednak pamiętać, że różne wersje kompilatora mogą wygenerować różne wersje):

```
; Static Name Aliases
;
```



Rysunek 12-5. ***Rysowanie segmentu okręgu algorytmem Bresenhama.***

```
        TITLE   dcirc.asm
        NAME    dcirc
        .8087
_TEXT       SEGMENT WORD PUBLIC 'CODE'
_TEXT       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
_DATA       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
CONST       ENDS
_BSS        SEGMENT WORD PUBLIC 'BSS'
_BSS        ENDS
DGROUP      GROUP   CONST,_DATA,_BSS
        ASSUME   CS:_TEXT,DS:DGROUP,SS:DGROUP
EXTRN       __setpixel:FAR
EXTRN       __setcolor:FAR
_TEXT       SEGMENT
        ASSUME  CS:_TEXT
;Line  10
        PUBLIC  _circle
_circle   PROC  NEAR
        push  bp
        mov   bp,sp
        sub   sp,24
        push  si
        push  di
        push  WORD PTR [bp+10]    ; colr
        call  FAR PTR __setcolor
        add   sp, 2
        mov   ax,WORD PTR [bp+8]
        mov   WORD PTR [bp-4],ax
        shl   ax,1
        sub   ax, 3
        neg   ax
        mov   WORD PTR [bp-6],ax ; delta
        mov   WORD PTR [bp-2],0  ; x
        mov   WORD PTR [bp-4],0  ; y
        jg    $JCC44
        jmp   $FB268
$JCC44:
        mov   WORD PTR [bp-8],6
        mov   di, WORD PTR [bp-4]; y
        mov   si, WORD PTR [bp-2]; x
$L20000:
        mov   ax,si
        add   ax,WORD PTR [bp+4] ; cenx
        mov   WORD PTR [bp-10],ax
```

```
mov   ax,di
add   ax,WORD PTR [bp+6] ; ceny
mov   WORD PTR [bp-12],ax
push  ax
push  WORD PTR [bp-10]
call  FAR PTR __setpixel
add   sp,4
mov   ax,WORD PTR [bp+6] ; cenx
sub   ax,di
mov   WORD PTR [bp-14],ax
push  ax
push  WORD PTR [bp-10]
call  FAR PTR __setpixel
add   sp, 4
mov   ax,WORD PTR [bp+4] ; cenx
sub   ax,si
mov   WORD PTR [bp-16], ax
push  WORD PTR [bp-14]
push  ax
call  FAR PTR __setpixel
add   sp, 4
push  WORD PTR [bp-12]
push  WORD PTR [bp-16]
call  FAR PTR __setpixel
add   sp, 4
mov   ax,di
add   ax,WORD PTR [bp+4]  ; cenx
mov   WORD PTR [bp-18],ax
mov   ax, si
add   ax, WORD PTR [bp+6] ; ceny
mov   WORD PTR [bp-20],ax
push  ax
push  WORD PTR [bp-18]
call  FAR PTR __setpixel
add   sp, 4
mov   ax,WORD PTR [bp+6]  ; ceny
sub   ax,si
mov   WORD PTR [bp-22],ax
push  ax
push  WORD PTR [bp-18]
call  FAR PTR __setpixel
add   sp, 4
mov   ax,WORD PTR [bp+4]  ; cenx
sub   ax,di
mov   WORD PTR [bp-24],ax
push  WORD PTR [bp-22]
```

```
        push  ax
        call  FAR PTR __setpixel
        add   sp, 4
        push  WORD PTR [bp-20]
        push  WORD PTR [bp-24]
        call  FAR PTR __setpixel
        add   sp, 4
        cmp   WORD PTR [bp-6],0    ; delta
        jge   $I269
        mov   ax, WORD PTR [bp-8]
        add   WORD PTR [bp-2],ax   ; delta
$I269:
        mov   ax,si
        sub   ax, di
        shl   ax,1
        shl   ax,1
        add   ax,10
        add   WORD PTR [bp-6],ax   ; delta
        dec   di
$I270:
        add   WORD PTR [bp-8],4
        inc   si
        cmp   si,di
        jge   $JCC260
        jmp   $L20000
$JCC260:
        mov   WORD PTR [bp-4],di   ; y
        mov   WORD PTR [bp-2],si   ; x
$FB268:
        cmp   WORD PTR [bp-4],0    ; y
        je    $I271
        mov   si, WORD PTR [bp-4]  ; y
        add   si, WORD PTR [bp+4]  ; cenx
        mov   di, WORD PTR [bp-4]  ; y
        add   di, WORD PTR [bp+6]  ; ceny
        push  di
        push  si
        call  FAR PTR __setpixel
        add   sp, 4
        mov   ax,WORD PTR [bp+6]   ; ceny
        sub   ax,WORD PTR [bp-4]   ; y
        mov   WORD PTR [bp-24],ax
        push  ax
        push  si
        call  FAR PTR __setpixel
        add   sp, 4
```

```
        mov     si,WORD PTR [bp+4]   ; cenx
        sub     si, WORD PTR [bp-4] ; y
        push    WORD PTR [bp-24]
        push    si
        call    FAR PTR __setpixel
        add     sp, 4
        push    di
        push    si
        call    FAR PTR __setpixel
        add     sp, 4
$I271:
        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret
_circle ENDP
_TEXT   ENDS
END
```

Na początku programu pojawił się komentarz, który zaczyna się od słowa "Linia". Oznacza on, że od tej właśnie linii pochodzi kod z pliku źródłowego. Kompilator umieszcza podobne komentarze w celu łatwiejszego zlokalizowania części zbioru źródłowego, który odpowiada poszczególnym fragmentom kodu asemblerowego.

Oczywiście kod ten może być w wielu miejscach zoptymalizowany. Można na przykład, zmienne x i y na początku funkcji umieścić w rejestrach. W tym wypadku x znajduje się w rejestrze SI, i w DI. Następnie można również zmniejszyć ilość operacji na stosie. W pętli głównej wywoływana jest tylko jedna funkcja: **__setpixel**. Można zaoszczędzić trochę czasu, przydzielając jej argumentom na stałe miejsce na stosie, a potem po prostu modyfikując wartości na stosie zamiast odkładania i pobierania ich z niego przy każdym wywołaniu. Ponieważ w większości wypadków zmieniany jest tylko jeden argument, to można w ten sposób uzyskać znaczne przyspieszenie programu.

Jeszcze trochę oszczędności można osiągnąć, analizując funkcję **__setpixel**. Można np. przekazać jej argumenty w rejestrach procesora. Ponieważ jednak jest ona jedną ze standardowych funkcji języka Microsoft C, dlatego rozwiązanie to nie wydaje się zbyt praktyczne.

Po dokonaniu modyfikacji moduł będzie wyglądał następująco:

```
; Static Name Aliases
;
     TITLE  dcirc.asm
     NAME   dcirc
     .8087
_TEXT   SEGMENT WORD PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT WORD PUBLIC 'CONST'
```

```
CONST   ENDS
_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP CONST,_DATA,_BSS
     ASSUME  CS:_TEXT,DS:DGROUP,SS:DGROUP
EXTRN   __setpixel:FAR
EXTRN   __setcolor:FAR
_TEXT   SEGMENT
     ASSUME  CS:_TEXT
     PUBLIC  _circle
_circle   PROC NEAR
     push  bp
     mov   bp,sp
     sub   sp,2                  ; rezerwacja miejsca
                                 ; na zmienną delta
     push  si
     push  di
     sub   sp, 4                 ; rezerwacja miejsca
                                 ; na argumenty funkcji
                                 ; _setpixel
     push  WORD PTR [bp+10]      ; colr
     call  FAR PTR __setcolor
     add   sp, 2
     mov   di,WORD PTR [bp+8]    ; y = promień
     mov   ax, di
     shl   ax,1                  ; promień * 2
     sub   ax, 3; - 3
     neg   ax
                   ; zmiana znaku
     mov   WORD PTR [bp-2],ax    ; delta
$L20002:
     mov   ax,di
     add   ax,WORD PTR [bp+6]    ; y + ceny
     mov   WORD PTR [bp-8],ax    ; drugi argument
     mov   ax,si
     add   ax,WORD PTR [bp+4]    ; x + cenx
     mov   WORD PTR [bp-10],ax   ; pierwszy argument
     call  FAR PTR __setpixel
     mov   ax,WORD PTR [bp+6]    ; ceny
     sub   ax,di; ceny - y
     mov   WORD PTR [bp-8],ax
     call  FAR PTR __setpixel
     mov   ax,WORD PTR [bp+4]    ; cenx - x
     sub   ax,si
     mov   WORD PTR [bp-10], ax
     call  FAR PTR __setpixel
```

```
        mov     ax,WORD PTR [bp+6]    ; ceny + y
        add     ax,di
        mov     WORD PTR [bp-8],ax
        call    FAR PTR __setpixel
        mov     ax, WORD PTR [bp+4]
        add     ax, di; cenx + y
        mov     WORD PTR [bp-10], ax
        mov     ax,WORD PTR [bp+6]    ; ceny + x
        add     ax,si
        mov     WORD PTR [bp-8], ax
        call    FAR PTR __setpixel
        mov     ax,WORD PTR [bp+6]    ; ceny - x
        call    FAR PTR __setpixel
        mov     ax,WORD PTR [bp+4]    ; cenx - y
        sub     ax, di
        mov     WORD PTR [bp-10],ax
        call    FAR PTR __setpixel
        mov     ax,WORD PTR [bp+6]    ; ceny + x
        add     ax, si
        mov     WORD PTR [bp-8],ax
        call    FAR PTR __setpixel
        cmp     WORD PTR [bp-2],0     ; delta
        jge     $I439
        mov     ax,si
        shl     ax,1
        shl     ax,1
        add     ax,6
        add     WORD PTR [bp-2],ax    ; delta
        jmp     $I440
$I439:
        mov     ax,si
        sub     ax,di
        shl     ax,1
        shl     ax,1
        add     ax,10
        add     WORD PTR [bp-2],ax    ; delta
        dec     di
$I440:
        inc     si
        cmp     si,di
        jge     $JCC928
        jmp     $L20002
$JCC928:
        or      di,di
        je      $I441
        mov     ax, WORD PTR [bp+4]
```

```
        add     ax, di                  ; cenx + y
        mov     WORD PTR [bp-10], ax    ; pierwszy argument
        mov     si, WORD PTR [bp+6]
        add     si, di
        mov     WORD PTR [bp-8], si     ; drugi argument
        call    FAR PTR __setpixel
        mov     ax,WORD PTR [bp+6]      ; ceny - y
        sub     ax, di
        mov     WORD PTR [bp-8],ax
        call    FAR PTR __setpixel
        mov     ax,WORD PTR [bp+4]
        sub     ax, di
        mov     WORD PTR [bp-10],ax
        call    FAR PTR __setpixel
        mov     WORD PTR [bp-8], si
        call    FAR PTR __setpixel
$I441:
        add     sp, 4
        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret
_circle ENDP
_TEXT   ENDS
END
```

W tym programie wartości zmiennych x i y są przechowywane w rejestrach SI i DI, a nie jak poprzednio w komórkach pamięci. Jednak najbardziej znacząca modyfikacja polega na zmianie sposobu wykorzystania ramki stosu. Przed wywołaniem funkcji **__setcolor** na stosie jest rezerwowane miejsce dla dwóch argumentów procedury **__setpixel**. Funkcja ta jest następnie wywoływana od etykiety $L20002. Jednak zamiast przesyłania i zdejmowania jej argumentów ze stosu, funkcja **_circle** bezpośrednio na nim modyfikuje wartości tych parametrów.

Moduły startowe

Na początku programu, napisanego w języku wysokiego poziomu, wykonywane są pewne procedury, które inicjują środowisko pracy programu. Czynności te nie są dla każdej aplikacji konieczne. W niektórych systemach jednak wymagają one nawet do 50K kodu. W programach, które potrzebują dużo pamięci, można je zastąpić modułami, które wykonują tylko niezbędne czynności.

Poniższy przykład jest absolutnie minimalnym modułem startowym dla języka Microsoft C. Program, który korzysta z innych możliwości - takich jak wersja systemu DOS czy sterta - które zostały usunięte, może po prostu wywołać odpowiednie procedury. Dzięki temu programy mogą zawierać tylko rzeczywiście niezbędny kod.

```
; Moduł startowy dla języka Microsoft C
     DOSSEG
     .MODEL   SMALL
extrn    _main:PROC
PUBLIC   __acrtused
__acrtused = 9876h
        .CODE
Start   PROC
        mov   ax, DGROUP  ; Ustawienie segmentu danych
        mov   ds, ax
        assume  ds:DGROUP, ss:DGROUP
        call  _main       ; Wywołanie programu w języku C
        mov   ah, 4Ch     ; Wyjście do systemu
        int   21h
Start   ENDP
        .STACK  1024
END     Start
```

Przykładowy program generujący dźwięki

Często jedynym sposobem na bezpośredni dostęp do sprzętu jest napisanie odpowiedniej procedury w języku asemblera. W poniższym przykładzie została przedstawiona procedura, która generuje dźwięki na komputerach zgodnych z IBM PC. Może być ona wywoływana z programów pisanych w językach wysokiego poziomu. Do wygenerowania dźwięku o określonej częstotliwości i o określonym czasie trwania, używa ona układu czasowego. W komputerach zgodnych z IBM PC istnieją trzy układy czasowe. Pierwszy z nich używany jest przez funkcje BIOS-u, które podają aktualną datę i czas. Generuje on przerwanie 18.2 razy na sekundę. BIOS zlicza te przerwania w czterobajtowej zmiennej o adresie **40:6C**. Natomiast w drugim układzie czasowym można zaprogramować czas pomiędzy kolejnymi przerwaniami. Jego wyjście może być więc podane na głośnik, w celu wygenerowania dźwięku o określonej częstotliwości. Trzeci układ jest używany przy odświeżaniu pamięci RAM, a w przykładzie nie został on wykorzystany.
Argumentami funkcji **Play** są nuta i czas trwania dźwięku. Nuta określa częstotliwość dźwięku. Parametr ten może przyjmować wartości z zakresu od 0 do 13. Wartość 0 oznacza wyłączenie głośnika, jeden oznacza środkowe C, dwa podwyższone C, trzy D itd. Czas trwania podawany jest w taktach (65,536 taktów na godzinę, około 18.2 taktu na sekundę). W poniższym programie do odtworzenia melodii została użyta właśnie procedura **Play**:

```
struct {
       int note;
       int duration;
} Song[] =
{ {5, 9}, {3, 9}, {1, 9}, {3, 9}, {5, 9}, {5, 9}, {5, 9},
  {3, 9}, {3, 9}, {3, 9}, {5, 9}, {8, 9}, {8, 9}, {5, 9},
  {3, 9}, {1, 9}, {3, 9}, {5, 9}, {5, 9}, {5, 9}, {5, 9},
  {3, 9}, {3, 9}, {5, 9}, {3, 9}, {1, 18}, {-1, -1} };
```

```
main()
{
    for( int i=0; Song[i].note != -1; i++ )
          Play( Song[i].note, Song[i].duration );
}
```

Funkcja **Play** przedstawiona poniżej, odszukuje w tablicy, odpowiadający podanej nucie, przedział czasu dla układu czasowego. Następnie po zaprogramowaniu tego układu, dodaje ona zawartość licznika taktów o adresie **40:6C** do czasu trwania dźwięku. Będzie on tak długo generowany, aż wartość licznika przekroczy tak obliczoną wartość. Następnie głośnik zostaje wyłączony i następuje powrót do programu głównego.

```
; Program muzyczny
; z programu w języku C funkcja powinna zostać wywołana
; w następujący sposób:
; Play( Nuta, Czas_trwania );
; Nuta równa 0 oznacza wyłączenie głośnika. Czas trwania
; podany jest w taktach (18.2 na sekundę).
     .MODEL SMALL, C
     .DATA
; w tablicy znajdują się wartości liczników taktów,
; które odpowiadają nutom od średniego C do
; wysokiego C. Każda nuta jest o połowę tonu
; wyższa niż poprzednia.
PUBLIC  _Play
NoteTab  DW  0, 4561, 4305, 4063, 3835, 3620, 3417, 3225,
3044
         DW       2873, 2712, 2560, 2416, 2280
     .CODE
_Play    PROC
ARG      Note:WORD, Duration:WORD
     mov   bx, [Note]       ; Pobierz nutę
     shl   bx, 1            ; Pomnożenie przez dwa ze
                            ; względu na rozmiar elementu
                            ; tablicy
     jz    Delay            ; Jeśli jest to 0, odczekanie
                            ; tylko zadanego czasu
     mov   al, 0B6h
     out   43h, al          ; Przygotowanie portu zegara
     mov   al, BYTE PTR NoteTab[bx]     ; Pobranie młodszego
                                        ; bajtu licznika
     out   42h, al
     mov   al, BYTE PTR NoteTab+1[bx]   ; Pobranie starszego
                                        ; bajtu licznika
     out   42h, al
     in    al, 61h          ; Pobranie maski
     or    al, 3            ; Ustawienie bitów głośnika
```

```
     out    61h, al          ; Wysłanie nowej maski
Delay:
     push   ds               ; Zachowanie rejestru segmentu
                             ; danych
     mov    ax, 40h          ; DS wskazuje na obszar
                             ; danych BIOS-u
     mov    ds, ax
     mov    bx, ds:[6Ch]     ; Pobierz młodsze słowo
                             ; licznika taktów
     mov    cx, ds:[6Eh]     ; Pobierz starsze słowo
     add    bx, [Duration]   ; Dodanie czasu trwania dźwięku
     adc    cx, 0            ; Dodanie przeniesienia
                             ; do starszego słowa
Ploop:                       ; Oczekiwanie na wartość
                             ; licznika taktów
     cmp    ds:[6Eh], cx     ; odpowiadającą zawartości
                             ; rejestrów cx:bx
     jl     Ploop
     cmp    ds:[6Ch], bx
     jl     Ploop
     in     al, 61h          ; Pobranie maski
     and    al, 0FCh         ; Wyzerowanie bitów głośnika
     out    61h, al          ; Wysłanie nowej maski
     pop    ds
     ret
_Play ENDP
END
```

Podsumowanie

Każdy język komputerowy ma takie zastosowanie, do którego się najbardziej nadaje. Język asemblera doskonale nadaje się do pisania programów, które muszą bezpośrednio operować na sprzęcie. Stosuje się go również przy tworzeniu fragmentów kodu, które muszą działać tak szybko jak to jest tylko możliwe. Natomiast języki wysokiego poziomu są łatwiejsze i do zrozumienia, i do wykorzystania. Najlepszym więc rozwiązaniem jest połączenie w programie zalet ich obu i zastosowanie właściwego języka do właściwego problemu.

Turbo Assembler posiada kilka komend, które czynią ten proces łatwiejszym. Dyrektywa **.MODEL** pozwala na zorganizowanie segmentów i wybranie określonej konwencji wywoływania podprogramów tak, by moduł asemblerowy mógł być wykorzystany w językach wysokiego poziomu. Gdy w dyrektywie **.MODEL** zostanie podany specyfikator języka, to asembler potrafi automatycznie wygenerować odpowiedni kod, który przekazuje funkcjom argumenty, a ponadto pozwala na nadawanie im nazw symbolicznych.

Wiele informacji, które dotyczą współpracy modułów asemblerowych z modułami powstałymi w innych językach, może dostarczyć sam kompilator. Bowiem większość z nich potrafi wygenerować ze zbioru źródłowego odpowiadający mu zbiór asemblerowy. Dzięki temu, analizując je, można dowiedzieć się, jak współpracować z danym kompilatorem.

Rozdział 13 Wykorzystanie zasobów systemu

Komputery zgodne z IBM PC posiadają wiele zasobów, które mogą być wykorzystywane w programach. Zasoby te to pamięć, koprocesory, stacje dysków, porty szeregowe i równoległe, generowanie dźwięku, klawiatura, mysz czy karta graficzna. Są one dostępne poprzez specjalne sterowniki, BIOS lub MS-DOS.
Oczywiście nie każdy komputer posiada wszystkie wyżej wymienione zasoby. Niektóre mogą mieć zainstalowane jeszcze inne urządzenia. W wypadku dysków, czy pamięci, różne systemy mogą różnić się między sobą rozmiarem pamięci lub liczbą dysków. W pierwszej części rozdziału przedstawione zostały metody identyfikowania zasobów w systemie.
Pewne zasoby są do działania programu niezbędne. Na przykład, musi on mieć wystarczającą ilość wolnej pamięci. Inne, jak np. mysz, mogą być dla niego opcjonalne. Jednakże w obu wypadkach program musi mieć procedury, które zarządzają tymi zasobami. Druga część rozdziału pokazuje sposoby wykorzystania pamięci, dysków, portów szeregowych i równoległych, klawiatury, myszy i karty graficznej.

Identyfikacja zasobów

Przed wykorzystaniem zasobów konieczne jest sprawdzenie ich dostępności. Programy, napisane w Turbo Assemblerze, mogą działać na wielu komputerach, które mogą być przecież różnie skonfigurowane. Niestety, informacja o tym jest przechowywana w wielu różnych miejscach. Potrzeba użyć wielu sposobów, by uzyskać pełną informację o systemie.

Usługi i dane BIOS-u

Powodem wprowadzenia programu BIOS do komputera jest zapewnienie standardowej komunikacji ze sprzętem na wszystkich maszynach. Część tego systemu ma za zadanie udostępniać informację o istniejących zasobach. Informacja ta może być dostępna w dwojaki sposób: pierwszy to przerwania programowe, drugi to bezpośredni odczyt danych z obszaru BIOS-u.

FUNKCJE BIOS-U

BIOS wykorzystuje przerwania o numerach od 10h do 1Ah. Przerwanie 5, które przesyła zawartość ekranu na drukarkę, jest również jego częścią. Z każdym przerwaniem związanych jest kilka funkcji. Wybierane są one poprzez umieszczenie numeru żądanej funkcji w rejestrze AH. Tabela 13-1 zawiera funkcje BIOS-u, które mogą być wykorzystane do testowania konfiguracji komputera.

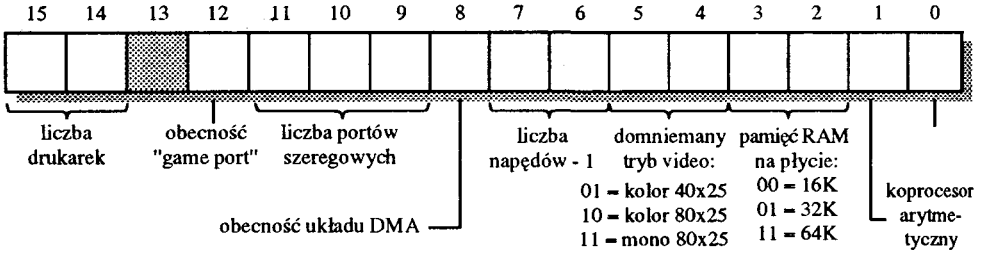
Tabela 13-1. Funkcje BIOS-u, które służą do ustalania konfiguracji systemu.

Przerwanie	Funkcja	Opis
10h	0Fh	Pobranie trybu pracy karty graficznej
11h	-	Pobranie listy dostępnych urządzeń
12h	-	Pobranie rozmiaru pamięci podstawowej
13h	15h	Ustalenie typu dysku
15h	88h	Pobranie rozmiaru pamięci rozszerzonej
15h	0C0h	Pobranie parametrów konfiguracji systemu

Przerwanie BIOS-u o numerze **10h** dostarcza zestaw funkcji do obsługi karty graficznej. Funkcja o numerze **15h** pozwala na pobranie aktualnego trybu pracy karty. Jest ona potrzebna, gdy program bezpośrednio zapisuje znaki do pamięci video. W trybie tekstowym pamięć ta może zaczynać się od dwóch różnych adresów. Jeśli karta pracuje w trybie 7, to pamięć zaczyna się od adresu **0B000:0000**. W trybie 2 lub 3 jej początek wyznacza adres **0B800:0000**. (W rozdziale 14 można znaleźć więcej informacji, które dotyczą wykorzystania karty graficznej).

Przerwanie **11h** w rejestrze AX zwraca listę urządzeń zainstalowanych w systemie. Można więc ustalić liczbę istniejących w systemie drukarek, dysków i portów szeregowych. Można też sprawdzić obecność koprocesora, portu gier lub urządzenia wskaźnikowego. Rysunek 13-1 przedstawia format słowa, umieszczanego przez to przerwania w rejestrze AX.

Przerwanie **12h** zwraca w rejestrze AX rozmiar podstawowej pamięci, wyrażony w kilobajtach. W starszych komputerach wartość ta jest ustawiana na podstawie położenia przełączników na płycie głównej. Ponieważ w systemach tych można dodać nową pamięć bez zmieniania nastawów przełączników, to zwrócona wartość nie musi więc odpowiadać rozmiarowi fizycznie zainstalowanej pamięci. Komputery PC/AT, PS/2 i zgodne mogą również używać pamięci rozszerzonej. Funkcja o numerze **88h** przerwania **15h** zwraca jej rozmiar w rejestrze AX.



Rysunek 13-1. ***Lista urządzeń w systemie.***

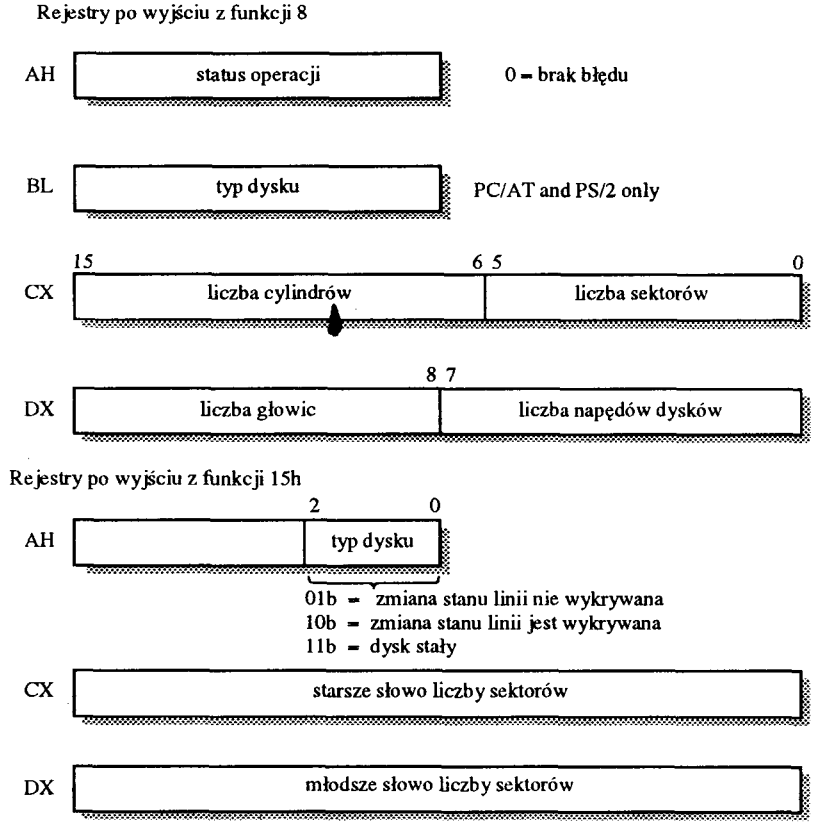
Funkcje **8h** i **15h** przerwania **13h** zwracają typ zainstalowanego dysku. Argumentem tych funkcji jest fizyczny numer dysku, podany w rejestrze DL. Numer ten musi mieścić się w zakresie od 0 do pomniejszonej o jeden liczby dysków w systemie. Ponadto, numer,

odnoszący się do dysku twardego, ma ustawiony najstarszy bit, a numer stacji dyskietek ma ten bit wyzerowany. Funkcja **8h** zwraca liczbę dysków, podłączonych do tego samego kontrolera, liczbę głowic dysku, cylindrów i sektorów na ścieżce. Ponadto w komputerach PC/AT i PS/2 zwraca również typ stacji dyskietek. Funkcja **15h** informuje, czy w stacji wystąpiła zmiana dyskietki. Natomiast dla dysku twardego zwraca łączną liczbę jego sektorów. Rysunek 13-2 przedstawia rozmieszczenie tych danych w poszczególnych rejestrach.

Funkcja **0C0h** przerwania **15h** pozwala na komputerach PC/AT i PS/2 uzyskać więcej informacji o systemie. Zwraca ona w rejestrach ES:BX wskaźnik do obszaru, w którym umieszczone są parametry konfiguracji. Rysunek 13-3 przedstawia format tych danych.

OBSZAR DANYCH BIOS-U

Część informacji, które są zwracane przez funkcje BIOS-u, jest dostępna również w jego obszarze danych. Ponadto, zapisana jest tu również informacja, której poprzez te funkcje uzyskać się nie da. Na przykład, typ i podtyp komputera, pochodzące z parametrów konfiguracji komputera, jest przechowywane pod adresem **F000:FFFE**. Pod adresem **F000:FFF5** umieszczona jest data powstania programu BIOS.

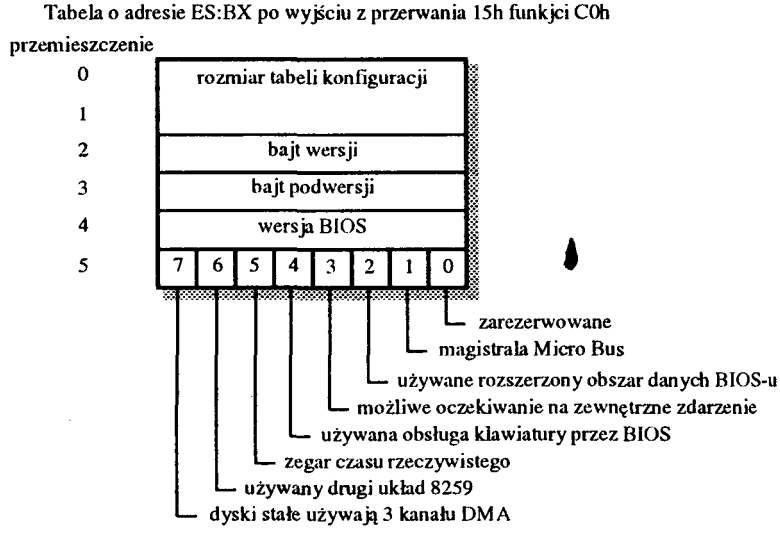


Rysunek 13-2. ***Informacja o dysku zwrócona przez funkcje 8 i 15h przerwania 13h.***

Większość użytecznych danych może być znaleziona w segmencie danych BIOS-u. Aby móc to wykorzystać, należy do rejestru segmentowego załadować wartość 40h, a następnie wszystkie odwołania powinny być realizowane poprzez ten właśnie rejestr. Adresy kilku interesujących danych tego obszaru umieszczone zostały w tabeli 13-2.

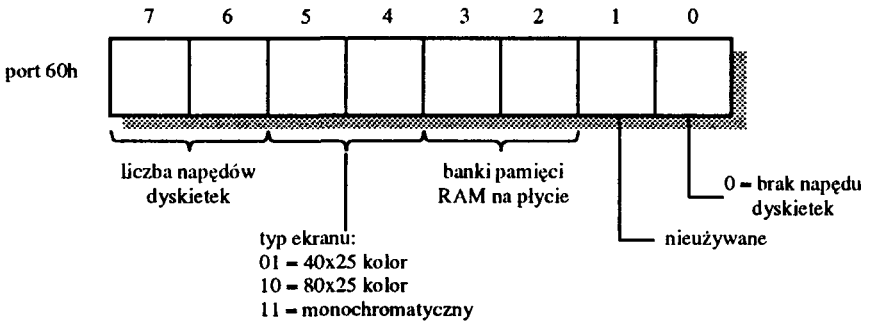
Korzystanie z informacji inicjujących

Wiele użytecznych informacji zawierają również dane inicjujące. W komputerach PC/XT pochodzą one z przełączników, które znajdują się na płycie głównej. Komputery PC/AT i PS/2 informację tę mają zapisaną w specjalnej pamięci RAM. Zawiera ona ilość zainstalowanej pamięci, rodzaj karty graficznej, ilość portów typu COM oraz rodzaj dysku. Znajdować się może tu więcej danych, zależnie od modelu komputera i jego producenta. Stan przełączników na komputerach PC/XT może być odczytany z portów o numerach **60h** i **61h**. Najstarszy bit portu 61h niesie informację o typie danych, umieszczonych w porcie 60h. Jeśli jest on ustawiony, to dane na porcie 60h pochodzą ze sterownika klawiatury. Gdy nie jest on równy zero, wówczas port 60h zawiera stan przełączników. Tak więc, by go odczytać, należy wyzerować ten bit, a następnie pobrać daną już z portu. Rysunek 13-4 pokazuje jej format.



Rysunek 13-3. ***Parametry konfiguracji systemu.***

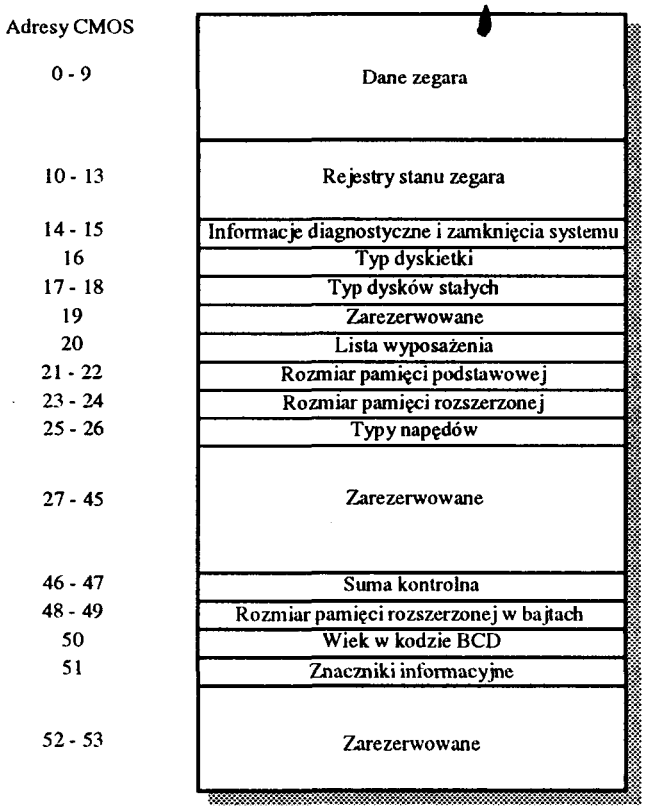
W komputerach PC/AT i PS/2 informacja konfigurująca umieszczona jest w specjalnej pamięci RAM, która znajduje się w układzie zegara czasu rzeczywistego (RTC). Układ RTC korzysta przy komunikacji z procesorem z dwóch portów (**70h** i **71h**). Port 70h jest portem indeksowym, który zawiera przemieszczenie żądanej danej w tej pamięci. Można ją pobrać lub zapisać do portu 71h. Poniższy przykład pokazuje, w jaki sposób z pamięci tej można pobrać aktualny rok:



Rysunek 13-4. ***Format danej na porcie 60h.***

```
cli                 ; Zablokowanie przerwań
mov   al, 9         ; Indeks bajtu, który przechowuje rok
out   70h, al       ; Wysłanie go do rejestru indeksowego
in    al, 71h       ; Pobranie do rejestru AL aktualnego roku
sti                 ; Włączenie przerwań
```

Rysunek 13-5 przedstawia informację, która znajduje się w pamięci RTC.



Rysunek 13-5. ***Informacja przechowywana w pamięci RTC.***

Programowe testowanie zasobów

Innym sposobem ustalania, jakie zasoby są dostępne w systemie, jest napisanie programu, który przeprowadzałby specjalne testy na ich obecność. W wielu wypadkach metoda ta okazuje się najbardziej wiarygodna. Prawdziwość informacji, które znajdują się w obszarze danych BIOS-u, gwarantowana jest tylko na prawdziwych komputerach IBM. Maszyny, które są tylko zgodne z nimi, mogą mieć pewne szczegóły inaczej zaimplementowane lub nawet może ich nie być w ogóle.

PAMIĘĆ

Większości programów wystarcza pamięć przydzielona przez DOS. Są jednak takie, które potrzebują całej dostępnej pamięci. Na przykład, program testujący pamięć powinien mieć dostęp do każdej komórki.

Tabela 13-2. Segment danych BIOS-u.

Przemieszczenie	Rozmiar w bajtach	Opis
0010h	2	Lista zainstalowanych urządzeń. Dana ta jest również zwracana przez przerwanie 11h.
0013h	2	Rozmiar pamięci. Dana ta jest również zwracana przez przerwanie 12h.
0049h	1	Bieżący tryb karty graficznej. Dana ta jest również zwracana przez funkcję 0Fh przerwania 10h.
004Ch	2	Rozmiar pamięci video.
0063h	2	Adres sterownika CRT.
0087h	4	Konfiguracja karty EGA.

Testowanie pamięci odbywa się poprzez zapisanie do niej pewnej wartości i następnie pobranie jej i porównanie z daną wyjściową. Jeśli dla kilku wartości otrzyma się dane prawidłowe, to testowana komórka uznawana jest za dobrą komórkę pamięci RAM. Jeśli jednak pobrana wartość jest inna niż zapisywana, wówczas oznaczać to może, że podany adres odnosi się do pamięci ROM, albo wskazywana komórka nie istnieje.
Oczywiście tego typu metoda powoduje zamazanie starej zawartości komórki. Jeśli dana nie powinna zostać zniszczona, to na czas przeprowadzania testu należy przechować ją w innym miejscu.
Poniższa funkcja sprawdza, czy komórka o adresie ES:BX jest komórką pamięci RAM:

```
isRAM   PROC
        mov   dx, es:[bx]    ; Pobranie zawartości komórki
        mov   es:[bx], 55h   ; Ustawienie co drugiego bitu
        cmp   es:[bx], 55h   ; Sprawdzenie poprawności danej
        jne   NotRAM
        mov   es:[bx], 0AAh  ; Zanegowanie bitów
        cmp   es:[bx], 0AAh
        je    RAM
```

```
NotRAM:
        sub     ax,ax           ; Komórka nie jest
                                ; w pamięci RAM
        jmp     SHORT Done
RAM:
        mov     ax, 1           ; Komórka jest w pamięci RAM
Done:
        mov     es:[bx], dx     ; Odtworzenie poprzedniej
                                ; zawartości pamięci
        ret
isRAM   ENDP
```

PORTY SZEREGOWE

Metoda wykrywania obecności portu szeregowego polega na wysłaniu do niego kilku rozkazów, a następnie porównaniu otrzymanych wyników z oczekiwanymi. Poniższy program testuje port szeregowy, przestawiając go w tryb z pętlą zwrotną i sprawdzając, czy znak wysłany jest taki sam jak odebrany.

```
IsPort  PROC
        mov     dx, 3FCh    ; Rejestr sterujący portu COM1
        in      al, dx      ; Pobranie z niego danej
        mov     bl, al      ; Zachowanie jej
        or      al, 10h     ; Ustawienie trybu z pętlą
                            ; zwrotną
        out     dx, al      ; Wysłanie komendy do układu UART
        mov     dx, 3F8h    ; Rejestr danych portu COM1
        mov     al, 55h
        out     dx, al      ; Wysłanie bajtu
        mov     cx, 5000h   ; Liczba obiegów pętli
        mov     dx, 3FDh    ; Rejestr statusowy portu COM1
Wait1:
        in      al, dx      ; Pobierz stan linii portu
        test    al, 1       ; Czy otrzymano daną
        jnz     ReadIt      ; Jeśli tak, odczytanie znaku
        loop    Wait1
        jmp     SHORT Noport
ReadIt:
        mov     dx, 3F8h    ; Rejestr danych portu COM1
        in      al, dx      ; Pobranie danej
        cmp     al, 55h     ; Porównanie go z daną wysłaną
        clc                 ; Oznaczenie obecności portu
                            ; szeregowego
        je      PortOK      ; Jeśli dane są zgodne, port
                            ; jest obecny
Noport:
        stc                 ; Zaznaczenie, że port nie
```

```
                                ; odpowiada
PortOK:
        mov     dx, 3FCh        ; Rejestr sterujący portu COM1
        mov     al, bl          ; Pobranie poprzedniego
                                ; stanu układu
        out     dx, al
        ret
IsPort  ENDP
```

W pierwszej części funkcji port COM1 jest przełączany w tryb pracy z pętlą zwrotną. W trybie tym każdy wysłany znak powinien pojawić się w rejestrze danych portu. W funkcji wysłany zostaje bajt **55h**. Właśnie w pętli, która zaczyna się od etykiety **Wait1**, realizowane jest oczekiwanie na pojawienie się wysłanego znaku. Następnie dana ta jest odczytywana i porównywana z wartością **55h**. Jeśli nastąpi przekroczenie czasu w pętli **Wait1** lub też zostanie odczytana inna wartość, to ustawiamy znacznik przeniesienia wskazując w ten sposób, że port nie został wykryty. Gdy zaś dane są identyczne, wówczas znacznik ten jest zerowany, a sterowanie jest przekazywane pod etykietę **PortOK**. W końcowym fragmencie funkcji przywracany jest poprzedni stan portu szeregowego i następuje powrót do programu wywołującego.

TYP PROCESORA

Każdy procesor rodziny 80x86 ma swoją cechę szczególną. Różnice między nimi mogą być wykorzystane do ustalenia typu zainstalowanego procesora. Na przykład, w procesorach 8088 i 8086 instrukcja przesuwania będzie wykonana również wtedy, gdy wartość rejestru CL jest większa lub równa 32. W jednostkach 80188 lub 80186 rozkaz ten zostanie zignorowany. W procesorach 80286 zostało natomiast zmienione działanie instrukcji **push sp**. W starszych wersjach na stos przesyłana jest niezmodyfikowana wartość wskaźnika stosu. Procesory 80286 i 80386 umieszczają wartość już po dekrementacji. Inna zmiana dokonana w procesorach 80286 i nowszych polega na wykorzystaniu przerwania 6 przy próbie wykonania przez CPU nielegalnej instrukcji. Dzięki temu rozkaz, który istnieje tylko dla 80386 na procesorze 80286 spowoduje wygenerowanie przerwania.
Poniższy program, wykorzystując przedstawione różnice, ustala typ procesora:

```
; Ustalenie typu procesora
          .MODEL   SMALL
          .DATA
          .386                  ; Udostępnienie instrukcji
                                ; procesora 80386
OldInt6   DD   ?
M88       DB   "To 8088/86", 0Dh, 0Ah, '$'
M186      DB   "To 80188/86", 0Dh, 0Ah, '$'
M286      DB   "To 80286", 0Dh, 0Ah, '$'
M386      DB   "To 80386", 0Dh, 0Ah, '$'
          .CODE
Start     PROC
          mov   ax, SEG DGROUP
          mov   ds, ax
```

```
; Czy jest to jeden z procesorów 8088/86 lub 80188/86
          mov   dx, sp        ; Zachowaj wartość wskaźnika
                              ; stosu
          push  sp
          pop   cx
          cmp   dx, cx        ; Wartości te są różne tylko
                              ; dla procesorów 8088/86
                              ; lub 80188/86
          je    Cpu1
; Czy to procesor 80186
          mov   ax, 1
          mov   cl, 32
          shl   ax, cl        ; Sprawdzenie rezultatu
                              ; przesuwania o 32 bity
          jz    Not8088       ; Rozkaz ten nie zostanie
                              ; wykonany przez procesor
                              ; 8088/86
          mov   dx, OFFSET M88
          jmp   pexit
Not8088:
          mov   dx, OFFSET M186
          jmp   pexit
; Czy to procesor 80386
          mov   ax, 3506h     ; Pobierz adres procedury
                              ; obsługi przerwania 6
          int   21h
          mov   WORD PTR OldInt6, bx
          mov   WORD PTR OldInt6+2, es
          push  ds
          push  cs
          pop   ds
          mov   dx, OFFSET Not 80386
          mov   ax, 2506h
          int   21h
          pop   ds
          inc   eax           ; Instrukcja procesora 80386
          mov   dx, OFFSET M386
          jmp   rexit         ; Przywrócenie starej procedury
                              ; obsługi przerwania 6,
                              ; wyświetlenie komunikatu
                              ; i wyjście z funkcji
Not386:
; Operacja na rejestrze 32-bitowym nie powiodła się
; musi to być procesor 80286
          pop   ax            ; Jest to procedura obsługi
                              ; przerwania 6.
```

```
        pop   ax            ; Instrukcje usuwają ze stosu
                            ; znaczniki oraz
        pop   ax            ; adres powrotu
        mov   dx, OFFSET M286
  ~zywrócenie starej procedury obsługi przerwania 6
  ':
        push  dx
        push  ds
        mov   dx, WORD PTR OldInt6
        mov   ds, WORD PTR OldInt6+2
        mov   ax, 2506h
        int   21h
        pop   ds
        pop   dx
pexit:
        mov   ah, 9
        int   21h
        mov   ah, 4Ch
        int   21h
Start   ENDP
        .STACK  1024
END     Start
```

Pierwszy test wykrywa obecność jednego z procesorów 8088, 8086, 80188 lub 80186. Wykorzystuje on różnice w działaniu instrukcji push sp. Procesory 8088, 8086, 80188 lub 80186 zmieniają wartość wskaźnika stosu przed przesłaniem jej na stos, podczas gdy pozostałe umieszczają na stosie wartość nie zmienioną. Następny test rozróżnia procesory 8088/86 od 80188/86. Wykorzystano w nim fakt, że procesor 8088/86 nie wykonuje instrukcji przesunięć o 32 lub więcej bitów. Ostatni test sprawdza, czy CPU jest procesorem 80286, czy 80386. Procesory te generują bowiem przerwanie 6 przy próbie wykonania nielegalnej instrukcji. Rozkaz **inc eax** jest na procesorze 80386 poprawny, ale 80286 wygeneruje przerwanie przy próbie jej wykonania.

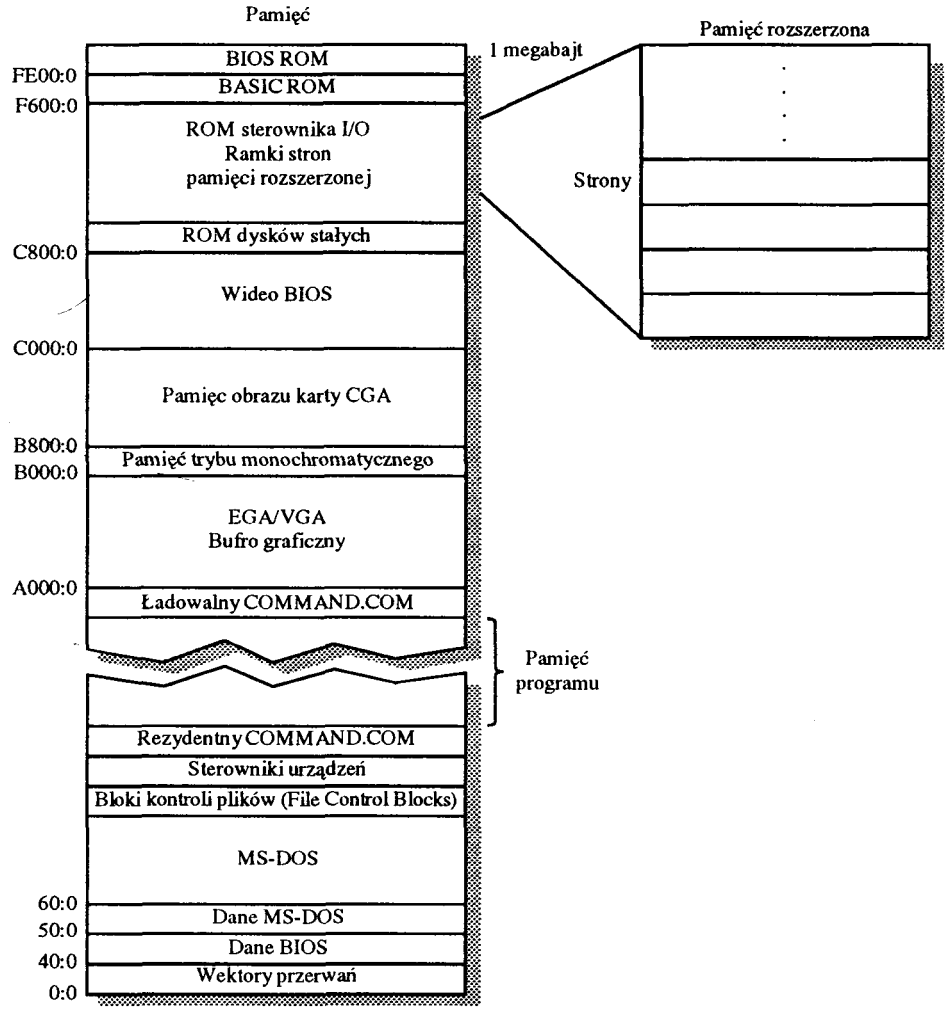
Dostęp do zasobów

W komputerach IBM PC i zgodnych dostęp do zasobów można zrealizować na cztery sposoby. Pierwszy z nich to bezpośrednie wykonywanie operacji na sprzęcie. (W rozdziale 14 zostały przedstawione dokładniej tego typu techniki). Trzy pozostałe to wykorzystanie DOS-u, BIOS-u oraz programowych sterowników urządzeń. Wybór konkretnego sposobu zależy od typu zasobu, jak również od rodzaju przeprowadzanych na nim operacji.

Pamięć

Każdy program musi korzystać z pamięci. Rysunek 13-6 przedstawia jej typowy podział dla komputerów PC. Gdy DOS uruchamia program, przydziela mu całą wolną pamięć w obszarze programów nierezydentnych (TPA). Po jego zakończeniu jest ona zwalniana i staje się znowu dostępna dla innych programów.

Programy, które uruchamiają inne aplikacje, lub też programy rezydentne (TSR), muszą zwolnić część pamięci w obszarze TPA tak, by mogła ona zostać przydzielona innym zadaniom. System DOS zawiera cztery funkcje, które służą do zmiany przydziału pamięci. Są to funkcje: **31h**, która kończy program, ale nie usuwająca go z pamięci, **48h**, która przydziela blok, **49h**, która zwalnia blok i **4Ah**, która zmienia rozmiar bloku pamięci.
Funkcja **31h** kończy program, ale nie usuwa go z pamięci. Przed jej wywołaniem należy się upewnić, że istnieje sposób na przejęcie sterowania przez ten program. Większość programów rezydentnych jest uaktywniana pojawieniem się określonego przerwania. Może to być wciśnięcie pewnego klawisza, pojawienie się pewnej liczby przerwań od układu czasowego, czy nawet przerwanie programowe, wygenerowane przez inny program.



Rysunek 13-6. ***Mapa pamięci w komputerach PC.***

By wywołać tę funkcję w rejestrze AH, należy umieścić numer **31h**, kod powrotu w AL, a w DX rozmiar żądanej pamięci wyrażony w 16-bajtowych paragrafach. Odpowiedni kod przedstawia poniższy przykład:

```
        .MODEL    SMALL
        .CODE
        .
        .
        .
        mov    ax, 3100h        ; Numer funkcji i kod powrotu
        mov    dx, SEG STACK    ; Pobranie adresu ostatniego
                                ; segmentu
        sub    dx, SEG _TEXT    ; Odjęcie od niego adresu
                                ; pierwszego segmentu wyznacza
                                ; rozmiar niezbędnej pamięci
        add    dx, 1024/16      ; Dodanie do niego rozmiaru stosu
        int    21h
        .STACK    1024
END
```

Funkcje **48h**, **49h** oraz **4Ah** pozwalają operować na blokach pamięci. Ponieważ w czasie ładowania DOS przydziela programowi całą dostępną pamięć, to część z niej musi zostać zwolniona przed użyciem tych funkcji. Poniższy przykład pokazuje, w jaki sposób, za pomocą funkcji **4Ah**, zmniejszyć blok programu do minimalnego rozmiaru:

```
        .MODEL   SMALL
        .CODE
Start   PROC
        push   ds               ; Po uruchomieniu rejestr
                                ; DS wskazuje na początek
                                ; bloku programu
        pop    es               ; Przesłanie go do ES jako
                                ; argumentu funkcji
        mov    dx, SEG STACK    ; Pobranie adresu ostatniego
                                ; segmentu
        sub    dx, SEG _TEXT    ; Odjęcie od niego adresu
                                ; pierwszego segmentu
        add    dx, 1024/16      ; Dodanie do rozmiaru stosu
        mov    ah, 4Ah          ; Funkcja, która zmienia
                                ; rozmiar bloku pamięci
        int    21h
        .
        .
        .
Start   ENDP
        .STACK   1024
END     Start
```

Po zredukowaniu rozmiaru bloku programu, można użyć funkcji **48h** i **49h** do przydzielania i zwalniania bloków pamięci. Aby przydzielić blok, należy umieścić jego rozmiar wyrażony w paragrafach w rejestrze BX i wywołać funkcję **48h**. Zwraca ona w rejestrze AX adres segmentu nowego bloku, jeśli znacznik przeniesienia jest wyzerowany. W przeciwnym razie AX zawiera kod błędu. Aby zwolnić blok, należy w rejestrze ES podać jego adres i wywołać funkcję **49h**.

Jeśli w systemie nie istnieje tyle pamięci, ile program żąda, to funkcja, która przydziela blok, zwróci błąd. W tym wypadku należy znaleźć sposób albo na zmniejszenie zapotrzebowania na pamięć albo na wykorzystanie większej ilości pamięci. Jednym ze sposobów zwiększenia wolnej pamięci jest zastosowanie sterownika pamięci rozszerzonej (EMS) firm Lotus, Intel Corporation i Microsoft (LIM). Jeśli pamięć ta jest fizycznie zainstalowana, to program może jej użyć, wydając temu sterownikowi odpowiednie komendy.

Sterownik LIM EMS pozwala wykorzystać na procesorach 8086 pamięć, która znajduje się poza granicą jednego megabajta. Sterownik ten może być również użyty na procesorach 80286 i 80386, pracujących w trybie rzeczywistym. Pamięć, którą on zarządza, jest podzielona na 16K strony. Sterownik kopiuje jedną lub kilka z tych stron do specjalnie w tym celu wyznaczonego obszaru pamięci podstawowej. Zarządza on również stronami, które zostały przydzielone innym programom. Następny przykład przedstawia sposób, w jaki program może wykorzystać sterownik LIM EMS do przechowania danych nie mieszczących się w pamięci głównej:

```
; Przykład wykorzystujący pamięć rozszerzoną
          .MODEL  SMALL
          .DATA
DevName   DB   "EMMXXXX0"     ; Nazwa sterownika LIM EMS
EMMHand   DW   ?              ; Deskryptor bloku pamięci EMS
PageSeg   DW   ?              ; Segment pamięci podstawowej
                              ; przeznaczony do, przez który
                              ; widziana jest pamięć EMS
PageNumb  DW   0              ; Pierwsza strona w tym oknie
          .CODE
          JUMPS               ; Zezwolenie na bliskie skoki
                              ; warunkowe
LIMexam   PROC
          mov   ax, SEG DGROUP
          mov   ds, ax
; Sprawdzenie czy pamięć LIM jest zainstalowana
          sub   ax, ax
          mov   es, ax        ; Tablica przerwań znajduje
                              ; się w zerowym segmencie
          mov   bx, 19Ch      ; Przemieszczenie wektora
                              ; przerwania 67h
                              ; (sterownik EMS)
          les   bx, DWORD PTR es:[bx]
          mov   cx, 8
          mov   di, 0Ah       ; Przemieszczenie nazwy
```

```
                                ; sterownika
        mov   si, OFFSET DevName
        cld
        repe  cmpsb             ; Porównanie nazw
        jne   Exit              ; Jeśli nie są identyczne,
                                ; koniec
; Czy jest dostępne wystarczająco dużo wolnych stron
        mov   ah, 42h           ; Pobranie ilości nie
                                ; przydzielonych stron
        int   67h
        or    ah, ah            ; Sprawdzenie kodu powrotu
        jnz   Exit
        cmp   bx, 10            ; Czy jest wolnych więcej
                                ; niż 10 stron
        jb    Exit
; Przydzielenie stron
        mov   ah, 43h           ; Przydzielenie stron
                                ; pamięci LIM
        mov   bx, 10            ; Liczba stron
        int   67h
        or    ah, ah
        jnz   Exit
        mov   EMMHand, dx       ; Zapamiętaj deskryptor
; Pobranie adresu ramki strony
        mov   ah, 41h           ; Pobranie adresu ramki strony
        int   67h
        or    ah, ah
        jnz   Exit
        mov   PageSeg, bx       ; Zapamiętaj numer segmentu
                                ; ramki strony
; Przesłanie danych do pamięci
        cld
loop1:
        mov   ah, 44h           ; Odwzorowanie strony EMS
                                ; w ramkę strony
        mov   al, 0             ; Odwzorowywana będzie
                                ; strona zerowa
        mov   bx, PageNumb      ; Numer strony EMS
        mov   dx, EMMHand       ; Deskryptor bloku
                                ; pamięci EMS
        int   67h
        or    ah, ah
        jnz   DeAlloc
        mov   cx, 3FFFh         ; Liczba bajtów na stronie
        mov   es, PageSeg       ; Wskazania na ramkę strony
        sub   di, di
```

```
        sub   al, al        ; Wyzerowanie AL
loop2:
        stosb
        loop  loop2
        inc   PageNumb      ; Następna strona
        cmp   PageNumb, 10
        jb    loop1
; Zwolnienie stron
DeAlloc:
        mov   ah, 45h       ; zwolnienie bloku pamięci EMS
        mov   dx, EMMHand
        int   67h
; Powrót do DOS-u
Exit:
        mov   ah, 4Ch
        int   21h
LIMexam ENDP
        .STACK   512
END     LIMexam
```

Pierwsza część funkcji sprawdza, czy został zainstalowany sterownik LIM EMS. Z tablicy wektorów przerwań pobiera ona adres punktu wejścia do procedury obsługi przerwania **67h**. Następnie sprawdza zawartość pamięci 10 bajtów za tym adresem. Jeśli zawiera ona łańcuch **"EMMXXXX0"**, to sterownik jest zainstalowany. W dalszej części przykładu znajdują się wywołania funkcji, konieczne do wykorzystania pamięci LIM EMS. Pierwsza z nich (**AH = 42h**) zwraca ilość wolnych stron. Następna (**AH = 43h**) powoduje przydzielenie 10 stron pamięci LIM EMS. Za pomocą funkcji **41h** pobierany jest adres ramki strony. Jest to obszar pamięci głównej, który zawiera określone strony pamięci LIM EMS. W tym właśnie obszarze funkcja **44h** umieszcza dane, które znajdują się w pamięci LIM EMS. Pętle **loop1** i **loop2** wypełniają wszystkie strony zerami. Ostatnia funkcja (**AH = 45h**) zwalnia używane strony. Niewątpliwie większość aplikacji będzie wykonywała więcej i bardziej złożonych operacji, niż przedstawiony przykładowy program. Dzięki niemu można jednak zrozumieć zasadę działania sterownika pamięci LIM EMS.

Klawiatura i ekran

MS-DOS i BIOS dostarczają wielu funkcji, które pozwalają obsługiwać klawiaturę oraz ekran. Są one przedstawione w tabeli 13-3. (W Dodatku C można znaleźć więcej informacji o nich).

Wiele z tych funkcji zostało już wykorzystanych w zamieszczonych przykładach. Zastosowanie kilku nowych można zobaczyć w przykładzie dotyczącym obsługi myszy.

Możliwości MS-DOS-u mogą być rozszerzone poprzez dołączenie sterowników urządzeń. Sterownik to zbiór funkcji, które mogą być wywoływane przez system przy odwoływaniu się do danego urządzenia. MS-DOS rozpoznaje dwa typy urządzeń, blokowe i znakowe. Urządzenia blokowe to na przykład dyski, które zawierają pliki. Są one identyfikowane pojedynczymi literami alfabetu. Urządzenia znakowe mogą odczytać lub zapisać tylko jeden

znak. Są to np. drukarki, modemy. Są one identyfikowane przez swoją nazwę. MS-DOS, odwołując się do klawiatury i ekranu, używa nazwy CON:.
Urządzenia mogą być do systemu dodawane lub wymieniane przez podanie w pliku konfiguracyjnym CONFIG.SYS nazwy zbioru, zawierającego sterownik. Kilka sterowników jest dostarczanych wraz z systemem MS-DOS. Jeden z nich, ANSI.SYS zastępuje wbudowane urządzenie CON:. Przegląda on wszystkie znaki, wysyłane na konsolę i wyszukuje w nich łańcuchy swoich poleceń. Aby zainstalować ten sterownik, należy w pliku CONFIG.SYS umieścić poniższą linię i wyzerować maszynę:

```
device=c:\ansi.sys
```

Tabela 13-3. Funkcje dotyczące do obsługi klawiatury i ekranu.

Przerwanie	Wejście	Wyjście	Opis
10h	AH=2 BH=numer strony ekranu DH=wiersz DL=kolumna	Brak	Ustaw pozycję kursora
10h	AH=6 AL=Ilość linii BH=Atrybut wypełnienia CH=Górny wiersz CL=Lewa kolumna DH=Dolny wiersz DL=Prawa kolumna	Brak	Przewinięcie zawartości okienka w górę
10h	AH=7 AL=Ilość linii BH=Atrybut wypełnienia CH=Górny wiersz CL=Lewa kolumna DH=Dolny wiersz DL=Prawa kolumna	Brak	Przewinięcie zawartości okienka w dół
10h	AH=9 AL=Znak BH=Numer strony BL=Atrybuty znaku CX=Ilość powtórzeń	Brak	Zapis znaku i jego atrybutów
10h	AH=0Ah AL=Znak BH=Numer strony BL=Kolor znaku CX=Ilość powtórzeń	Brak	Zapis znaku
10h	AH=0Eh AL=Znak BH=Numer strony BL=Kolor znaku	Brak	Zapis znaku w trybie dalekopisowym

Przerwanie	Wejście	Wyjście	Opis
16h	AH=0	AH=numer klawisza AL=kod ASCII	Odczyt znaku z klawiatury
16h	AH=1	Znacznik ZF jest ustawiony, gdy wciśnięty jest klawisz AH=numer klawisza AL=kod ASCII	Sprawdzenie stanu klawiatury
16h	AH=2	AL= bit 7 = Insert bit 6 = Caps lock bit 5 = Num lock bit 4 = Scroll lock bit 3 = Alt bit 2 = Ctrl bit 1 = Lewy shift bit 0 = Prawy shift	Pobranie stanu klawiszy sterujących
21h	AH=1	AL=Znak	Odczytanie znaku z klawiatury i wysłanie echa na ekran
21h	AH=7	AL=Znak	Bezpośrednie odczytanie znaku z klawiatury bez wysłania echa
21h	AH=8	AL=Znak	Odczytanie znaku z klawiatury bez wysłania echa
21h	AH=0Ah DS:DX=adres bufora wejściowego		Pobranie wiersza
21h	AH=0Bh	AL=0FFh, jeśli jest wciśnięty klawisz AL=0, gdy nie jest wciśnięty żaden klawisz	Sprawdzenie stanu klawiatury
21h	AH=2 DL=Znak		Wyświetlenie znaku
21h	AH=9 DS:DX=łańcuch zakończony znakiem '$'		Wyświetlenie łańcucha

Przy takim zapisie plik ANSI.SYS musi znajdować się w katalogu głównym. Jeśli znajduje się on jednak w innym katalogu, to należy podać do niego pełną ścieżkę dostępu.

Jeśli sterownik ANSI.SYS jest załadowany, wówczas programy mogą wysyłać do niego polecenia, używając zwykłych funkcji MS-DOS-u do komunikacji z konsolą. W tabeli 13-4 przedstawiono kilka łańcuchów jego komend.

Tabela 13-4. Polecenia sterownika ANSI.SYS

W poniższych komendach x jest liczbą wyrażoną w cyfrach ASCII.

Komenda	Funkcja
ESC[x;xH	Ustawienie pozycji kursora. Jeśli nie zostanie podana żadna współrzędna, kursor zostanie przesunięty do lewego górnego rogu.
ESC[xA	Przesunięcie kursora o x linii do góry
ESC[xB	Przesunięcie kursora o x linii w dół
ESC[xC	Przesunięcie kursora o x kolumn w prawo
ESC[xD	Przesunięcie kursora o x kolumn w lewo
ESC[6n	Zwrócenie raportu o stanie urządzenia
ESC[s	Zapisanie bieżącej pozycji kursora
ESC[u	Odtworzenie pozycji kursora
ESC[2J	Wyczyszczenie ekranu
ESC[K	Wyczyszczenie linii
ESC[x;...;xm	Ustawienie atrybutów grafiki
	0 Wszystkie atrybuty wyłączone
	1 Pogrubienie
	2 Cieniowanie
	3 Kursywa
	5 Migotanie
	6 Szybkie migotanie
	7 Inwersja
	8 Wyświetlenie znaków ukrytych
	30 Czarny
	31 Czerwony
	32 Zielony
	33 Żółty
	34 Niebieski
	35 Karmazynowy
	36 Turkusowy
	37 Biały
	40 Czarny
	41 Czerwony
	42 Zielony
	43 Żółty
	44 Niebieski
	45 Karmazynowy
	46 Turkusowy
	47 Biały
	48 Indeks dolny

Komenda	Funkcja
	49 Indeks górny
ESC=xh	Ustawienie trybu video
	0 40 X 25 Czarno-biały
	1 40 X 25 Kolor
	2 80 X 25 Czarno-biały
	3 80 X 25 Kolor
	4 320 X 200 Kolor
	5 320 X 200 Czarno-biały
	6 640 X 200 Czarno-biały
	7 Zawijanie na końcu wiersza

Poniższy przykład, posługując się komendami sterownika ANSI.SYS, czyści ekran, a następnie na jego środku wyświetla komunikat:

```
        .MODEL  SMALL
ESC     EQU  27h                ; Definicja znaku ESC
        .DATA
MsgStr  DB   ESC, "[2J"         ; Wyczyszczenie ekranu
        DB   ESC, "[12;34"      ; Ustawienie pozycji kursora
        DB   "Naciśnij dowolny klawisz$"  ; Komunikat
        .CODE
Start   PROC
        mov   ax, SEG DGROUP
        mov   ds, ax
        mov   dx, OFFSET MsgStr; Pobranie adresu łańcucha
        mov   ah, 9             ; Funkcja, która wyświetla
                                ; łańcuch znaków
        int   21h
        mov   ah, 0
        int   16h               ; Czekaj na naciśnięcie
                                ; klawisza
        mov   ah, 4Ch
        int   21h
Start   ENDP
        .STACK
END     Start
```

Dysk

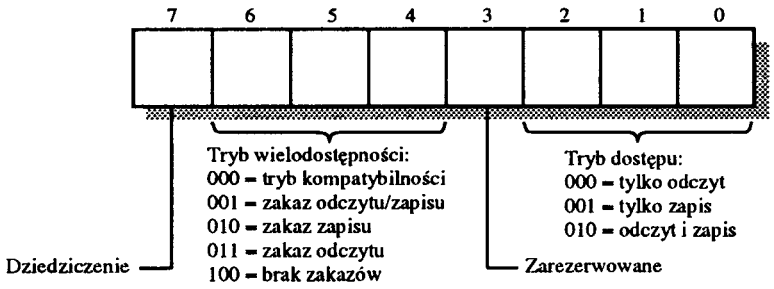
MS-DOS posiada dwa zbiory funkcji, które operują na plikach. W pierwszym z nich otwarte pliki są identyfikowane za pomocą bloków FCB (File Control Blok). Funkcje zostały dodane dla zapewnienia zgodności ze starszymi systemami operacyjnymi. Są one jednak niewygodne w użyciu. Pozostałe funkcje do identyfikacji plików używają uchwytów.
Przed wykonaniem operacji na pliku musi on zostać otworzony. MS-DOS wyszukuje plik o podanej nazwie, a po jego znalezieniu informację o nim umieszcza w specjalnej tablicy.

Następnie zwraca on użytkownikowi uchwyt otworzonego pliku, który jest używany przy każdej operacji na zbiorze. Poniższy przykład pokazuje sposób otwierania pliku:

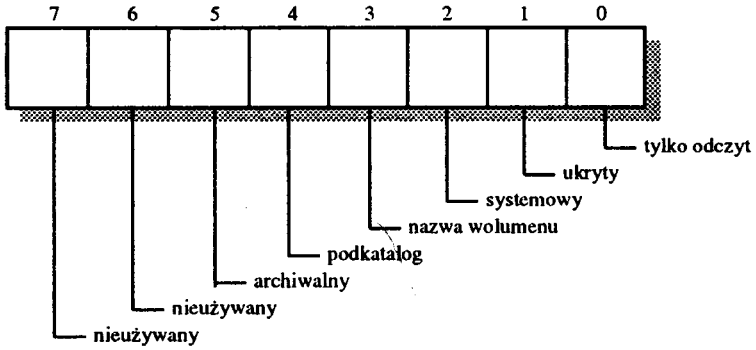
```
            .MODEL  SMALL, C
            .CODE
; Otwarcie pliku o nazwie FName. FName jest wskaźnikiem
; umieszczonym na stosie, który to wskaźnik wskazuje nazwę
; pliku. Informacja o dyrektywie ARG znajduje się
; w rozdziale 10.
OpenFile    PROC
ARG         FName:PTR
            mov   dx, FName    ; Pobierz adres nazwy pliku
            mov   ax, 3D02h    ; Otwórz plik w trybie do
                               ; czytania i pisania
            int   21h
            jnc   Opdone       ; Jeśli nie wystąpił błąd,
                               ; to rejestr AX zawiera
                               ; uchwyt pliku
                               ; Plik nie może być otworzony,
                               ; ponieważ najprawdopodobniej
                               ; nie istnieje. Należy go więc
                               ; utworzyć
            mov   dx, FName    ; Pobierz adres nazwy pliku
            mov   cx, 0
            mov   ah, 3Ch      ; Utwórz nowy plik
            int   21h
            jnc   Opdone       ; Jeśli nie wystąpił błąd,
                               ; rejestr AX zawiera uchwyt
                               ; pliku
            sub   ax, ax       ; Jeśli wystąpił błąd rejestr
                               ; AX zawiera wartość 0
Opdone:
            ret
OpenFile    ENDP
END
```

W pierwszej części procedury podejmowana jest próba otwarcia istniejącego pliku do czytania i pisania. Rejestry **DS:DX** wskazują nazwę pliku, a rejestr AL zawiera prawa dostępu do niego. Rysunek 13-7 przedstawia znaczenie jego poszczególnych bitów. Jeśli jednak plik o podanej nazwie nie istnieje, wywoływana jest funkcja MS-DOS-u, która powoduje jego utworzenie. Przy tworzeniu pliku muszą zostać podane jego atrybuty. Przekazuje się je w rejestrze CX. Ich znaczenie zostało podane na rysunku 13-8.
Uchwyt pliku, zwrócony przez funkcję MS-DOS-u, jest następnie używany w pozostałych funkcjach operujących na nim. Na przykład, by wykonać operację zapisu, należy podać dane, ich rozmiar oraz uchwyt pliku:

```
.MODEL SMALL, C
.CODE
```



Rysunek 13-7. ***Prawa dostępu do pliku.***



Rysunek 13-8. ***Atrybuty pliku.***

```
; Zapis danych z bufora Buff do pliku o uchwycie Hand.
; Cnt oznacza ilość przesyłanych bajtów. Wszystkie te
; zmienne są umieszczone w ramce stosu (Patrz rozdział 10).
WriteFile PROC
ARG       Hand:WORD, Buff:PTR, Cnt:WORD
          mov   ah, 40h     ; Funkcja, która zapisuje
                            ; ciąg bajtów do pliku
          mov   bx, Hand    ; Uchwyt pliku w rejestrze BX
          mov   dx, Buff    ; Adres bufora danych w
                            ; rejestrach DS:DX
          mov   cx, Cnt     ; Liczba przesyłanych
                            ; bajtów w CX
          int   21h
          ret
WriteFile ENDP
END
```

Operacja odczytu wygląda niemal dokładnie tak samo. Tylko zamiast funkcji zapisującej została użyta funkcja **3Fh**, która odczytuje dane z pliku:

```
          .MODEL  SMALL, C
          .CODE
; Odczyt danych z pliku o uchwycie Hand do bufora Buff.
; Cnt oznacza ilość przesyłanych bajtów. Wszystkie
; te zmienne są umieszczone w ramce stosu
; (Patrz rozdział 10).
ReadFile  PROC
ARG       Hand:WORD, Buff:PTR, Cnt:WORD
          mov   ah, 3Fh     ; Funkcja, która odczytuje
                            ; z pliku ciąg bajtów
          mov   bx, Hand    ; Uchwyt pliku w rejestrze BX
          mov   dx, Buff    ; Adres bufora danych w
                            ; rejestrach DS:DX
          mov   cx, Cnt     ; Liczba przesyłanych
                            ; bajtów w CX
          int   21h
          ret
ReadFile  ENDP
END
```

Jednym z elementów informacji, którą MS-DOS przechowuje o każdym otwartym zbiorze, jest wskaźnik pozycji w pliku. Wskazuje on miejsce, od którego zacznie się następna operacja odczytu lub zapisu. Każda taka operacja powoduje przesunięcie tego wskaźnika na koniec przesłanych danych. Taki sekwencyjny sposób dostępu do danych nadaje się, np. do plików tekstowych, które są najczęściej przeglądane od początku do końca. Zawodzi jednak zupełnie w wypadku baz danych, gdzie operacje zapisu i odczytu nie są przeprowadzane w taki uporządkowany sposób. Można jednak wtedy wykorzystać funkcję **42h**, która powoduje przesunięcie wskaźnika pliku na żądaną pozycję. Pozycja ta może być określana względem początku albo końca pliku lub względem bieżącej wartości wskaźnika. Asemblerowa funkcja, realizująca tę operację, może wyglądać następująco:

```
          .MODEL  SMALL, C
          .CODE
; Przesunięcie wskaźnika pliku o uchwycie Hand o Locat
; bajtów. Subfunc oznacza sposób przesuwania. Informacja
; o dyrektywie ARG znajduje się w rozdziale 10.
lseek     PROC
ARG       Hand:WORD, Locat:DWORD, subfunc:BYTE
          mov   ah, 42h
          mov   bx, Hand        ; Uchwyt pliku w
                                ; rejestrze BX
          mov   dx, LOW Locat   ; Młodsze słowo w DX
          mov   cx, HIGH Locat  ; Starsze słowo w CX
```

```
        mov   al, subfunc     ; 0 - przesunięcie względem
                              ; początku pliku
                              ; 1 - przesunięcie względem
                              ; bieżącej pozycji wskaźnika
                              ; pliku
                              ; 2 - przesunięcie względem
                              ; końca pliku
        int   21h
        ret
lseek   ENDP
END
```

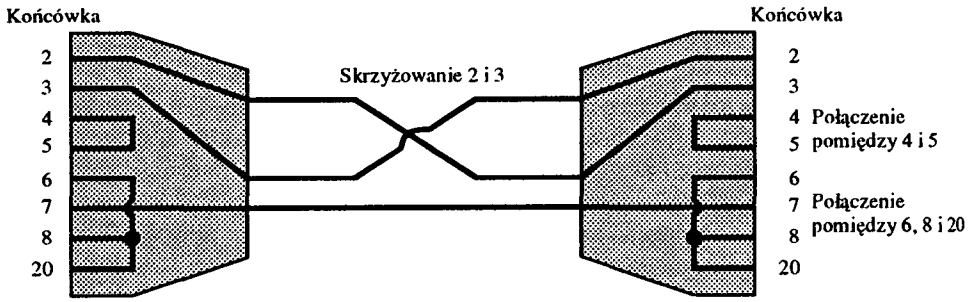
Gdy wszystkie operacje na pliku zostaną już wykonane, wówczas musi on zostać zamknięty. Operacja ta powoduje zapisanie na dysk wszystkich niezbędnych o nim informacji. Ponadto MS-DOS ogranicza ilość otwartych plików w systemie. Tak więc, zamknięcie pliku powoduje zwolnienie zajmowanego przez niego miejsca, które może zostać przydzielone innemu plikowi. Operacji zamknięcia pliku można dokonać, wywołując funkcję **3Eh** lub **4Ch** kończącą program. Powoduje ona między innymi zamknięcia wszystkich otwartych w programie plików. Mimo że operacja ta jest wykonywana przez DOS automatycznie na zakończenie programu, to powinno się zamykać pliki zaraz po wykonaniu na nich żądanych działań. Przed zakończeniem programu może się przecież zdarzyć coś nieprzewidzianego np. zanik zasilania. Może to spowodować, że nie będzie można z tego pliku dalej korzystać. Poniższa funkcja zamyka wskazany plik:

```
        .MODEL  SMALL, C
        .CODE
; Zamknięcie pliku o uchwycie Hand. Informacja o
; dyrektywie ARG znajduje się w rozdziale 10.
Close   PROC
ARG     Hand:WORD
        mov   ah, 3Eh       ; Funkcja zamykająca plik
        mov   bx, Hand      ; Uchwyt pliku w rejestrze BX
        int   21h
        ret
Close   ENDP
END
```

Porty szeregowe i równoległe

System MS-DOS zawiera wbudowane sterowniki portów szeregowych i równoległych. Porty szeregowe identyfikowane są przez nazwy COM1: i COM2:, port równoległy zwany jest LPT1:. Urządzenia te mogą być traktowane jak pliki. Na przykład, wysyłanie i odbieranie z nich danych może się odbywać za pośrednictwem funkcji zapisu i odczytu z pliku. (Jednak w większości systemów do urządzenia LPT1: można dane tylko zapisywać).
Jednak istnieją operacje, które z poziomu systemu DOS są niedostępne. Na przykład, żadna jego funkcja nie może zmienić prędkości transmisji portu szeregowego. Operację tę można jednak przeprowadzić wykorzystując odpowiednie funkcje BIOS-u. Porty szeregowe są obsługiwane przez przerwanie **14h**, a równoległe -przez przerwanie **17h**.

Poniższy program realizuje transmisję danych pomiędzy konsolą a portami szeregowymi i równoległymi. Może on współpracować z modemem lub może służyć do komunikacji z innym komputerem. Komputery muszą być jednak połączone kablem, który jest przedstawiony schematycznie na rysunku 13-9. Procedury BIOS-u mogą okazać się jednak zbyt wolno działające w stosunku do wymagań pewnych programów. W tych wypadkach należy użyć funkcji znajdujących się w rozdziale 14, które to funkcje bezpośrednio operują na układach portów.



Rysunek 13-9. ***Kabel łączący dwa komputery.***

```
; Przesyłanie danych pomiędzy konsolą a portami szeregowymi
; i równoległymi
        .MODEL  SMALL
        .DATA
Prnname DB   "LPT1:", 0
Prnhand DW   ?
Serdata DB   ?
        .CODE
Start   PROC
        mov   ax, SEG DGROUP
        mov   ds, ax
; Inicjacja portu szeregowego
        mov   ah, 0
        mov   al, 01000011b       ; Prędkość 300 bodów,
                                  ; bez bitu parzystości,
                                  ; 1 bit stopu, 8 bitów
                                  ; danych
        mov   dx, 0               ; Wywołanie funkcji
                                  ; BIOS-u,  które
                                  ; obsługują port
                                  ; szeregowy
        int   14h
; Inicjacja portu drukarki
        mov   ah, 3Dh             ; Funkcja DOS-u, która
```

```
                                        ; otwiera plik
        mov     al, 1                   ; Otwarcie tylko
                                        ; do zapisu
        mov     dx, OFFSET Prnname
        int     21h
        mov     Prnhand, ax             ; Zapamiętanie uchwytu
; Pętla główna
mloop:
; Sprawdzenie stanu klawiatury
        mov     ah, 1                   ; Funkcja BIOS-u, która
                                        ; testuje stan klawiatury
        int     16h
        jnz     NoKey
        mov     ah, 0                   ; Odczytanie znaku
        int     16h
; Wysłanie znaku do portu szeregowego
        mov     ah, 1                   ; Funkcja BIOS-u,
                                        ; która wysyła znak
                                        ; z rejestru
        mov     dx, 0                   ; AL do portu szeregowego
                                        ; o numerze w DX
        int     14h
NoKey:
; Sprawdzenie stanu portu szeregowego
        mov     ah, 3                   ; Funkcja BIOS-u, która
                                        ; pobiera stan portu
        mov     dx, 0
        int     14h
        test    al, 1                   ; Czy jest sygnał
                                        ; gotowości danych
        jz      NoSerial
        mov     ah, 2                   ; Funkcja BIOS-u, która
                                        ; pobiera znak z portu
        mov     dx, 0
        int     14h
        mov     Serdata, al             ; Zapamiętanie znaku
; Wyświetlenie znaku na ekranie
        mov     dl, al                  ; Umieszczenie znaku w
                                        ; rejestrze DL
        mov     ah, 2                   ; Funkcja DOS-u, która
                                        ; wyświetla jeden znak
        int     21h
; Wysłanie znaku na drukarkę
        mov     ah, 40h                 ; Funkcja DOS-u, która
                                        ; zapisuje dane do pliku
        mov     dx, OFFSET Serdata      ; Adres danych
```

```
        mov   cx, 1              ; Zapisz 1 znak
        int   21h
NoSerial:
        jmp   mloop
Start   ENDP
        .STACK  512
END     Start
```

W pierwszej części programu przeprowadzana jest inicjalizacja portów szeregowych i równoległych. Następnie w pętli, zaczynającej się od etykiety mloop, sprawdzany jest stan klawiatury i portu szeregowego. Jeśli zostanie wciśnięty klawisz, to jest on wysyłany do portu szeregowego. Natomiast znaki z niego odczytywane są wyświetlane na ekranie i drukowane na drukarce.

Mysz

Coraz więcej komputerów jest wyposażanych w mysz. Dzięki temu urządzeniu obsługa wielu programów staje się łatwiejsza i szybsza. W komputerach, pracujących pod kontrolą systemu MS-DOS, komunikacja z myszą odbywa się poprzez jej sterownik. Program ten jest dostarczany razem z myszą. Jego zadaniem jest śledzenie położenia myszy i obsługa przerwań wywoływanych przez programy, które jej używają.

Sterowniki, pochodzące od różnych producentów, wymagają różnych sposobów ich instalowania. Wszystkie one dostarczają jednak jednolitego sposobu komunikacji. Polecenia te muszą być umieszczone w rejestrach, a następnie należy wywołać przerwanie **33h**. Jest to więc metoda podobna do tej, stosowanej przez MS-DOS i BIOS. Różnica polega tylko na tym, że oba te programy są zawsze dostępne, podczas gdy sterownik myszy nie musi być zainstalowany. Dlatego też pierwszym krokiem programu, który chce wykorzystać mysz, jest sprawdzenie obecności jej sterownika.

Operacja ta nie jest trudna. Należy tylko sprawdzić, czy obsługiwane jest przerwanie **33h**. Adres procedury obsługi tego przerwania znajduje się w miejscu pamięci o adresie **0000:00CC** (33h * 4). Jeśli daleki wskaźnik, który znajduje się w tym miejscu, jest równy zeru lub wskazuje na instrukcję **IRET**, to sterownik myszy nie został zainstalowany. Dopiero po wykonaniu tego testu można używać funkcji, które sterują myszą.

Poniższy program rozpoczyna właśnie swoje działanie od przeprowadzenia tego testu. Jeśli sterownik myszy został zainstalowany, to jest ona inicjowana i wyświetlany jest jej kursor. Następnie obsługiwane są zdarzenia. Jeśli zostanie wciśnięty lewy przycisk myszy, wówczas w miejscu kursora zostanie wyświetlony znak (*). Jeśli zostanie wciśnięty prawy przycisk, to wyświetlony zostanie znak (^). Aktualne współrzędne kursora myszy są wyświetlane w lewym górnym rogu ekranu.

```
        .MODEL  SMALL
        .DATA
BStatus DW  ?                ; Stan przycisków myszy
BXpos   DW  ?                ; Numer kolumny kursora
BYpos   DW  ?                ; Numer wiersza kursora
        .CODE
Mouse   PROC
        mov   ax, SEG DGROUP
```

```
        mov   ds, ax
; Sprawdzenie obecności sterownika myszy
        sub   ax, ax        ; Wyzerowanie rejestru AX
        mov   es, ax
        mov   bx, 0CCh      ; Adres procedury obsługi
                            ; przerwania 33h
        mov   ax, WORD PTR es:[bx]
        or    ax, WORD PTR es:[bx+2]
        jz    Exit          ; Jeśli wektor przerwania
                            ; jest zerowy, koniec
        les   bx, DWORD PTR es:[bx]
        cmp   BYTE PTR es:[bx], 0CFh
        jnz   MouseOK       ; Jeśli nie jest to instrukcja
                            ; IRET, to sterownik jest
                            ; zainstalowany
Exit:
        mov   ah, 4Ch       ; Funkcja, która kończy
                            ; program
        int   21h
MouseOK:
; Inicjacja myszy
        mov   ah, 0         ; Funkcja, która zeruje mysz
        int   33h           ; Jeśli mysz jest nieobecna,
                            ; to funkcja zwraca w
                            ; rejestrze AH wartość 0FFh
        inc   ah            ; Czy zainstalowano mysz
        jz    Exit
        mov   ah, 10        ; Komenda, która ustawia
                            ; kursor tekstowy
        mov   bx, 0         ; Będzie to kursor programowy
        mov   cx, 0f000h    ; Maska
        mov   dx, 0704h     ; Znak kursora
        int   33h
; Wyczyszczenie ekranu
        mov   ah, 6         ; Funkcja BIOS-u, która
                            ; przewija zawartość ekranu
        mov   al, 25        ; o 25 linii
        mov   bh, 7         ; Atrybuty znaków
        mov   cx, 0         ; Lewy górny róg
        mov   dx, 184Ah     ; Prawy dolny róg
        int   10h
MainLP:
        mov   ah, 1         ; Testowanie stanu klawiatury
        int   16h
        jnz   NoKey
        mov   ah, 0         ; Funkcja BIOS-u, która
```

```
                                ; odczytuje znak
          int   16h             ; Pobranie znaku
                                ; i zakończenie programu
          mov   ah, 4Ch
          int   21h
NoKey:
; Ustalenie pozycji kursora myszy i stanu jej przycisków
          mov   ax, 3
          int   33h
          mov   BStatus, bx ; Zapamiętanie stanu
                                ; przycisków
          mov   BXpos, cx   ; Zapamiętanie numeru
                                ; kolumny kursora
          mov   BYpos, dx   ; Zapamiętanie numeru
                                ; wiersza kursora
; Sprawdzenie stanu lewego przycisku
          cmp   BStatus, 1
          jne   NotLeft
          mov   ah, 2           ; Ustawienie pozycji
                                ; kursora ekranu
          sub   bh, bh          ; Zerowa strona ekranu
          mov   dh, dl          ; Numer wiersza w DH
          mov   dl, cl          ; Numer kolumny w DL
          int   10h
          mov   ax, 2           ; Ukrycie kursora myszy.
                                ; Przed każdą operacją,
                                ; która zmienia zawartość
                                ; ekranu, kursor myszy musi
                                ; być chowany. Po prostu nie
                                ; może on zostać zamazany.
          mov   ah, 2           ; Wyświetlenie znaku na ekranie
          mov   dh, '*'
          int   21h
          mov   ax, 1           ; Ponowne wyświetlenie
                                ; kursora myszy
          int   33h
; Sprawdzenie stanu prawego przycisku
NotLeft:
          cmp   BStatus, 2
          jne   NotRight
          mov   ah, 2           ; Ustawienie pozycji kursora
                                ; ekranu
          sub   bh, bh          ; Zerowa strona ekranu
          mov   dh, dl          ; Numer wiersza w DH
          mov   dl, cl          ; Numer kolumny w DL
          int   10h
```

```
        mov   ax, 2        ; Ukrycie kursora myszy.
                           ; Przed każdą operacją, która
                           ; zmienia zawartość ekranu,
                           ; kursor myszy musi być
                           ; chowany. Po prostu nie może
                           ; on zostać zamazany.
        mov   ah, 2        ; Wyświetlenie znaku na ekranie
        mov   dh, '^'
        int   21h
        mov   ax, 1        ; Ponowne wyświetlenie
                           ; kursora myszy
        int   33h
; Wyświetlenie w lewym górnym rogu ekranu współrzędnych
; kursora myszy
NotRight:
        mov   ah, 2        ; Ustawienie pozycji
                           ; kursora ekranu
        sub   bh, bh
        sub   dx, dx
        int   10h
        mov   ax, BXpos    ; Pobranie numeru kolumny
        aam                ; Konwersja na rozpakowany
                           ; kod BCD
        push  ax           ; Zachowanie rezultatu
        mov   dh, ah       ; Starsza cyfra w DH
        add   dh, '0'      ; Konwersja na znak ASCII
        mov   ah, 2        ; Wyświetlenie znaku na
                           ; ekranie
        int   21h
        pop   ax           ; Odtworzenie rezultatu
        mov   dh, al       ; Młodsza cyfra w DH
        add   dh, '0'      ; Konwersja na znak ASCII
        mov   ah, 2        ; Wyświetlenie znaku na ekranie
        int   21h
        mov   ah, 2
        mov   dh, ','
        int   21h
        mov   ax, BYpos    ; Pobranie numeru wiersza
        aam                ; Konwersja na rozpakowany
                           ; kod BCD
        push  ax           ; Zachowanie rezultatu
        mov   dh, ah       ; Starsza cyfra w DH
        add   dh, '0'      ; Konwersja na znak ASCII
        mov   ah, 2        ; Wyświetlenie znaku
                           ; na ekranie
        int   21h
```

```
        pop     ax          ; Odtworzenie rezultatu
        mov     dh, al      ; Młodsza cyfra w DH
        add     dh, '0'     ; Konwersja na znak ASCII
        mov     ah, 2       ; Wyświetlenie znaku
                            ; na ekranie
        int     21h
        jmp     MainLP
Mouse   ENDP
        .STACK  1024
END      Mouse
```

Pierwsza część funkcji sprawdza, czy jest zainstalowany sterownik myszy i czy została ona przyłączona. Jeśli wszystko jest w porządku, to czyszczony jest ekran i zaczyna się wykonywać pętla **MainLP**. Jeśli nie został wciśnięty żaden klawisz, wówczas sprawdzany jest stan przycisków myszy oraz ustalana jest pozycja jej kursora. Wciśnięcie lewego przycisku powoduje wyświetlenie na ekranie znaku "*", a prawego - znaku "^". Fragment kodu za etykietą **NotRight** wyświetla aktualną pozycję kursora i przekazuje sterowanie na początek pętli.

Podsumowanie

Komputery IBM PC dostarczają wiele zasobów, które mogą być wykorzystywane w programach. Jednak przed ich użyciem należy sprawdzić, które z nich są dostępne. Operacja ta może być przeprowadzona na wieloma sposobami. Najprostszy polega na wykorzystaniu odpowiednich funkcji i danych BIOS-u. Część informacji może być również pobrana z portu **60h** lub rejestrów zegara czasu rzeczywistego. Ostatni metoda polega na napisaniu programu, testującego obecność danego zasobu.

Po upewnieniu się o dostępności danego zasobu, można go już wykorzystać. W systemie DOS istnieje wiele funkcji, które ułatwiają zarządzanie podstawowymi zasobami, jak pamięć, konsola czy dyski. Inne ,jak karta graficzna czy porty szeregowe, są dostępne poprzez funkcje BIOS-u. Ponadto pewne urządzenia są dostępne poprzez specjalne programy, dostarczane wraz z nimi przez producenta. Po zainstalowaniu takiego programu do urządzenia można się już odwoływać w podobny sposób, jak do zasobów obsługiwanych przez DOS. Przykładem tego typu zasobów może być pamięć rozszerzona lub mysz.

Rozdział 14 Dostęp i sterowanie sprzętem

Jednym z powodów, dla których stosuje się język asemblera jest fakt, że można w nim bezpośrednio operować na sprzęcie. Ma to dwie zasadnicze zalety: Można w ten sposób wykorzystać wszystkie możliwości, jakie oferuje sprzęt, a ponadto operacje bezpośrednie są najczęściej wykonywane o wiele szybciej, niż podobne przeprowadzane przez DOS czy BIOS.
Większość istniejących kart graficznych ma tryby, które wymagają specjalnego oprogramowania. Jest ono dostępne przede wszystkim w języku asemblera. Również zmiana priorytetów przerwań czy zmiana kolejności odczytywania kombinacji klawiszy (jak ALT-SHIFT) nie jest na ogół możliwa w językach wysokiego poziomu. Wynika to stąd, że aby sprostać tym problemom konieczne jest zastosowanie instrukcji IN i OUT oraz rozkazów, które włączają i wyłączają przerwania. Większość języków wysokiego poziomu nie udostępnia tego typu instrukcji.
Pierwsze dwie części rozdziału zawierają opis sprzętu, który steruje klawiaturą i przerwaniami. Przedstawiony tam zostanie między innymi sposób wykorzystania nie używanych kombinacji klawiszy do pobudzenia programu rezydentnego.
Sterownik portów szeregowych zawarty w BIOS-ie był pisany z myślą o obsłudze szeregowych drukarek. Jeśli jest on wykorzystywany do przesyłania dużych ilości danych z prędkością transmisji przekraczającą 2,400 bodów, to część danych ulega zagubieniu. W rozdziale tym pokazano również, w jaki sposób napisać sterownik, przesyłający dane z prędkością nawet 115,200 bodów. Sterownik ten eliminuje również konieczność użycia specjalnego kabla opisanego w rozdziale 13.
W ostatniej części zamieszczony został przykład rezydentnego programu, który transmituje przez port szeregowy pliki danych. Ilustruje on wiedzę zawartą w poprzednich częściach, a także pokazuje, w jaki sposób uniknąć pułapek związanych z tego typu programami.

Przerwania sprzętowe

Procesor 80x86 wykonuje program instrukcja po instrukcji. Kolejność ta może zostać zaburzona rozkazami **CALL** lub **JMP**, które przekazują sterowanie w inne miejsce pamięci. Jednak ten przepływ sterowania jest w pewien sposób uporządkowany i da się przewidzieć. Jeśli znana jest instrukcja właśnie wykonywana oraz stan znaczników procesora, to można dokładnie określić, który rozkaz zostanie wykonany jako następny.
Istnienie przerwań sprzętowych wnosi w to wszystko element przypadkowości. W każdym bowiem momencie, gdy znacznik zezwolenia na przerwanie ma wartość jeden, może się ono pojawić, zmuszając procesor do wykonania innego fragmentu kodu. Kod ten, zwany procedurą obsługi przerwania (Interrupt Service Routine - ISR), musi odpowiadać zdarzeniu, które spowodowało pojawienie się przerwania oraz zwracać sterowanie do przerwanego programu z niezmienionymi wartościami znaczników i rejestrów.

Przerwania są generowane przez różne urządzenia, które znajdują się na płycie głównej komputera lub są podłączone do jego magistrali. Poprzez wysłanie przerwania każde z urządzeń próbuje zwrócić na siebie uwagę procesora. W komputerach IBM PC obsługą, nadchodzących do systemu przerwań, zajmuje się układ sterownika przerwań 8259. Do jego zadań należy zapewnienie, by żadne z przerwań o niskim priorytecie nie mogło mieć wpływu na przerwania o priorytecie wysokim oraz, by żadne z przerwań, pojawiających się w systemie jednocześnie, nie zostało zignorowane. Zawiera on tablicę przerwań sprzętowych uporządkowaną względem ich priorytetów.
Gdy pojawi się przerwanie, to układ 8259 blokuje wszystkie pozostałe przerwania o niższym priorytecie, dopóki procedura jego obsługi nie wyśle sygnału, który oznacza jej zakończenie. W tabeli zostały zamieszczone przerwania obsługiwane przez układ 8259. Jeśli pojawi się któreś z nich, to układ ten sprawdza, czy aktualnie nie jest obsługiwane przerwanie o wyższym priorytecie. Jeśli nie, wówczas sygnał przerwania, oraz jego numer, jest przekazywany do procesora. Jeśli znacznik zezwolenia na przerwanie jest ustawiony, to CPU kończy wykonywanie bieżącej instrukcji, a następnie wywołuje procedurę obsługi zgłoszonego przerwania. Adresy tych procedur są umieszczone w tablicy wektorów przerwań (IVT), która znajduje się w pamięci pod adresem **0000:0000**. Każdy jej element zawiera segment i przemieszczenie procedury ISR dla każdego z 256 możliwych przerwań. Układ 8259 musi zostać poinformowany o zakończeniu procedury obsługi przerwania. Dopiero wtedy może on obsłużyć następne. Zadnie wysłania do portu 20h (podstawowy 8259) lub 0A0h (dodatkowy 8259) tego układu komendy EOI (End Of Interrupt) spoczywa na procedurze ISR. Polecenie EOI nie musi być wysłane na samym końcu tej procedury. Gdy w procedurze tej zaistnieje możliwość zezwolenia na przyjmowanie innych przerwań, to komenda EOI może zostać wysłana do układu 8259. Kod, który realizuje tę operację wygląda następująco:

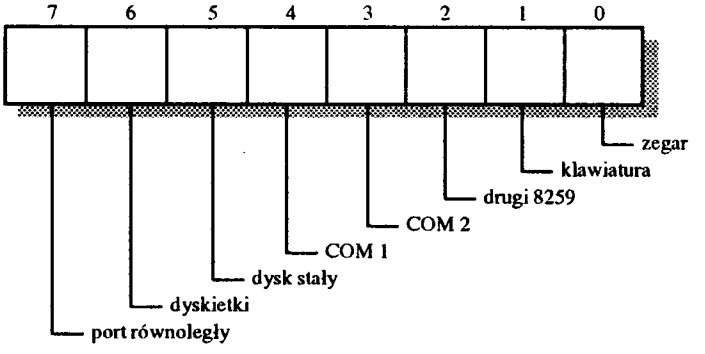
```
mov   al, 20h     ; Prześlij do AL komendę EOI
out   20h, al     ; Wyślij ją do podstawowego układu 8259
```

Tabela 14-1. Kanały przerwań

Kanał	Opis	Numer przerwania
IRQ0	Układ czasowy	08h
IRQ1	Klawiatura	09h
IRQ2	Drugi układ 8259 (tylko komputery AT)	0Ah
IRQ8	Zegar czasu rzeczywistego	70h
IRQ9	Symulowanie IRQ2	71h
IRQ10	Zarezerwowane	72h
IRQ11	Zarezerwowane	73h
IRQ12	Mysz	74h
IRQ13	Wyjątek koprocesora	75h
IRQ14	Sterownik dysku stałego	76h
IRQ15	Zarezerwowane	77h
IRQ3	Szeregowy port 2	0Bh
IRQ4	Szeregowy port 1	0Ch

Kanał	Opis	Numer przerwania
IRQ5	Sterownik dysku stałego	0Dh
IRQ6	Sterownik dysków elastycznych	0Eh
IRQ7	Port równoległy	0Fh

Układ 8259, oprócz rozpoznawania priorytetów i numerów przerwań, może również blokować przyjmowanie określonych przerwań. Dla każdego przerwania istnieje bowiem specjalny znacznik zwany maską przerwania, który oznacza, czy powinno ono zostać przez sterownik zignorowane, czy też nie. Przy instalacji procedury obsługi należy się upewnić, czy bit, odpowiadający danemu przerwaniu, jest wyzerowany tak, aby przerwania rzeczywiście mogły być przyjmowane. Rysunek 14-1 pokazuje układ bitów w masce przerwań. Podprogramy, które wymagają częstej obsługi ze strony procesora, mogą użyć tych bitów do zablokowania części przerwań (na przykład, gdy port szeregowy odbiera dane z prędkością 115,200 bodów). Na obsłużenie wszystkich przerwań, pochodzących od portu szeregowego, zegara i dysku, może bowiem nie starczyć czasu procesora. Poniższy kod blokuje wszystkie przewania oprócz przerwania, które pochodzi od portu szeregowego. Na większości komputerów PC umożliwia to przeprowadzenie takiej transmisji.



Rysunek 14-1. ***Maska przerwań układu 8259.***

```
mov   al, 11101111b     ; Wartość 0 zezwala na przerwania
                        ; związane z kanałem IRQ4 (COM1)
out   21, al            ; Wysłanie maski do układu 8259
```

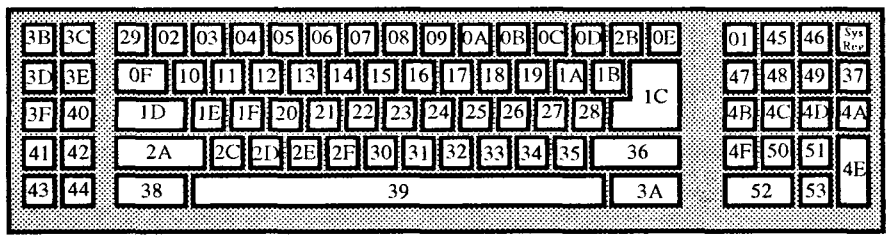
Oczywiście pozostawienie komputera w takim stanie nie jest dobrym rozwiązaniem. Przecież przerwania od wszystkich pozostałych urządzeń, jak klawiatury, twardego dysku czy zegara systemowego, nie będą przyjmowane. Dlatego też trik ten powinien być stosowany na możliwie krótki czas. Przed zmianą maski można z portu **21h** pobrać jej starą wartość, a następnie, po wykonaniu krytycznego odcinka programu, wartość tę przywrócić:

```
            .DATA
OldMask     DB    ?                 ; Poprzednia wartość maski
            .CODE
```

```
in      al, 21h         ; Pobranie bieżącej wartości
                        ; maski
mov     OldMask, al     ; Zapamiętanie jej
mov     al, 11101111b   ; Nowa wartość maski
out     21h, al
.                       ; Krytyczna część programu
.
.
mov     al, OldMask     ; Pobranie starej wartości
                        ; maski
out     21h, al         ; Wysłanie jej do układu 8259
```

Sterownik klawiatury

Każde naciśnięcie lub zwolnienie klawisza powoduje wysłanie przez klawiaturę danej do układu jej sterownika. Generuje on wtedy przerwanie **09h**, które mówi procesorowi, że jest dostępna dana z klawiatury. Zwykle przerwanie to jest obsługiwane przez procedurę, która znajduje się w BIOS ROM. Odczytuje ona daną z portu **60h**. Młodsze siedem bitów pobranej liczby stanowi numer wciśniętego lub zwolnionego klawisza. Na rysunku 14-2 pokazano numery klawiszy typowej klawiatury. Ponadto procedura ta, na podstawie odczytanej danej, uaktualnia stan klawiszy sterujących, a rozpoznane znaki wstawia do bufora klawiatury, które znajdują się w obszarze danych BIOS-u. W programie można pobrać te znaki, wywołując odpowiednią funkcję przerwania programowego **16h**.



Rysunek 14-2. ***Numery klawiszy.***

Można przechwycić przerwanie z klawiatury i umożliwić w ten sposób własnemu programowi reakcję na wciśnięcie pewnych kombinacji klawiszy. Na przykład, po wciśnięciu określonego klawisza na ekranie może pojawiać się system pomocy:

```
; Plik POPUP.ASM
        IDEAL                   ; Składnia trybu IDEAL
        MODEL   SMALL           ; Program będzie działał
                                ; w dowolnym modelu. Musi
                                ; on być tylko jednakowy
                                ; we wszystkich modułach.
        PUBLIC   OldKSR         ; Zmienne te będą
                                ; widoczne we
```

```
        PUBLIC   KeyBrdISR        ; wszystkich modułach
        EXTRN    HelpStuff:PROC   ; Procedura, która
                                  ; wyświetla system pomocy
        DATASEG
KeyDown EQU   03Bh                ; Kod wciśnięcia
                                  ; klawisza F1
OldKSR  DD    ?                   ; Adres poprzedniej
                                  ; procedury obsługi
KeyFlag DB    0                   ; Jeśli system pomocy jest
                                  ; już uaktywniony ma
                                  ; wartość jeden
        CODESEG
PROC    KeyBrdISR   FAR           ; Wszystkie procedury
                                  ; obsługi przerwań
                                  ; są typu FAR
        push  ax                  ; Zachowaj rejestr AX
        push  ds                  ; Zachowaj rejestr DS
        mov   ax, SEG DGROUP      ; Lokalny segment danych
        mov   ds, ax
        in    al, 60h             ; Odczyt portu klawiatury
        cmp   al, KeyDown         ; Czy został wciśnięty
                                  ; klawisz uaktywniający
        je    PopUp               ; Jeśli tak, uaktywnienie
                                  ; programu
DoDeflt:
        pushf                     ; Symulacja wystąpienia
                                  ; przerwania
        call  [OldKSR]            ; Wywołanie poprzedniej
                                  ; procedury obsługi
        pop   ds                  ; Odtworzenie wartości
                                  ; rejestru DS
        pop   ax                  ; Odtworzenie wartości
                                  ; rejestru AX
        iret                      ; Powrót z procedury
                                  ; obsługi przerwania
PopUp:
        cmp   [KeyFlag], 1        ; Czy program jest już
                                  ; aktywny
        je    DoDeflt             ; Jeśli tak wykonanie
                                  ; dotychczasowej procedury
                                  ; obsługi
        mov   [KeyFlag], 1        ; Zaznaczenie aktywności
        sti                       ; Zezwolenie na przerwania
        mov   al, 20h             ; Prześlij do AL komendę
EOI
        out   20h, al             ; Wyślij ją do podstawowego
```

```
                                        ; układu 8259
        call  HelpStuff                 ; Wywołanie procedury,
                                        ; która obsługuje klawisz
        mov   [KeyFlag], 0              ; Wyzerowanie znacznika.
                                        ; Ekran pomocy może być
                                        ; znów wywołany.
        pop   ds                        ; Odtworzenie wartości
                                        ; rejestru DS
        pop   ax                        ; Odtworzenie wartości
                                        ; rejestru AX
        iret
ENDP    KeyBrdISR
END
```

Aby procedura mogła zostać użyta, moduł ten musi zostać skonsolidowany z modułem, który zastępuje wektor przerwania **09h** adresem tej procedury. Powinien on również zawierać podprogram o nazwie **HelpStuff**, który wyświetlałby ekran pomocy. Należy pamiętać, że jest on częścią procedury obsługi przerwania. Dlatego też musi on zachowywać wartości wszystkich rejestrów. Poniższy przykład pokazuje, jak powinna być zainstalowana nowa procedura obsługi oraz zawiera prosty podprogram **HelpStuff**.

```
; Plik LOADPOP.ASM
        IDEAL
        MODEL  SMALL                    ; We wszystkich
                                        ; modułach musi być
                                        ; podany ten sam
                                        ; model pamięci
        EXTRN  OldKSR:DWORD             ; Adres poprzedniej
                                        ; procedury obsługi
        EXTRN  KeyBrdISR:FAR            ; Procedura obsługi
                                        ; przerwania
        PUBLIC HelpStuff                ; Symbol jest
                                        ; widoczny we
                                        ; wszystkich
                                        ; modułach
        DATASEG
ScrSeg  DW     0B800h                   ; Adres segmentu
                                        ; pamięci ekranu
HlpLine1  DB    "+=========================+", 0
HlpLine2  DB    "| Oto ekran pomocy        |", 0
HlpLine3  DB    "+=========================+", 0
MainMsg   DB    "Oto wiadomość", 0Dh, 0Ah, '$'
        CODESEG
; Zainstalowanie nowej procedury obsługi
; i wyświetlenie komunikatu
PROC    Start
        mov    ax, SEG DGROUP
```

```
            mov   ds, ax
            mov   ax, 3509h                ; Pobranie wektora
                                           ; przerwania 09h
            int   21h
            mov   [WORD PTR OldKSR], bx    ; Zapamiętanie
                                           ; przemieszczenia
            mov   [WORD PTR OldKSR+2], es  ; Zapamiętanie
                                           ; numeru segmentu
            push  ds                       ; Zachowanie numeru
                                           ; segmentu danych
            mov   ax, SEG KeyBrdISR
            mov   ds, ax                   ; Załadowanie numeru
                                           ; segmentu procedury
                                           ; obsługi przerwania
            mov   dx, OFFSET KeyBrdISR
            mov   ax, 2509h                ; Ustawienie nowego
                                           ; wektora
                                           ; przerwania 09h
            int   21h
            pop   ds                       ; Odtworzenie numeru
                                           ; segmentu danych
PrnLoop:
            mov   ah, 1                    ; Funkcja BIOS-u,
                                           ; która testuje stan
                                           ; klawiatury
            int   16h
            jnz   ExtLp
                     ; Jeśli został
                                           ; wciśnięty klawisz,
                                           ; wyjście
            mov   ah, 9                    ; Funkcja MS-DOS-u,
                                           ; która wyświetla
                                           ; łańcuch
            mov   dx, OFFSET MainMsg
            int   21h
            jmp   SHORT PrnLoop
ExtLp:
            mov   ah, 0                    ; Odczyt znaku z
                                           ; klawiatury
            int   16h
            mov   ax, 2509h                ; Przywrócenie
                                           ; starego wektora
                                           ; przerwania 09h
            lds   dx, [OldKSR]             ; Pobranie adresu
                                           ; starej procedury
                                           ; obsługi
```

```
          int   21h
          mov   ah, 4Ch                   ; Funkcja, która
                                          ; kończy program
          int   21h
ENDP      Start
; Wyświetlenie ekranu pomocy
PROC      HelpStuff
          push  bx                        ; Zachowanie
                                          ; wartości rejestrów
          push  cx
          push  dx
          push  si
          push  di
          push  bp
          push  es
          mov   es, [ScrSeg]              ; Pobranie numeru
                                          ; segmentu pamięci
                                          ; ekranu
          cld
          mov   si, OFFSET HlpLine1       ; Wyświetlany łań-
cuch
          sub   di, di                    ; Miejsce wyświetle-
nia
          call  PrnStr
          mov   si, OFFSET HlpLine2
          mov   di, 160                   ; Wiersz 2, kolumna
1
          call  PrnStr
          mov   si, OFFSET HlpLine3
          mov   di, 160                   ; Wiersz 3, kolumna
1
          call  PrnStr
          mov   ah, 0                     ; Oczekiwanie na
                                          ; wciśnięcie klawi-
sza
          int   16h
          pop   es
          pop   bp
          pop   di
          pop   si
          pop   dx
          pop   cx
          pop   bx
          ret
ENDP   HelpStuff
; Bezpośrednie wyświetlenie na ekranie łańcucha znaków
```

```
; SI = Adres wyświetlanego łańcucha
; DI = Przemieszczenie w pamięci ekranu
; ES = Segment pamięci ekranu
PROC      PrnStr
PrLoop:
          lodsb                          ; Pobranie jednego
                                         ; znaku z łańcucha
          or    al, al
          jz    Done
          stosb
          inc   di
          jmp   SHORT PrLoop
Done:
          ret
ENDP  PrnStr
          STACK   1024
END  Start
```

Aby utworzyć program wykonywalny o nazwie **LOADPOP**, należy użyć następujących komend:

```
tasm popup
tasm loadpop
tlink loadpop popup
```

Należy zwrócić uwagę na sposób wyświetlania łańcuchów w funkcji **PrnStr**. Zamiast wywołania odpowiedniej funkcji MS-DOS-u, są one bezpośrednio zapisywane do pamięci ekranu. (W rozdziale 15 można znaleźć więcej informacji o tej metodzie). Sposób ten musi zostać zastosowany ze względu na problemy związane z wywołaniem MS-DOS-u z procedur obsługi przerwań. Trudność polega na tym, że pierwszą rzeczą jaką wykonuje MS-DOS, jest przełączenie na swój własny stos. Jeśli teraz pojawi się przerwanie z klawiatury, to każde wywołanie MS-DOS-u z procedury obsługi tego przerwania spowoduje wyzerowanie stosu systemowego. Po zakończeniu podprogramu sterowanie wraca do przerwanego programu (MS-DOS), ale informacja na stosie zostanie już zniszczona. Prawidłowy powrót z funkcji MS-DOS-u nie jest już więc możliwy. Problem ten sprawia, że pisanie programów rezydentnych jest tak trudne.

Port szeregowy

Port szeregowy jest jednym z bardziej uniwersalnych urządzeń w komputerach PC. Można do niego podłączyć terminal, drukarkę, ploter, mysz, modem, inny komputer, a nawet pewne dyski. W większości komputerów IBM PC portem tym steruje układ 8250 lub jego odpowiednik. Procesor komunikuje się z nim za pośrednictwem ośmiu portów we/wy. Porty te są przedstawione na rysunku 14-3 .

Przerwanie **14h** BIOS-u udostępnia kilka funkcji do obsługi portu szeregowego. Funkcje te są używane przez MS-DOS do implementacji urządzeń COM1 i COM2. Podprogramy te sprawdzają rejestr stanu linii w celu stwierdzenia pojawienia się kolejnego znaku lub w celu

Buforowy rejestr przyjęcia znaku - Receiver Buffer Register (DLAB=0)
Tymczasowy rejestr wysyłania znaku - Transmitter Holding Register (DLAB=0)
Młodszy bajt dzielnika - Divisor Low Byte (DLAB=1)

7	0	COM1	COM2
Data		3F8h	2F8h

Rejestr zezwoleń na przerwania - Interrupt Enable Register (DLAB=0)
Starszy bajt dzielnika - Divisor High Byte (DLAB=1)

7	4	3	2	1	0		
Reserved		Modem Status	Receiving Data	Transmit Ready	Data Available	3F9h	2F9h

Interrupt Types

Rejestr identyfikacji przewania - Interrupt ID Register

7	3	2	1	0		
Reserved		Interrupt Type		No Interrupt	3FAh	2FAh

11b = Receiving Data
10b = Data Available
01b = Transmit Ready
00b = Modem Status

Rejestr parametrów transmisji - Line Control Register

7	6	5	4	3	2	1	0		
DLAB	Send Break	Stick Parity	Even Parity	Parity Enable	Stop Bits	Character Size		3FBh	2FBh

Rejestr sterowania modemem - Modem Control Register

7	5	4	3	2	1	0		
Reserved		Loop Back	OUT2	OUT1	RTS	DTR	3FCh	2FCh

Rejestr stanu transmisji - Line Status Register

7	6	5	4	3	2	1	0		
Reserved	Char. Sent	Trans-mitter	Break Detect	Framing Error	Parity Error	Overrun Error	Data Ready	3FDh	2FDh

Rejestr stanu modemu - Modem Status Register

7	6	5	4	3	2	1	0		
Carrier Detect	Ring Indicator	Data Set Ready	Clr to Send	Delta CD	Ring Indicator	Delta DSR	Delta CTS	3FEh	2FEh

Rysunek 14-3. ***Porty we/wy układu 8250.***

sprawdzenia gotowości wysłania następnego. Sposób ten jest dobry przy małych prędkościach transmisji wymaganych np. przez drukarki, ale przy dużych prędkościach dane są często gubione.
Utracie danych można zapobiec, wykorzystując przerwania. Najpierw jednak należy ustawić parametry portu szeregowego tak, by po odebraniu każdego znaku generował on przerwanie. Wtedy procedura ISR musi tylko odczytać daną i zapisać ją do bufora. Dzięki temu nie traci się czasu na sprawdzanie stanu linii portu. Program może więc wykonywać zupełnie inne zadania, a dane pobrać z bufora, gdy tylko się on zapełni.

Inicjacja portu szeregowego.

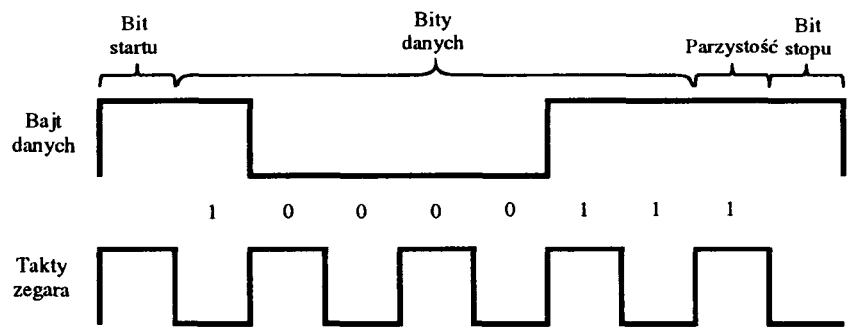
Inicjacja portu szeregowego do pracy z przerwaniami przebiega w dwóch etapach. Najpierw należy przesłać do układu 8250 informację o rodzaju transmitowanych danych, a następnie zainstalować odpowiednią procedurę obsługi przerwania. Poniższy przykład jest procedurą, która dokonuju inicjacji portu szeregowego:

```
; Inicjacja portu szeregowego
; Plik SERINIT.ASM
        IDEAL                   ; Składnia trybu IDEAL
        MODEL   SMALL           ; Program będzie działał
                                ; w dowolnym modelu. Musi
                                ; on być tylko jednakowy
                                ; we wszystkich modułach.
        PUBLIC  InitPort        ; Procedura ta będzie
                                ; widoczna we wszystkich
                                ; modułach
        EXTRN   SerISR:FAR      ; Adres procedury ISR

        DATASEG
; Prędkości transmisji
;               110     150     300     600     1200    2400    4800
Divisor DW      0417h, 0300h, 0180h, 00C0h, 0060h, 0030h,
0018h
;               9600    19200   38400   115200
        DW      000Ch, 0006h, 0003h, 0001h
OldInt  DD      ?
        CODESEG
; Inicjacja portu COM1
; AX = Parametry transmisji
; BX = Prędkość transmisji (1=110, 2=150, 3=300...)
PROC    InitPort
        push    ax              ; Zachowanie parametrów
        mov     dx, 3FBh        ; Rejestr parametrów
                                ; transmisji
        in      al, dx
            ; Pobranie bieżącego stanu
        or      al, 80h         ; Będzie ustawiana
```

```
                                ; prędkość transmisji
        out     dx, al
        mov     dx, 3F8h        ; Młodszy bajt podzielnika
        shl     bx, 1           ; Pomnożenie indeksu przez
                                ; dwa z względu na
                                ; tablicę słów
        mov     al, [BYTE PTR bx+Divisor]
        out     dx, al          ; Wysłanie młodszego bajtu
        inc     dx              ; Starszy bajt podzielnika
        inc     bx
        mov     al, [BYTE PTR bx+Divisor]
        out     dx, al          ; Wysłanie starszego
                                ; bajtu
        pop     ax              ; Odtworzenie parametrów
        mov     dx, 3FBh        ; Rejestr parametrów
                                ; transmisji
        out     dx, al
        mov     dx, 3F8h        ; Odczytanie znaku
        in      al, dx
; Ustawienie procedury obsługi przerwań
        mov     ax, 350Bh       ; Pobranie starego
                                ; wektora przerwania 0Bh
        int     21h
        mov     [WORD PTR OldInt], bx
        mov     [WORD PTR OldInt+2], es
        push    ds
        mov     ax, SEG SerISR  ; Pobranie numeru segmentu
                                ; procedury obsługi
                                ; przerwania
        mov     ds, ax
        mov     dx, OFFSET SerISR ; Pobranie przemieszczenia
                                ; procedury obsługi
                                ; przerwania
        mov     ax, 250Bh       ; Ustawienie nowego
                                ; wektora przerwania 0Bh
        int     21h
        pop     ds
        cli                     ; Wyłączenie przyjmowania
                                ; przerwań
        mov     dx, 3FCh        ; Rejestr sterowania
                                ; modemu
        mov     al, 0Fh         ; Udostępnienie
                                ; przerwania OUT2
        out     dx, al
        sub     dx, 3           ; Rejestr aktywacji
                                ; przerwań
```

```
        mov     al, 1               ; Przerwanie, gdy dana
                                    ; skompletowana w buforze
                                    ; odbiornika
        out     dx, al
        in      al, 21h             ; Pobranie maski przerwań
                                    ; z układu 8259
        and     al, 0EFh            ; Zezwolenie na przerwania
                                    ; od portu COM1
        out     21h, al
        sti
        ret
ENDP    InitPort
END
```

W pierwszej części procedury ustawiane są parametry transmisji. Ich wpływ na postać przesyłanych danych jest widoczny na rysunku 14-4. Ustawienie wartości podzielnika zmienia częstotliwość sygnału zegara. Sygnał ten wyznacza początek i koniec przesyłanych bitów. Liczba taktów tego zegara, które mieszczą się w jednej sekundzie, jest właśnie prędkością transmisji.



Rysunek 14-4. ***Dane szeregowe.***

Rejestr parametrów transmisji układu 8250 opisuje postać przesyłanych znaków (Patrz rysunek 14-3 i 14-4). Długość znaków może się wahać od pięciu do ośmiu bitów. W większości wypadków przesyłane są dane ośmiobitowe, gdyż odpowiadają one rozmiarowi jednego bajtu. Niektóre urządzenia, np. drukarki, wykorzystują tylko siedem bitów, które wystarczają do zakodowania pojedynczego znaku ASCII.

Za częścią, która zawiera dane, może wystąpić bit parzystości. Jest on używany do sprawdzania poprawności transmisji. W różnych urządzeniach bit ten może być różnie interpretowany. Część z nich nie używa tego bitu wcale, inne zawsze go ustawiają lub zerują. Jeśli w danej (razem z bitem parzystości) liczba bitów równych jeden jest parzysta, to mówi się, że dana ma parzystość parzystą. W przeciwnym zaś razie ma ona parzystość nieparzystą. Jeśli bit parzystości, odebrany przez odbiornik, różni się od oczekiwanego, wówczas dana uznawana jest za błędną.

Ostatnią część przesyłanego bajtu stanowi jeden lub kilka bitów stopu. W większości urządzeń stosowany jest jeden bit, który oznacza koniec transmitowanej danej. Niektóre starsze urządzenia mogą używać dwóch lub 1.5 bitu stopu. Po przesłaniu bitów stopu, sygnał danej przyjmuje poziom niski, aż do pojawienia się następnego znaku.
W drugiej części procesu inicjacji włączane są przerwania, które pochodzą od portu szeregowego oraz ustawiana jest procedura obsługi przerwań. Pierwszym krokiem jest uaktualnienie odpowiedniego elementu tablicy wektorów przerwań tak, by wskazywał na procedurę ISR. Operacja ta musi zostać wykonana, zanim zostaną udostępnione przerwania od portu szeregowego.
Następnie należy ustawić port w tryb, w którym generuje on przerwanie po każdym odebraniu danej. W tym celu do rejestru sterowania modemu (patrz rysunek 14-3) wpisywana jest dana, która udostępnia sygnał przerwania OUT2. Natomiast do rejestru przerwań wysyłany jest bajt, identyfikujący przyczynę powstania błędu. W naszym wypadku układ 8250 wygeneruje przerwanie po każdorazowym odebraniu znaku. Pozostałe trzy przyczyny to: gotowość do wysłania bajtu, wykrycie błędu i zmiana stanu linii modemu.
Ostatnią czynnością jest odblokowanie w układzie 8259 przerwań, które dotyczą portu szeregowego. Operacja ta jest przeprowadzana poprzez wyzerowanie odpowiedniego bitu w rejestrze maski tego układu. Dopiero teraz, instrukcją STI, można zezwolić procesorowi na przyjmowanie przerwań. Tak więc, po każdym odebraniu znaku wywoła on procedurę ISR.

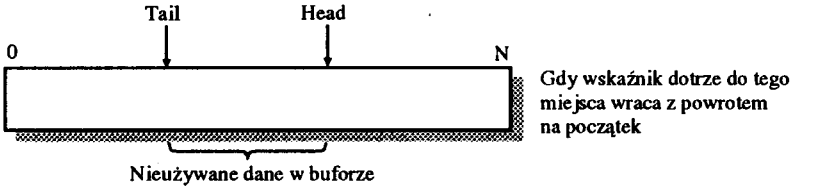
Procedura obsługi przerwań dla portu szeregowego

Poniższy podprogram jest przykładową procedurą obsługi przerwań, które pochodzą od portu szeregowego. Procedura odczytuje z niego daną i umieszcza ją w buforze.

```
; Plik SERISR.ASM
        IDEAL                   ; Składnia trybu IDEAL
        MODEL   SMALL           ; Program będzie działał w
                                ; dowolnym modelu. Musi on
                                ; być tylko jednakowy we
                                ; wszystkich modułach.
        PUBLIC  Head, Tail, Buffer, SerISR

        DATASEG
Head    DW   0                  ; Pozycja w buforze, pod
                                ; którą będzie zapisana
                                ; odczytana dana
Tail    DW   0                  ; Pozycja w buforze, spod
                                ; której dana zostanie
                                ; pobrana
Buffer  DB   1024 DUP (0)
        CODESEG
PROC    SerISR  FAR             ; Wszystkie procedury ISR
                                ; są typu FAR
        push  ax                ; Zachowanie rejestrów
        push  dx
        push  bx
```

```
        push  ds
        mov   ax, SEG DGROUP    ; Pobranie numeru
                                ; segmentu danych
        mov   ds, ax
        mov   dx, 3F8h          ; Rejestr danych układu 8250
        in    al, dx            ; Pobranie znaku
        mov   bx, [Head]        ; Pobranie adresu, pod
                                ; który zostania dana
                                ; przesłana
        mov   [Buffer+bx], al   ; Zapisanie danej
        inc   bx                ; Następna komórka bufora
        and   bx, 03FFh         ; Złączenie początku
                                ; i końca bufora
        mov   [Head], bx        ; Zapisanie nowej pozycji
                                ; w buforze
        pop   ds                ; Odtworzenie wartości
                                ; rejestrów
        pop   bx
        pop   dx
        mov   al, 20h           ; Komenda EOI dla
                                ; układu 8259
        out   20h, al
        pop   ax
        iret
ENDP    SerISR
END
```

W powyższej procedurze został zastosowany *bufor okrężny*. Dzięki temu nowe dane mogą być dodawane w jednym miejscu, a odczytywane w innym, bez żadnych dodatkowych operacji kopiowania. Ilustruje to rysunek 14-5.



Rysunek 14-5. ***Bufor okrężny.***

Zmienna **Head** zawiera przemieszczenie następnej wolnej pozycji w buforze, w której to pozycji zostanie umieszczony odebrany znak. Po jego zapisaniu wartość zmiennej zostaje zwiększona o jeden. Przekroczenie rozmiaru bufora powoduje natomiast ponowne ustawienie tego wskaźnika na jego początek. Bufor staje się więc pierścieniem.
Następny podprogram używa zmiennej **Tail** do pobierania danych z bufora. Jeśli bufor jest pusty, to wartości zmiennych **Tail** i **Head** są sobie równe. Wtedy funkcja **GetByte** czeka

na zmianę wartości **Head**. Następnie odczytuje daną, wskazywaną przez **Tail**, oraz odpowiednio modyfikuje ten wskaźnik. Jeśli po tej operacji wartości obu zmiennych się zrównają, wówczas oznaczać to będzie, że bufor jest już pusty.

```
         IDEAL                          ; Składnia trybu IDEAL
         MODEL  SMALL
         EXTRN  Head:WORD, Tail:WORD, Buffer:BYTE
         PUBLIC TimeOut
         DATASEG
TimeOut  DW     5641                    ; Czas oczekiwania
                                        ; 5 minut
         CODESEG
; Pobranie bajtu z portu szeregowego i zwrócenie go
; w rejestrze AL. Jeśli dana nie pojawi się w ciągu
; 5 minut, to ustawiany jest znacznik przeniesienia.
PROC     GetByte
         push es                        ; Zachowanie wartości
         push cx                        ; rejestrów
         push si
         mov  si, [Tail]                ; Pobranie adresu
                                        ; końca bufora
         mov  ax, 40h                   ; Numer segment danych
         mov  es, ax                    ; BIOS-u
         mov  ax, [WORD PTR es:6Ch]     ; Pobranie czasu
         mov  cx, [WORD PTR es:6Eh]
         add  ax, [TimeOut]             ; Dodanie czasu
         adc  cx, 0                     ; oczekiwania
GBloop:
         cmp  si, [Head]                ; Czy w buforze są nie
                                        ; pobrane znaki
         jne  GotOne                    ; Skok będzie wykonany
                                        ; gdy istnieje znak
         cmp  cx, [WORD PTR es:6Eh]
         ja   GBloop                    ; Jeśli nie został
                                        ; przekroczony czas
         cmp  ax, [WORD PTR es:6Ch]     ; oczekiwania, ponów
                                        ; próbę pobrania znaku
         ja   GBloop
         stc                            ; Przekroczony czas
         jmp  SHORT GBDone              ; oczekiwania
GotOne:
         mov  al, [Buffer + si]         ; Pobierz daną
         inc  si                        ; Następny znak
         and  si, 3FFh                  ; Modulo 1024
         mov  [Tail], si                ; Zapisz nową wartość
                                        ; wskaźnika
```

```
        clc                         ; Znak został
                                    ; prawidłowo odczytany
GBDone:
        pop  si                     ; Odtworzenie wartości
        pop  cx                     ; rejestrów
        pop  es
        ret
ENDP    GetByte
END
```

Zauważmy, że zmienna **Tail** zachowuje się w ten sam sposób, jak zmienna **Head**. Dzięki temu dane są pobierane z bufora w kolejności ich zapisywania. Zrównanie się wartości obu zmiennych oznacza, że bufor jest aktualnie pusty. Każda procedura, która oczekuje na jakieś zewnętrzne zdarzenie, musi być przygotowana na odczekanie pewnego określonego czasu. W naszym przykładzie po pięciu minutach oczekiwanie zostaje zawieszone i, aby wskazać, że nie został pobrany żaden znak, ustawiany jest znacznik przeniesienia.

Wysyłanie danych

Przed wysłaniem danej należy upewnić się, czy nadajnik w układzie 8250 jest gotowy do transmisji następnego bajtu. W tym celu można ustawić tryb pracy portu szeregowego, w którym port generuje przerwanie, gdy jest gotowy do przeprowadzenia operacji. Badając odpowiedni bit w rejestrze stanu transmisji można również określić, czy rejestr nadajnika jest pusty. Jeśli przerwanie zostanie wykorzystane zarówno do stwierdzenia odebrania danej jak i gotowości nadajnika, to procedura jego obsługi musi zostać odpowiednio zmodyfikowana. W celu wykrycia przyczyny przerwań konieczne staje się w niej sprawdzenie rejestru identyfikacji przerwań.
Prostszym sposobem jest sprawdzenie stanu rejestru transmisji. Sposób ten został wykorzystany w poniższej procedurze:

```
; Wysłanie bajtu z rejestru AL do portu szeregowego
SendByte PROC
         push ax            ; Zachowanie wysyłanego bajtu
         mov  dx, 3FDh      ; Rejestr stanu transmisji
SBLoop:
         in   al, dx
         test al, 20h       ; Sprawdzenie bitu rejestru
                            ; nadajnika
         jz   SBLoop        ; Czekaj na gotowość
         pop  ax            ; Odtworzenie wysyłanego bajtu
         mov  dx, 3F8h      ; Rejestr danych
         out  dx, al        ; Wysłanie bajtu
         ret
SendByte ENDP
```

Czas oczekiwania w pętli **SBLoop** nie będzie zbyt długi. Dzieje się tak dlatego, że układ 8250 zawsze wysyła daną z rejestru nadajnika do rejestru szeregującego i zeruje bit gotowości do transmisji nowego znaku.

Przesyłanie plików

Jednym z zastosowań podprogramów, przedstawionych w tym rozdziale, jest przesyłanie plików pomiędzy dwoma komputerami. Aby wysłać plik, należy najpierw odczytać go z dysku, a następnie, wykorzystując procedurę SendByte, przetransmitować każdy jego bajt. Na drugim komputerze, za pomocą procedur obsługi przerwania i GetByte, można odbierać kolejne bajty i zapisywać je na dysk. Taki sposób, niestety, nie daje żadnej gwarancji, że dane odbierane i dane wysłane będą identyczne.
W poniższym programie do przesyłania plików został zastosowany protokół XMODEM. Zapewnia on, że plik odebrany będzie taki sam jak wysłany.

```
; Program do transmisji plików
          .MODEL SMALL
PUBLIC    DwnLoad
EXTRN     SendByte:PROC
EXTRN     GetByte:PROC
EXTRN     TimeOut:WORD
SOH       EQU  1
STX       EQU  2
EOT       EQU  4
ACK       EQU  6
NAK       EQU  15h
CAN       EQU  18h
          .DATA
State     DB   0                    ; Stan programu
BlkNumb   DB   1                    ; Numer odbieranego bloku
ErrCnt    DB   0
jtab      DW   InitSt, GetBNmb
          DW   GetBlk, VerifyB
Buffer    DB   128 dup(0)
FHand     DW   ?
LineSt    DB   ?                    ; Stan parametrów
                                    ; transmisji
          .CODE
; Argumentem tej funkcji jest podany w rejestrach DS:DX
; wskaźnik nazwy transmitowanego pliku
DwnLoad   PROC
          mov  ah, 3Ch              ; Funkcja, która tworzy
          sub  cx, cx               ; nowy plik
          int  21h
          mov  FHand, ax            ; Zapamiętanie deskryp-
                                    ; tora pliku
          mov  dx, 3FBh             ; Rejestr parametrów
                                    ; transmisji
          in   al, dx               ; Pobranie bieżących
                                    ; wartości
          mov  LineSt, al           ; Zapamiętanie ich
```

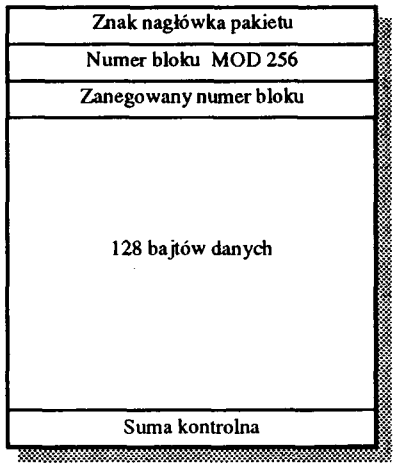
```
        mov  al, 00000011b      ; Ustawienie 8N1
        out  dx, al
DLloop1:
        mov  bl, State          ; Wykorzystanie bieżącego
        sub  bh, bh             ; stanu programu do
        shl  bx, 1              ; wybrania funkcji
        call [jtab+bx]          ; z tablicy jtab
        cmp  ErrCnt, 10
        jb   DLloop1            ; Wykonuj pętlę, gdy
                                ; wystąpiło mniej niż 10
                                ; błędów
        mov  dx, 3FBh           ; Rejestr parametrów
                                ; transmisji
        mov  al, LineSt         ; Przywrócenie poprzed-
                                ; nich parametrów
        out  dx, al
        ret
DwnLoad ENDP
; Inicjacja transmisji
InitSt  PROC
        mov  al, NAK            ; Wysłanie znaku NAK do
        call SendByte           ; nadajnika
        mov  TimeOut, 100       ; Czas oczekiwania < 6
                                ; sekund
        call GetByte            ; Pobranie odpowiedzi
                                ; nadajnika
        cmp  al, SOH            ; Czy to znak SOH
        je   NS1                ; Jeśli tak, następny
                                ; stan
        inc  ErrCnt             ; Zwiększ licznik błędów
        ret
NS1:
        inc  State              ; Następny stan
        mov  ErrCnt, 0          ; Wyzerowanie licznika
        ret                     ; błędów

InitSt  ENDP
; Pobranie numeru bloku
GetBNmb PROC
        mov  TimeOut, 50        ; Czas oczekiwania < 3
                                ; sekundy
        call GetByte            ; Pobierz numer bloku
        jc   GBNErr             ; Jeśli został przekro-
                                ; czony czas oczekiwania,
                                ; to skocz do GBNErr
        push ax                 ; Zachowaj numer bloku
```

```
        call GetByte            ; Pobierz numer testowy
        pop  bx                 ; Odtworzenie numeru
                                ; bloku
        jc   GBNErr             ; Jeśli został przekro-
                                ; czony czas oczekiwania,
                                ; skocz do GBNErr
        not  al                 ; Negacja numeru
                                ; testowego
        cmp  al, bl             ; Jeśli nie są takie
                                ; same, to błąd
        jne  GBNErr             ; Jeśli nie są takie
                                ; same, to skocz do
                                ; GBNErr
        cmp  bl, BlkNumb        ; Porównanie ze spodzie-
                                ; wanym numerem bloku
        jne  GBNErr             ; Jeśli nie są takie
                                ; same, to skocz do
                                ; GBNErr
        inc  State              ; w przeciwnym razie
                                ; przejście do następnego
                                ; stanu
        ret
GBNErr:
        jmp  ErrFnd
GetBNmb ENDP
; Pobranie 128 bajtów danych
GetBlk  PROC
        mov  TimeOut, 18        ; Czas oczekiwania < 1
                                ; sekundy
        mov  cx, 128            ; Liczba spodziewanych
                                ; bajtów
        mov  di, OFFSET Buffer  ; Pobierz adres bufora
        push ds                 ; ES = DS
        pop  es
        cld
GBLoop:
        call GetByte            ; Pobranie bajtu danych
        jc   GBErr              ; Jeśli został przekro-
                                ; czony czas oczekiwania,
                                ; to skocz do GBErr
        stosb                   ; Zapisanie bajtu
        loop GBLoop             ; Pobierz 128 bajtów
        inc  State              ; Przejdź do weryfikacji
                                ; danych
        ret
GBErr:
```

```
        jmp  ErrFnd
GetBlk  ENDP
; Sprawdzenie sumy kontrolnej
VerifyB PROC
        mov  TimeOut, 50
        call GetByte            ; Pobierz sumę kontrolną
        jc   VerErr             ; Jeśli został przekro-
                                ; czony czas oczekiwania,
                                ; to skocz do VerErr
        sub  dl, dl             ; Wyzeruj rejestr DL
        mov  bx, OFFSET Buffer
        mov  cx, 128            ; Rozmiar bufora
VerLoop:
        add  dl, BYTE PTR [bx]  ; Zsumowanie wszystkich
                                ; bajtów
        inc  bx
        loop VerLoop
        cmp  al, dl
        je   VerOK
VerErr:
        jmp  ErrFnd
VerOK:
        mov  ah, 40h            ; Zapis do pliku
        mov  bx, FHand
        mov  cx, 128
        mov  dx, OFFSET Buffer
        int  21h
        mov  al, ACK            ; Potwierdzenie
                                ; odebrania bloku
        call SendByte
        call GetByte            ; Czy pozostał jeszcze
                                ; nie odebrany blok
        jc   VerErr
        inc  BlkNumb
        cmp  al, EOT
        jne  NxtBlk
        mov  al, ACK            ; Potwierdzenie końca
                                ; pliku
        call SendByte
        mov  ErrCnt, 20         ; Przepełnienie licznika
                                ; błędów w celu
                                ; zakończenia programu
NxtBlk:
        mov  State, 1           ; Pobranie następnego
                                ; bloku
        ret
```

```
VerifyB  ENDP
; Obsługa błędów
ErrFnd   PROC
         mov  TimeOut, 10
EF1:                                ; Opróżnienie bufora
         call GetByte
         jnc  EF1                   ; przez pobieranie
                                    ; znaków, aż do przekro-
                                    ; czenia czasu
                                    ; oczekiwania
         mov  TimeOut, 50
         mov  al, NAK
         call SendByte
         inc  ErrCnt
         call GetByte               ; Czy pozostał jeszcze
                                    ; nie odebrany blok
         mov  State, 1              ; Ponownie stan
                                    ; początkowy
         ret
ErrFnd   ENDP
END
```

Protokół XMODEM wymaga, by przesyłany plik został podzielony na 128 bajtowe części zwane pakietami. Rysunek 14-6 przedstawia format takiego pakietu. Komputer, który wysyła dane kończy każdą ich porcję bajtem sumy kontrolnej. Jeśli suma, wyliczona przez komputer odbierający, zgadza się z sumą wysłaną, to nadaje on potwierdzenie odbioru. W przeciwnym razie przesyłane jest żądanie ponownej transmisji bloku. Po przesłaniu wszystkich bloków nadawca wysyła znak EOT (End Of Transmission). Odbiorca potwierdza jego przyjęcie i zamyka plik.



Rysunek 14-6. ***Format pakietu w protokole XMODEM.***

Podsumowanie

Wykorzystanie języka asemblera jest najlepszą, a czasami nawet jedyną, metodą oprogramowania sprzętu. W większości języków wysokiego poziomu nie istnieją bowiem instrukcje wymagane przy dostępie do urządzeń takich jak, np. sterownik przerwań 8259. Inne, jak np. sterownik klawiatury czy portu szeregowego, mimo wszystko wymagają programowania na niskim poziomie.

Układ sterownika przerwań 8259 zajmuje się obsługą przerwań sprzętowych. Informuje on procesor o ich pojawieniu się oraz ustala ich przyczynę. Dopóki nie zostanie zakończona obsługa już przyjętego przerwania, to układ 8259 nie zezwala na przyjmowanie innych. Każda procedura obsługi przerwania musi więc wysłać do sterownika komendę EOI, która powoduje wznowienie przyjmowania innych przerwań. Inną cechą tego układu jest możliwość zablokowania określonych przerwań. Pozwala to na większą kontrolę czasu procesora.

Sterownik klawiatury reaguje na każde wciśnięcie i zwolnienie klawisza. Jego numer dostępny jest poprzez port o numerze **60h**. Ponadto, każde zdarzenie na klawiaturze powoduje wygenerowanie przerwania. Cecha ta może zostać wykorzystana do aktywowania rezydentnych programów, po wciśnięciu określonej kombinacji klawiszy.

Do portu szeregowego można podłączyć wiele urządzeń. Mogą to być drukarki, plotery, modemy czy inne komputery. Do obsługi wolnych urządzeń można wykorzystać funkcje oferowane przez BIOS i MS-DOS. Jeśli jednak wymagana jest duża szybkość komunikacji, wówczas do odbierania danych należy wykorzystać przerwania. Sposób ten zapewnia, że każda nowo pojawiająca się dana zostanie odczytana.

Rozdział 15 *Karta graficzna: tryb tekstowy i graficzny*

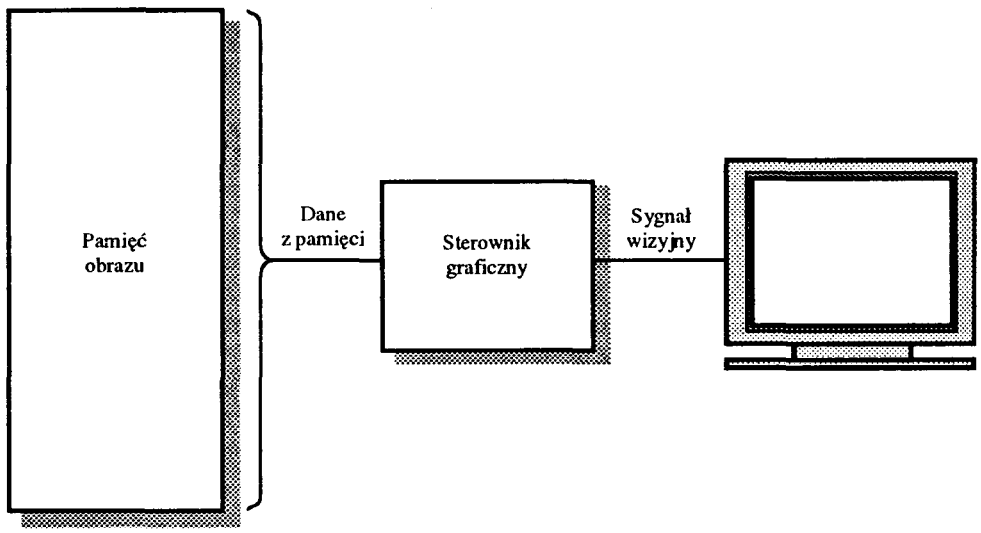
Każdy, kto kiedykolwiek próbował przedstawić pewną ideę, wie, że dobra prezentacja jest równie ważna, jak sama idea. Odnosi się to również do programów komputerowych. Najbardziej wyrafinowany program, ale bez chociażby nawet prostego interfejsu, uwydatniającego jego zalety, nie jest nic wart.
Do tej pory zostało już przedstawione kilka prostych przykładów, które dotyczą wyświetlania na ekranie (na przykład procedura, która rysuje ramkę w Rozdziale 9 lub okrąg w Rozdziale 12). W pierwszej części tego rozdziału zamieszczonych jest więcej szczegółów, wyjaśniających ich działanie, a także zostały przedstawione sposoby adaptacji tych funkcji na inne typy urządzeń wyświetlających.
W ostatniej części został poruszony temat trybów graficznych i ich zastosowanie. Pierwszy przykład pokazuje sposób rysowania na ekranie poszczególnych punktów, które yo punkty mogą być następnie łączone w linie. Ostatni program jest ciekawym przykładem, ilustrującym zasadę animacji.

Sprzęt

W komputerach IBM PC sprzęt wyświetlający składa się z dwóch części: karty graficznej oraz monitora. Karta umieszczona jest wewnątrz komputera. Zawiera ona układy sterujące oraz pamięć video. Monitor to ekran, który wyświetla tekst lub grafikę. Karta graficzna powoduje zamianę danych, umieszczonych w jej pamięci, na serię punktów na ekranie, zwanych pikselami. Komponenty te są przedstawione na rysunku 15-1.

Karty graficzne

Karta graficzna zawiera układy, które sterują wyświetlaniem danych na ekranie. Może ona interpretować pamięć video jako tekst, który musi zostać przekształcony na piksele, lub jako tablicę pikseli. Steruje ona również liczbą znaków lub punktów w linii oraz ilością linii na ekranie. W komputerach IBM istnieje pięć głównych typów kart graficznych: MDA (Monochrome Display Adapter), CGA (Color Graphics Adapter), EGA (Enhanced Graphics Adapter), MCGA (Multi Color Graphics Array) i VGA (Video Graphics Array). Każda z nich może wyświetlić więcej pikseli i w większej liczbie kolorów niż poprzednia.
Najprostszą kartą jest karta MDA. Tekst może być wyświetlany tylko w jednym kolorze na ekranie o formacie 80x25. Każdy znak może mieć swoje atrybuty, takie jak wyświetlenie normalne lub w inwersji, migotanie, rozjaśnienie lub podkreślenie. Następnym krokiem było pojawienie się karty CGA. Tekst może być wyświetlany w formacie 80x25 lub 40x25. Zamiast atrybutów każdy znak może być wyświetlony w różnym kolorze. Oprócz trybów tekstowych karta ta posiada również dwa tryby graficzne: 320x200 pikseli w czterech kolorach lub 640x200 w dwóch kolorach. W karcie EGA istnieją dodatkowo jeszcze cztery nowe tryby graficzne: 320x200 w 16 kolorach, 640x200 w 16 kolorach, monochromatyczny



Rysunek 15-1. ***Wyświetlanie danych na ekranie.***

tryb 640x350 oraz 640x350 w czterech kolorach. Niektóre karty EGA pozwalają na rozszerzenie pamięci video, dzięki czemu, również w trybach wysokiej rozdzielczości, może być używane 16 kolorów. W karcie MCGA dodany został tylko jeden tryb: 640x480 w dwóch kolorach. Wreszcie karta VGA posiada jeszcze dwa dodatkowe tryby: 640x480 w 16 kolorach oraz 320x200 w 256 kolorach.
Wielu producentów oferuje karty zgodne z kilkoma kartami IBM. Mogą one również posiadać swoje własne tryby o większej rozdzielczości i liczbie możliwych kolorów. Bardziej powszechne to karta HGC (Hercules Graphics Card) - przeznaczona dla monitorów monochromatycznych oraz Super-VGA, która zapewnia osiąganie wysokich rozdzielczości. Ponieważ tryby te nie są standardowe, to wraz z kartami dostarczane są sterowniki, które pozwalają na wykorzystanie ich dodatkowych zalet.

Monitory

Zastosowany monitor musi odpowiadać użytej karcie graficznej. Karta MDA wymaga monitora monochromatycznego, CGA - kolorowego, a VGA kolorowego monitora o wysokiej rozdzielczości. Istnieją dwa typy monitorów, które mogą współpracować z jedną lub kilkoma rodzajami kart. Monitory PS/2 mają możliwość poinformować kartę o swoim typie tzn. czy są monochromatyczne czy kolorowe. Wtedy może się ona dostosować do rodzaju monitora. Mogą one współpracować z kartami MCGA lub VGA. Natomiast inne monitory analizują dochodzący do nich sygnał i to one się do niego dopasowują.
Typ monitora nie ma wpływu na sposób programowania karty graficznej, ale od niego zależą kolory obrazu (lub ich brak). Dlatego dobrze jest umożliwić użytkownikowi programu zmianę kolorów używanych w nim.

Programowanie karty graficznej

Wiele programów wyświetla swoje wyniki, korzystając tylko z funkcji DOS-u. Przykładem może być program, który wyświetla zawartość katalogu. Wiele pożytecznych komend, dotyczących ekranu, dostarcza plik ANSI.SYS (Patrz rozdział 13). Dzięki nim, np. katalog może być wyświetlony w określonym miejscu na ekranie lub za pomocą kursora można zaznaczać określone pliki. Wadą tej metody jest to, że wysłana komenda przechodzi najpierw przez DOS, potem jest zamieniana przez ANSI.SYS na wywołania funkcji BIOS-u i dopiero na końcu jej efekt staje się widoczny na ekranie. Jest więc ona czasochłonna.
Na najniższym poziomie karta graficzna może być programowana przez zapisywanie i odczytywanie danych z różnych rejestrów we/wy. Jednak każdy jej typ wykorzystuje różne typy portów, które znajdują się pod różnymi adresami. Nawet karty, które są zgodne na poziomie języków wysokiego poziomu, mogą używać różnych portów. Dlatego też na karcie często znajduje się program BIOS, który ją obsługuje. Musi on zapewniać funkcje, które są zgodne z dostarczonymi przez IBM. Programowanie karty na tym poziomie nie powinno więc sprawiać większych trudności.
Zamieszczone w tym rozdziale przykłady pokazują w jaki sposób można tworzyć efektywne procedury do obsługi ekranu. Wykorzystują one zarówno funkcje BIOS-u, jak i bezpośrednio operują na pamięci video. Funkcje BIOS-u wykonują złożone zadania jak ustawienie określonego trybu karty czy palety kolorów. Bezpośredni dostęp do pamięci video jest najszybszym sposobem wyświetlenia żądanej rzeczy na ekranie.

Tryby graficzne

Istnieje 14 standardowych trybów graficznych, obsługiwanych przez karty IBM. Każdy nowy sterownik może dostarczać kilka nowych. Tryby graficzne i obsługujące je karty, przedstawia tabela 15-1.

Tabela 15-1. Tryby graficzne

Tryb	Karta	Rozdzielczość	Typ	Kolory	Rozmiar znaku	Liczba stron	Bufor
0, 1	CGA	320x200	Tekst	16	8x8	8	B8000h
0, 1	EGA	320x350	Tekst	16	8x14	8	B8000h
0, 1	VGA	360x400	Tekst	16	9x16	8	B8000h
2, 3	CGA	640x200	Tekst	16	8x8	8	B8000h
2, 3	EGA	640x350	Tekst	16	8x14	8	B8000h
2, 3	VGA	720x400	Tekst	16	9x16	8	B8000h
4, 5	CGA	320x200	Grafika	4	8x8	1	B8000h
6	CGA	640x200	Grafika	2	8x8	1	B8000h
7	MDA	720x350	Tekst	MONO	9x14	8	B0000h
7	VGA*	720x400	Tekst	MONO	9x16	8	B0000h
13	EGA	320x200	Grafika	16	8x8	8	A0000h
14	EGA	640x200	Grafika	16	8x8	4	A0000h
15	EGA	640x350	Grafika	MONO	8x14	2	A0000h
16	EGA	640x350	Grafika	16	8x16	2	A0000h

Tryb	Karta	Rozdzielczość	Typ	Kolory	Rozmiar znaku	Liczba stron	Bufor
17	VGA	640x480	Grafika	2	8x16	1	A0000h
18	VGA	640x480	Grafika	16	8x16	1	A0000h
19	VGA	320x200	Grafika	256	8x8	1	A0000h

Późniejsze karty mogą obsługiwać wszystkie tryby wcześniejsze, z wyjątkiem trybu 7.
* Karta VGA może użyć trybu 7, jeśli dołączony jest monitor monochromatyczny.

Najprostszym sposobem ustawienia trybu graficznego jest zlecenie tego zadania programowi BIOS. Wszystkie funkcje, dotyczące grafiki, wywoływane są przerwaniem 10h. Funkcja, która ustawia określony tryb, ma natomiast numer 0. W części tego rozdziału, opisującej tryby, zostanie przedstawiony odpowiedni przykład.

Palety kolorów

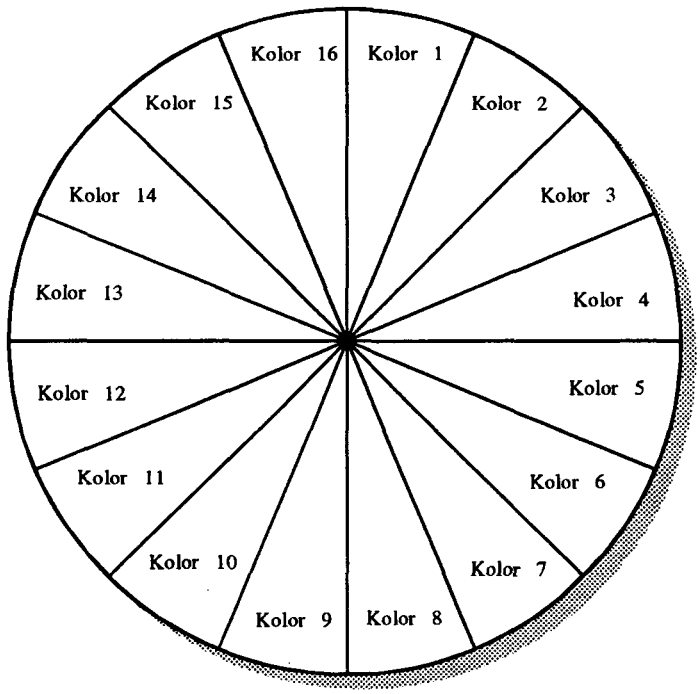
Karty EGA i VGA mogą wyświetlać szeroki zakres kolorów. EGA ma możliwość wyświetlenia 256, a VGA 262,144 kolorów. Żadna z nich nie może jednak ich wszystkich wyświetlić jednocześnie. Na karcie EGA w tej samej chwili może być wyświetlonych do 16, a na VGA do 256 kolorów. Umieszczając na ekranie piksel, program musi podać numer jego koloru (0-15 dla karty EGA i 0-255 dla karty VGA). Dopiero na podstawie rejestru palety kolorów zostanie wybrany właściwy kolor tego punktu.
BIOS zawiera funkcje, które pozwalają na zmianę rejestrów palety, a przez to również i kolorów na ekranie. Sposób ten może zostać wykorzystany do stworzenia prostej animacji. Na rysunku 15-2 narysowane jest jest koło, podzielone na wycinki. Każdy z nich wyświetlony jest w innym kolorze. Przesuwając odpowiednio kolory w rejestrze palety, można osiągnąć efekt obracania się tego koła. Do stworzenia takiego koła może zostać wykorzystana procedura z rozdziału 12, rysująca okrąg, oraz zamieszczone w tym rozdziale procedury, rysujące linie. Procedura, która przesuwa rejestr palety kolorów może wyglądać następująco:

```
        .MODEL SMALL            ; Program będzie działał
                                ; w dowolnym modelu
        PUBLIC Animate          ; Funkcja ta będzie
                                ; widoczna we wszystkich
                                ; modułach
        .DATA
PBuff   DB   256 * 3 DUP ( ? )  ; Rejestry palety
                                ; kolorów
Wrap    DB   3 DUP ( ? )        ; Kolor, który zostanie
                                ; umieszczony na końcu
        .CODE
; Funkcja zakłada, że ustawiony został tryb 13h oraz że
; rysunek jest już narysowany
Animate PROC
        mov  ax, SEG DGROUP
        mov  ds, ax
        mov  es, ax             ; Rejestr ES ustawiany
```

```
                                        ; ze względu na funkcje
                                        ; BIOS-u
        mov  dx, OFFSET PBuff           ; Wskazanie na bufor
                                        ; palety
        sub  bx, bx                     ; Początkowy numer
                                        ; koloru
        mov  cx, 256                    ; Liczba kolorów
        mov  ax, 1017h                  ; Funkcja BIOS-u, która
                                        ; pobiera paletę
        int  10h
        cld
AnLoop:
        mov  di, dx                     ; Przesłanie adresu do
                                        ; rejestru DI
        mov  si, di                     ; i do SI
        lodsw                           ; Pobranie dwóch bajtów
        mov  WORD PTR Wrap, ax          ; Zapamiętanie ich
        lodsb                           ; Pobranie następnego
                                        ; bajtu
        mov  Wrap+2, al
        mov  cx, (256 * 3)/2            ; Liczba słów do
                                        ; przesunięcia
        rep  movsw                      ; Przesunięcie każdego
                                        ; koloru o jedną pozycję
        mov  ax, WORD PTR Wrap          ; Pobranie 2 bajtów
                                        ; ostatniego koloru
        stosw                           ; Przesłanie ich
        mov  al, Wrap+2                 ; Pobranie ostatniego
                                        ; bajtu
        stosb                           ; Przesłanie go
        sub  bx, bx                     ; Początkowy numer
                                        ; koloru
        mov  cx, 256                    ; Liczba kolorów
        mov  ax, 1012h                  ; Funkcja BIOS-u, która
                                        ; zapisuje rejestr
                                        ; palety
        int  10h
        mov  ah, 1                      ; Sprawdzenie stanu
                                        ; klawiatury
        int  16h
        jnz  AnLoop                     ; Jeśli nie został
                                        ; wciśnięty klawisz, to
                                        ; powtórz pętlę
        mov  ah, 0                      ; Odczytanie znaku
        int  16h
        ret
```

```
Animate  ENDP
         .STACK 1024
END
```

Przed wywołaniem tej procedury należy najpierw napisać program, który wyświetla koło wraz z kilkoma jego średnicami. Do jego stworzenia można wykorzystać procedurę z rozdziału 12, rysująca okrąg, oraz zamieszczone w tym rozdziale procedury rysujące linie. Jeśli rysunek jest już gotowy, to wywołanie procedury **Animate** wprawi go w ruch.
Pierwszą rzeczą, wykonywaną w procedurze **Animate** jest pobranie rejestru palety kolorów (Patrz rysunek 15-2). Zostaje on umieszczony w tablicy o nazwie **PBuff**. Zadanie pętli **AnLoop** polega na cyklicznej zmianie położenia numerów kolorów w tym rejestrze. Właśnie dzięki temu powstaje złudzenie ruchu.
Podobny efekt może być również uzyskany na karcie EGA. Do pobrania i ustawienia rejestrów palety należy tylko użyć funkcji o numerach 1002h i 1009h.



Rysunek 15-2. ***Wykorzystanie rejestrów palety do stworzenia animacji.***

Ustalenie typu karty graficznej

Przed użyciem karty graficznej musi najpierw zostać rozpoznany jej typ. Określa on między innymi dostępne tryby oraz położenie pamięci video. Poniższy przykład wykrywa typ zainstalowanej karty i odpowiednio ustawia wartości zmiennych **ScrBase** i **VidPage**, które to zmienne będą używane w kilku następnych procedurach.

```
; Ustalenie trybu i położenia pamięci video
          .MODEL SMALL                ; Program będzie
                                      ; działał w dowolnym
                                      ; modelu
          PUBLIC VideoType, VidType   ; Zmienne te będą
          PUBLIC CurMode, ScrBase     ; widoczne we
          PUBLIC VidPage              ; wszystkich modułach
MDA       EQU  0
CGA       EQU  1
EGA       EQU  2
VGA       EQU  3
          .DATA
VidType   DB   ?                      ; Rodzaj karty
                                      ; graficznej
CurMode   DB   ?                      ; Bieżący jej tryb
                                      ; pracy
ScrBase   DW   0B800h                 ; Segment pamięci video
                                      ; w trybie tekstowym
VidPage   DW   0                      ; Przemieszczenie
                                      ; bieżącej strony
                                      ; ekranu
          .CODE
VideoType PROC
          mov  ah, 0Fh                ; Pobranie stanu karty
          int  10h
          mov  CurMode, al            ; Zapisanie bieżącego
                                      ; trybu jej pracy
          sub  bl, bl                 ; Wyzerowanie rejestru
                                      ; BL
          shl  bx, 1                  ; Wyznaczenie adresu
                                      ; strony ekranu
          shl  bx, 1
          shl  bx, 1
          shl  bx, 1
          mov  VidPage, bx
          cmp  al, 7                  ; Czy to tryb 7
          jne  NotMDA
          mov  ScrBase, 0B000h        ; Segment pamięci dla
                                      ; karty MDA
          mov  VidType, MDA
          jmp  SHORT VTDone
NotMDA:
          mov  ax, 1B00h              ; Sprawdzenie, czy
                                      ; działa funkcja
          int  10h                    ; przeznaczona tylko
                                      ; do pracy na
```

```
        cmp  al, 1Bh                ; karcie VGA
        jne  NotVGA
        mov  VidType, VGA
        jmp  SHORT VTDone
NotVGA:
        mov  ax, 40h                ; Numer segmentu danych
                                    ; BIOS-u
        mov  es, ax
        test BYTE PTR es:[87h], 0FFh
        je   NotEGA
        mov  VidType, EGA
        jmp  SHORT VTDone
NotEGA:
        mov  VidType, CGA           ; Musi więc to być
                                    ; karta CGA
VTDone:
        ret
VideoType ENDP
```

Podprogram ten po skompilowaniu może być dołączony do biblioteki, która zawiera różne użyteczne moduły. W programach, wykorzystujących grafikę, może on być użyty do wykrycia typu zainstalowanej karty graficznej.
Pierwszą rzeczą w programie jest pobranie z BIOS-u numeru trybu pracy karty oraz adresu pamięci video. Jeśli jest to tryb 7, to musi to być karta MDA (może to być również MCGA lub VGA, która pracuje w tym trybie). Następny test sprawdza obecność karty VGA. Wywoływana jest w nim funkcja BIOS-u, która działa tylko w BIOS-ie tej karty. Kolejny test wykrywa kartę EGA, sprawdzając zawartość bajtu o adresie **0040:0087**. Jest to młodszy bajt słowa stanu sterownika video. Jego wartość jest różna od **0FFh** tylko na karcie EGA. Jeśli żaden z testów się nie powiedzie, oznacza to, że w systemie obecna jest karta CGA.

Tryb tekstowy

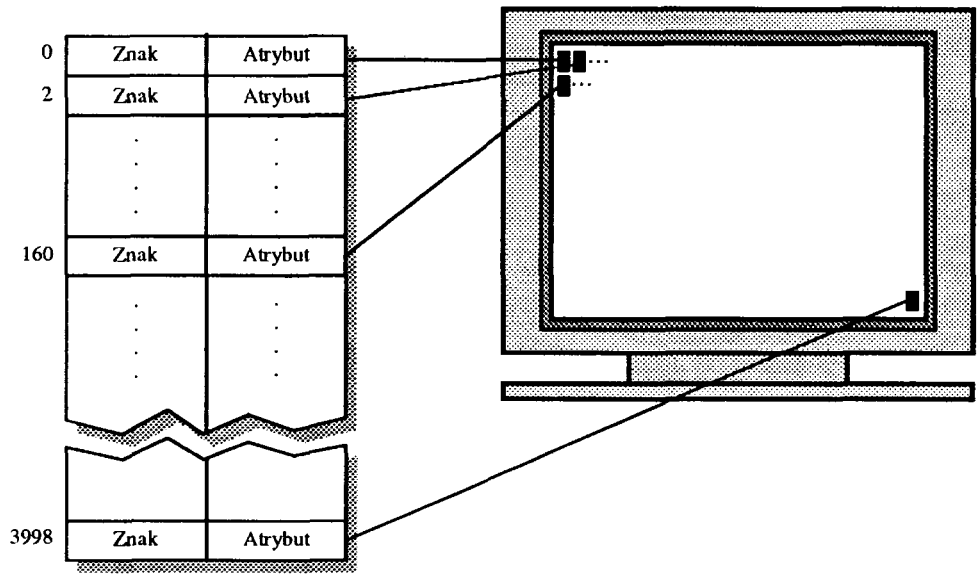
Po ustaleniu położenia pamięci (zmienne ScrBase i VidPage) video wyświetlenie znaku jest już rzeczą prostą. Wystarczy tylko przesłać kod ASCII żądanego znaku do bajtu pamięci o parzystym adresie. Zmiany koloru lub atrybutu wyświetlanego znaku można dokonać, zapisując odpowiednią wartość do bajtu nieparzystego. Rysunek 15-3 pokazuje relację pomiędzy położeniem znaku na ekranie a jego adresem w pamięci.

Atrybuty znaków

Każdy znak w pamięci video posiada swój atrybut. Na karcie MDA określa on takie jego cechy, jak podkreślenie, rozjaśnienie czy migotanie. Na pozostałych kartach używany jest on do ustawienia koloru znaku i tła. Rysunek 15-4 pokazuje format bajtu atrybutu.

Problem śnieżenia na kartach CGA

Pamięć video musi być jednocześnie dostępna dla dwóch procesorów. Jednym z procesorów jest CPU, a drugim sterownik graficzny. Ponieważ układ pamięci może w tym samym czasie



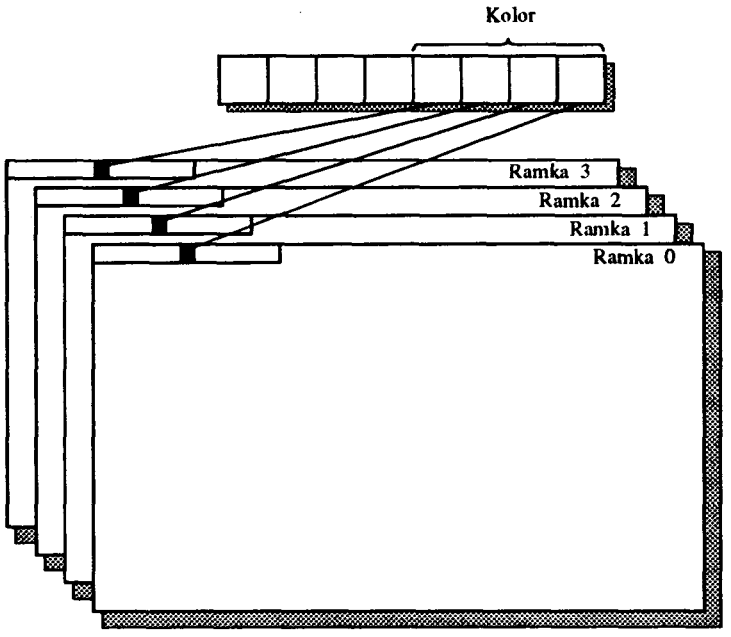
Rysunek 15-3. ***Odwzorowanie pamięci video na ekran.***

udostępnić tylko jedną jej komórkę, jedno z tych urządzeń musi czekać na zakończenie operacji przez drugie. Na niektórych kartach CGA może to powodować pojawienie się śnieżenia na ekranie. Pozostałe karty mają zamontowane specjalne układy, które zapobiegają temu efektowi.

Aby zapobiec efektowi śnieżenia na kartach CGA, operacja zapisu do pamięci video musi być przeprowadzona, gdy sterownik karty z niej nie korzysta. Sytuacja taka występuje po pojawieniu się impulsu odchylania poziomego lub pionowego. Wtedy to wiązka elektronów w monitorze jest przesuwana z jego lewej strony na prawą lub z dołu do góry. Jest ona wtedy wyłączona, a więc na ekranie nie może pojawić się żaden śnieg. Rejestr stanu karty CGA znajduje się pod adresem **3DAh**. Jego najmłodszy bit jest ustawiany podczas impulsu odchylania poziomego. Zastosowanie poniższego kodu zapewnia, że dana będzie zapisana zaraz po jego pojawieniu się:

```
        mov  dx, 3DAh        ; Rejestr stanu karty CGA
RtrcOn:
        in   al, dx          ; Pobranie bajtu stanu
        test al, 1           ; Testowanie bitu powrotu
                             ; wiązki
        jnz  RtrcOn          ; Oczekiwanie na impuls
RtrcOff:
        in   al, dx          ; Oczekiwanie na ponowne
                             ; ustawienie bitu
        test al, 1           ; Zapewnia to wystarczającą
                             ; ilość czasu na przeprowa-
        jz   RtrcOff         ; dzenie żądanej operacji
```

```
.                    ; Operacja dostępu do pamięci
.                    ; video może być już
.                    ; bezpiecznie przeprowadzona
```



Rysunek 15-4. ***Bity ramek karty EGA.***

Kilka podstawowych procedur, działających w trybie tekstowym

Wcześniej przedstawiona procedura VideoType pokazuje sposób określenia położenia pamięci video. Poniższe podprogramy wykorzystują tę informację przy wyświetlaniu tekstu na ekranie.

WYŚWIETLENIE ZNAKU

Podprogram wyświetla znak z rejestru DL w wierszu określonym przez rejestr AL i kolumnie determinowanej przez AH. Jego atrybut znajduje się w rejestrze DH.

```
; Plik PUTCHAR.ASM
        .MODEL  SMALL           ; Program będzie działał
                                ; w dowolnym modelu
        PUBLIC  PutChar         ; Symbol PutChar będzie
                                ; widoczny we wszystkich
                                ; modułach
EXTRN   VidType:BYTE
EXTRN   ScrBase:WORD
EXTRN   VidPage:WORD
```

```
CGA       EQU  1
          .CODE
PutChar   PROC
          mov  es, ScrBase        ; Pobranie adresu bazowego
          sub  bh, bh             ; Wyzerowanie rejestru BH
          mov  bl, al             ; Numer kolumny w BX
          mov  al, ah             ; Numer wiersza do AH
          mov  cl, 80             ; Razy 80 znaków w wierszu
          mul  cl
          add  bx, ax             ; Dodanie do numeru kolumny
          shl  bx, 1              ; Dwa bajty na znak
          cmp  VidType, CGA       ; Czy może wystąpić
                                  ; śnieżenie
          jne  NoSnow
          push dx
          mov  dx, 3DAh           ; Rejestr stanu karty CGA
RtrcOn:
          in   al, dx             ; Pobranie bajtu stanu
          test al, 1              ; Testowanie bitu powrotu
                                  ; wiązki
          jnz  RtrcOn             ; Oczekiwanie na impuls
RtrcOff:
          in   al, dx             ; Oczekiwanie na ponowne
                                  ; ustawienie bitu
          test al, 1              ; Zapewnia to wystarczającą
                                  ; ilość czasu na
          jz   RtrcOff            ; przeprowadzenie żądanej
                                  ; operacji
          pop  dx
NoSnow:
          mov  WORD PTR es:[bx+VidPage], dx
          ret
PutChar   ENDP
END
```

W pierwszej części procedury obliczany jest adres pamięci video, pod który będzie zapisany znak. Wykorzystywana jest przy tym zmienna **ScrBase**, której wartość została nadana przez funkcję **VideoType**. W pierwszej linii jej zawartość jest przepisywana do rejestru ES. Następnie numer wiersza zostaje umieszczony w rejestrze AL i pomnożony przez 80 (liczba znaków w wierszu). Następnie do wyniku jest dodawany numer kolumny. Adres znaku uzyskiwany jest przez pomnożenie go przez dwa. Wpisanie pod ten adres znaku oraz jego atrybutu powoduje już wyświetlenie go na ekranie.

WYŚWIETLENIE ŁAŃCUCHA ZNAKÓW

Jednym ze sposobów wyświetlenia łańcucha znaków jest kolejne ich wysyłanie do funkcji PutChar. Wadą tego podejścia jest to, że za każdym wywołaniem tej funkcji będzie musiał

być na nowo obliczany adres pamięci. Poniższa funkcja PrintStr wyświetla łańcuch, wskazywany przez rejestry DS:SI, w wierszu o numerze zawartym w AH, a kolumnie w AL. Łańcuch musi być zakończony zerowym bajtem. Podobnie jak poprzednio, atrybuty wyświetlanych znaków podane są w rejestrze DH.

```
; Plik PRNSTR.ASM
; Wyświetlenie łańcucha znaków
        .MODEL   SMALL          ; Program będzie działał
                                ; w dowolnym modelu
        PUBLIC   PrintStr
EXTRN   VidType:BYTE
EXTRN   ScrBase:WORD
EXTRN   VidPage:WORD
CGA     EQU  1
        .CODE
PrintStr PROC
        mov  es, ScrBase        ; Pobranie adresu
                                ; bazowego
        sub  bh, bh             ; Wyzerowanie rejestru BH
        mov  bl, al             ; Numer kolumny w BX
        mov  al, ah             ; Numer wiersza do AH
        mov  cl, 80             ; Razy 80 znaków
                                ; w wierszu
        mul  cl
        add  bx, ax             ; Dodanie do numeru
                                ; kolumny
        shl  bx, 1              ; Dwa bajty na znak
        mov  di, OFFSET VidPage ; Adres początku strony
        add  di, bx             ; Wyznaczenie adresu
                                ; pamięci
        mov  ah, dh             ; Atrybut znaków
                                ; w rejestrze AH
        cld
PSLoop:
        lodsb                   ; Pobranie następnego
                                ; znaku łańcucha
        or   al, al             ; Czy to bajt zerowy
        jz   PSDone
        push ax                 ; Zachowanie znaku
                                ; i jego atrybutu
        cmp  VidType, CGA       ; Czy może wystąpić
                                ; śnieżenie
        jne  NoSnow
        mov  dx, 3DAh           ; Rejestr stanu karty CGA
RtrcOn:
        in   al, dx             ; Pobranie bajtu stanu
```

```
        test al, 1          ; Testowanie bitu powrotu
                                          ; wiązki
        jnz  RtrcOn                       ; Oczekiwanie na impuls
RtrcOff:
        in   al, dx                       ; Oczekiwanie na ponowne
                                          ; ustawienie bitu
        test al, 1                        ; Zapewnia to wystarcza-
                                          ; jącą ilość czasu na
        jz   RtrcOff                      ; przeprowadzenie żądanej
                                          ; operacji
NoSnow:
        pop  ax                           ; Odtworzenie znaki
                                          ; i atrybutów
        stosw                             ; Wyświetlenie znaku na
                                          ; ekranie
        jmp  PSLoop
PSDone:
        ret
PrintStr ENDP
END
```

W pierwszej części procedury obliczany jest adres pamięci video, pod którym będą umieszczane znaki łańcucha. Wykorzystywana jest przy tym zmienna **ScrBase** oraz **VidPage**. W pętli **PSLoop** kolejne znaki łańcucha przesyłane są do pamięci video. Do tego celu została zastosowana instrukcja **STOSW**, która jednocześnie modyfikuje wartość wskaźnika ES:DI. Pętla ta kończy się po odczytaniu zerowego bajtu z łańcuch.
Poniższy przykład umożliwia zapoznanie się z działaniem przedstawionych procedur. Najpierw inicjowane są zmienne, a następnie, przy użyciu funkcji PrintStr, wyświetlanych jest kilka łańcuchów:

```
; Plik TEXTTEST.ASM
        .MODEL SMALL                      ; Program będzie działał
                                          ; w dowolnym modelu
EXTRN   PrintStr:PROC
EXTRN   VideoType:PROC
        .DATA
String1 DB "How many programmers does it take to replace"
        DB " a light bulb?", 0
String2 DB "None, that's a hardware problem!", 0
        .CODE
StrTest PROC
        mov  ax, SEG DGROUP               ; Numer segmentu danych
        mov  ds, ax
        mov  si, OFFSET String1           ; Adres pierwszego
                                          ; łańcucha
        mov  dh, 70h                      ; Czarne litery na
                                          ; białym tle
```

```
        mov  ax, 0A0Ah            ; Wyświetlenie na
                                  ; pozycji (10, 10)
        call PrintStr
        mov  ah, 0                ; Testowanie stanu
                                  ; klawiatury
        int  16h
        mov  si, OFFSET String2   ; Adres drugiego
                                  ; łańcucha
        mov  dh, 17h              ; Białe litery na
                                  ; niebieskim tle
        mov  ax, 0A0Ah            ; Wyświetlenie na
                                  ; pozycji (11, 10)
        call PrintStr
        mov  ah, 4Ch
        int  21h
StrTest ENDP
END
```

CZYSZCZENIE EKRANU

Kolejna procedura zapisuje w miejscu ekranu, wskazanym przez rejestry AL i AH, znak i jego atrybut zawarty w rejestrach DL i DH. Czynność ta powtarzana jest CX razy. Jeśli w rejestrze AX zostanie umieszczona wartość 0 (początek ekranu), **0720h** w DX (czarne spacje), a 2000 w CX (liczba znaków na ekranie), to funkcja ta spowoduje wyczyszczenie ekranu.

```
; Plik CLS.ASM
        .MODEL SMALL              ; Program będzie działał
                                  ; w dowolnym modelu
        PUBLIC Fill
EXTRN   VidType:BYTE
EXTRN   ScrBase:WORD
EXTRN   VidPage:WORD
CGA     EQU  1
        .CODE
Fill    PROC
        mov  es, ScrBase          ; Pobranie adresu bazowego
        sub  bh, bh               ; Wyzerowanie rejestru BH
        mov  bl, al               ; Numer kolumny w BX
        mov  al, ah               ; Numer wiersza do AH
        push cx                   ; Zachowanie licznika
                                  ; powtórzeń
        mov  cl, 80               ; Razy 80 znaków w wierszu
        mul  cl
        add  bx, ax               ; Dodanie do numeru
                                  ; kolumny
        shl  bx, 1                ; Dwa bajty na znak
```

```
        pop  cx                 ; Odtworzenie wartości
                                ; licznika powtórzeń
        mov  ax, dx             ; Przesłanie znaku
                                ; i atrybutu do AX
FLoop:
        cmp  VidType, CGA       ; Czy może wystąpić
                                ; śnieżenie
        jne  NoSnow
        push dx
        mov  dx, 3DAh           ; Rejestr stanu karty CGA
RtrcOn:
        in   al, dx             ; Pobranie bajtu stanu
        test al, 1              ; Testowanie bitu powrotu
                                ; wiązki
        jnz  RtrcOn             ; Oczekiwanie na impuls
RtrcOff:
        in   al, dx             ; Oczekiwanie na ponowne
                                ; ustawienie bitu
        test al, 1              ; Zapewnia to wystarcza-
                                ; jącą ilość czasu na
        jz   RtrcOff            ; przeprowadzenie żądanej
                                ; operacji
        pop  dx
NoSnow:
        mov  WORD PTR es:[bx+VidPage], dx
        inc  bx
        inc  bx
        loop FLoop
        ret
Fill    ENDP
END
```

Pierwsza część procedury jest bardzo podobna do poprzednich. Oblicza adres pamięci video, na którym będzie wykonywana operacja. Następnie sprawdzana jest możliwość wystąpienia śnieżenia. Jeśli efekt taki może zaistnieć, to zapis do pamięci video odbywa się dopiero po pojawieniu się impulsu odchylania poziomego. Proces ten jest powtarzany do momentu wyzerowania licznika CX.

Tryby graficzne

Pierwszym krokiem, przy wykorzystywaniu trybów graficznych, jest wybór odpowiedniego trybu. Najczęściej zależy on od typu zainstalowanej karty. Na każdej karcie można wybierać natomiast pomiędzy wyższą rozdzielczością lub większą liczbą dostępnych kolorów. Programy, takie jak CAD/CAM, które tworzą bardzo szczegółowe rysunki, lepiej działają przy dużych rozdzielczościach. Dla innych, na przykład dla programów wyświetlających fotografie, bardziej nadają się tryby, oferujące dużą liczbę kolorów.

W trybie graficznym każdy bajt reprezentuje jeden lub kilka pikseli. W trybie czarno-białym jeden piksel wymaga tylko jednego bitu. Zwiększanie liczby dostępnych kolorów wymaga użycia większej ilości bitów, które odpowiadają jednemu punktowi ekranu. W czterokolorowym trybie karty CGA na jeden piksel przypadają dwa bity, zakodowanie 16 kolorów karty EGA wymaga użycia czterech bitów, zaś 256 -na karcie VGA ośmiu bitów na piksel.

Ustalanie położenia pamięci video

Na karcie CGA pamięć video zaczyna się od adresu **0B8000h**, a kończy na adresie **0BFFFFh**. Adres piksela zależy od użytego trybu. Karty EGA i VGA wymagają więcej niż 32,768 bajtów pamięci. Dlatego też pamięć video na nich zaczyna się od adresu **0A0000h**. Dzięki temu istnieje 131,071 bajtów dostępnej pamięci.
Istnieją dwa sposoby zaadaptowania pamięci do wyświetlania grafiki. Jeden z nich polega na podzieleniu każdego bajtu na kilka pikseli. W trybie dwukolorowym na karcie CGA każdy bajt zawiera osiem pikseli; przy czterech kolorach cztery piksele itd.

PIKSELE W KARCIE CGA

Karta CGA posiada tylko dwa tryby graficzne. W jednym z nich dostępne jest 64,000 pikseli i cztery kolory, a w drugim 128,000 pikseli, ale za to tylko w dwóch kolorach. Aby narysować pojedynczy punkt, najpierw należy wyliczyć adres bajtu, który go zawiera, a następnie ustawić odpowiednie jego bity. Niestety, ze względu na sposób zorganizowania pamięci video, wzór, pozwalający wyznaczyć adres odpowiedniego bajtu, jest dość skomplikowany. Sąsiednie linie na ekranie są bowiem umieszczone w oddzielnych połówkach pamięci. Jednakże dzięki temu karta CGA może wykorzystywać wolne układy pamięci, które pozwalają na osiągniecie gotowości przez układ, zawierający następną linię w czasie wyświetlania bieżącej linii. Technika ta nazywana jest przeplataniem (ang. interlacing).
Jeśli y jest numerem kolumny, w której piksel ma zostać wyświetlony, a x numerem wiersza, to wzór na obliczenie przemieszczenia bajtu w trybie czterokolorowym ma postać:

```
Przemieszczenie = y * 80 + x / 4 + ( y AND 1 ) * 8192
```

Składnik **(y AND 1) * 8192** jest równy 8192 tylko wtedy, gdy numer linii jest nieparzysty. W przeciwnym razie jest on zerem. Zmiana stałej w tej formule pozwala otrzymać wzór dla trybu dwukolorowego:

```
Przemieszczenie = y * 80 + x / 8 + ( y AND 1 ) * 8192
```

Po wyznaczeniu adresu bajtu pozostało tylko wyznaczenie w nim pozycji bitów piksela i zmiana ich wartości. Do zamaskowania niepotrzebnych bitów potrzebne będzie utworzenie odpowiedniej maski. Następnie bity piksela muszą zostać przesunięte na właściwą pozycję za pomocą operacji OR umieszczone w bajcie pamięci. Liczba pozycji, o którą należy wykonać przesunięcie, może być wyznaczona z poniższych wzorów:

```
Tryb dwukolorowy:
     Przesunięcie = x mod 8

Tryb czterokolorowy
     Przesunięcie = ( x mod 4 ) * 2
```

Powyższe informacje zostały wykorzystane do napisania procedury, która wyświetla jeden piksel:

```
; Plik CGAPIX.ASM
          .MODEL  SMALL
          .DATA
          PUBLIC  CGAPixel
ypos      DB   ?                      ; Tymczasowa wartość y
ScrBase   DW   0B800h                 ; Segment pamięci video
          .CODE
; Narysowanie piksela we współrzędnych (AL, AH) w kolorze
; podanym w DL
CGAPixel PROC
          mov  es, ScrBase            ; Segment pamięci video
          mov  cl, 3
          sub  bh, bh                 ; Wyzerowanie starszego
                                      ; bajtu rejestru BX
          mov  bl, ah                 ; Teraz BX = y
          mov  ah, bh                 ; Wyzerowanie starszego
                                      ; bajtu rejestru AX
          mov  ypos, bl               ; Zapamiętanie wartości
                                      ; współrzędnej y
          and  bx, 0FEh               ; Zagubienie wartości
                                      ; najmłodszego bitu
          shl  bx, cl                 ; bx = y razy 8
          mov  cx, bx
          shl  bx, 1
          shl  bx, 1                  ; bx = y razy 32
          add  bx, cx                 ; bx = y razy 40
          mov  cx, ax                 ; Pobranie wartości x
          shr  cx, 1
          shr  cx, 1                  ; Podzielenie x przez 4
IFE  GMODE EQ 6                       ; Dla trybu 6
          shr  cx, 1                  ; Podzielenie x przez 8
ENDIF
          add  bx, cx                 ; Dodanie do łącznego
                                      ; wyniku
          test ypos, 1                ; Czy y jest
                                      ; nieparzyste
          jz   NotOdd
          add  bx, 8192               ; Jeśli y jest niepa-
                                      ; rzyste, użyj drugiej
                                      ; połowy pamięci
NotOdd:
IFE  GMODE EQ 6                       ; Dla trybu 6
          and  ax, 7                  ; Obliczenie wartości
```

```
                                        ; przesunięcia
        mov  cl, 7
        mov  ah, 0FEh                   ; Maska
ELSE
        and  ax, 3                      ; Obliczenie wartości
                                        ; przesunięcia
        shl  ax, 1                      ; Pomnożenie przez 2
                                        ; bity na piksel
        mov  cl, 6
        mov  ah, 0FCh                   ; Maska
ENDIF
        sub  cl, al
        mov  al, BYTE PTR es:[bx]
        rol  ah, cl                     ; Przesunięcie maski na
                                        ; odpowiednią pozycję
        and  al, ah                     ; Zamaskowanie pozosta-
                                        ; łych bitów
        shl  dl, cl                     ; Przesunięcie koloru
                                        ; na właściwą pozycję
        or   al, dl                     ; Wpisanie go do bajtu
        mov  BYTE PTR es:[bx], al
        ret
CGAPixel ENDP
END
```

W pierwszej części podprogramu obliczany jest adres odpowiedniego bajtu pamięci video. Do tego celu zostały użyte wcześniej zaprezentowane wzory. Dzięki zastosowaniu dyrektyw kompilacji warunkowej można stworzyć różne wersje tej funkcji, które działają w trybach 4 i 5 lub 6. W pierwszym bloku wartość x jest dzielona przez 8 zamiast przez 4, jeśli symbol **GMODE** ma wartość 6.

W kolejnej części, tuż przed etykietą **NotOdd**, sprawdzane jest, czy piksel leży w linii o nieparzystym numerze. Jeśli tak, to do adresu bajtu dodawana jest liczba 8192.

Zadaniem następnego bloku warunkowego jest utworzenie odpowiedniej maski. Różnica wartości rejestrów CL i AL jest ilością pozycji, o którą należy przesunąć maskę, znajdującą się w rejestrze AH. Ostatni fragment funkcji pobiera bajt pamięci video, zawierający żądany piksel, wylicza wartość maski, zeruje bity piksela, dokonuje przesunięcia jego nowej wartości na właściwą pozycję, umieszcza ją w bajcie i ponownie zapisuje go do pamięci.

Przy kompilacji tego programu należy podać asemblerowi odpowiednią opcję, która powoduje zdefiniowanie wartości symbolu **GMODE**. Na przykład, aby utworzyć wersję działającą w trybie 6, można użyć następującej komendy:

```
tasm /dGMODE=6 cgapix
```

PIKSELE NA KARCIE EGA

Innym sposobem zaaranżowania pamięci do wykorzystania w grafice jest podzielenie jej na warstwy czy ramki bitowe. Każda ramka jest tablicą pikseli, w której jeden bit odpowiada jednemu pikselowi. Ramki te są umieszczone jedna nad drugą, tak jak to pokazano na

rysunku 15-4. Kolor każdego punktu jest ustalany na podstawie wartości bitów w poszczególnych warstwach. W karcie EGA istnieją cztery takie ramki, które pozwalają na jednoczesne uzyskanie 16 kolorów.
Wadą tej metody jest to, że zapis jednego piksela wymagałby dostępu do każdego z bitów w tych ramkach. Na karcie EGA wymagałoby to aż ośmiu odwołań do pamięci (cztery na pobranie bajtów i cztery na ich zapisanie). Dlatego też problem ten został rozwiązany sprzętowo. Na karcie EGA został dodany specjalny rejestr zatrzaskowy (ang. latch), który przechowuje wartości bitów z każdej ramki. Dzięki temu zapisanie lub odczytanie piksela jest bardzo proste i odbywa się właśnie poprzez rejestr zatrzaskowy. Rejestr ten umożliwia dostęp do pamięci na wiele różnych sposobów. Tak więc, przed narysowaniem piksela należy się upewnić, że pracuje on w trybie 2 (zatrzaśnięcie pojedynczego piksela). Poniższy kod może zostać użyty do wysłania karcie EGA odpowiedniej komendy:

```
mov  dx, 3CEh        ; Rejestr indeksowy karty EGA
mov  ax, 0205h       ; Tryb pracy rejestru i wysyłana dana
out  dx, ax          ; Wyjście 16 bitowe
```

W powyższym fragmencie dana jest wysyłana jednocześnie na dwa sąsiadujące porty. Pierwszy z nich, o adresie **3CEh**, jest rejestrem indeksowym. Umieszczona w nim wartość informuje kartę o typie danej, która znajduje się w rejestrze danych (**3FCh**). Każdy z nich oczywiście może przyjąć tylko jeden bajt. W programie została użyta instrukcja, która wysyła dane do obu rejestrów jednocześnie. Ponieważ niektóre urządzenia nie potrafią zaakceptować danych wysyłanych w ten sposób, to przed zastosowaniem tej metody należy najpierw przeczytać dokumentację karty.
Następnym krokiem jest obliczenie adresu bajtu, który zawiera żądany piksel. Ponieważ karta EGA nie stosuje przeplotu linii, dlatego jest to operacja trochę prostsza niż poprzednia:

```
Dla 320 pikseli w linii (tryb 13):
     Przemieszczenie = y * 40 + x/8

Dla 640 pikseli w linii (tryb 14-16):
     Przemieszczenie = y * 80 + x/8
```

Po wyznaczeniu adresu bajtu, pozostaje jeszcze znalezienie w nim pozycji piksela. Można to zrobić w ten sam sposób jak dla karty CGA, pracującej w trybie dwukolorowym. Reszta z dzielenia numeru kolumny piksela przez 8 jest liczbą pozycji, o jaką należy przesunąć maskę. Przez odczytaniem lub zapisem do pamięci video, maska ta musi zostać wysłana do rejestru zatrzaskowego. Dzięki temu każde odwołanie do bajtu pamięci, który zawiera piksel, powoduje pobranie danej z tego rejestru, która to dana oznacza jego kolor. Jednak aby zatrzask mógł poprawnie pracować, przed zapisaniem bajtu musi on zostać odczytany. Pozwala to na zignorowanie niepotrzebnych bitów, tak jak było to robione przy wyświetlaniu punktu na karcie CGA.
Poniższy podprogram wyświetla piksel na ekranie karty EGA. Przed jego wywołaniem numer kolumny należy umieścić w rejestrze AX, wiersza - w BX, a jego kolor - w DL.

```
; Plik EGAPIX.ASM
          .MODEL  SMALL              ; Program będzie
                                     ; działał w dowolnym
                                     ; modelu
```

```
        PUBLIC  EGAPixel            ; Funkcja ta będzie
                                    ; widoczna we
                                    ; wszystkich modułach
        .DATA
ScrBase DW   0A000h                 ; Segment pamięci video
        .CODE
; Narysowanie piksela we współrzędnych (AX, BX) w kolorze
; podanym w DL
EGAPixel PROC
        mov  es, ScrBase            ; Segment pamięci video
IFE  GMODE EQ 13                    ; Dla trybu 13 pomnoże-
                                    ; nie numeru wiersza
        mov  cl, 3                  ; przez 40
ELSE                                ; w przeciwnym razie
                                    ; przez 80
        mov  cl, 4
ENDIF
        shl  bx, cl                 ; bx = wiersz razy 8
                                    ; lub 16
        mov  cx, bx
        shl  bx, 1
        shl  bx, 1                  ; bx = wiersz razy 32
                                    ; lub 64
        add  bx, cx                 ; bx = wiersz razy 40
                                    ; lub 80
        mov  cx, ax                 ; Pobranie numeru
                                    ; kolumny
        shr  cx, 1
        shr  cx, 1
        shr  cx, 1                  ; Podzielenie go
                                    ; przez 8
        add  bx, cx                 ; Dodanie do łącznego
                                    ; wyniku
        and  ax, 7                  ; Obliczenie wartości
                                    ; przesunięcia
        mov  cl, al
        mov  ah, 080h               ; Maska
        shr  ah, cl                 ; Przesunięcie maski na
                                    ; odpowiednią pozycję
        mov  al, 8                  ; Indeks rejestru maski
                                    ; karty EGA
        push dx                     ; Zachowanie wartości
                                    ; koloru
        mov  dx, 3CEh               ; Rejestr indeksowy
                                    ; karty EGA
        out  dx, ax                 ; Wysłanie maski
```

```
        mov  al, BYTE PTR es:[bx] ; Pobranie danej do
                                  ; rejestru zatrzasku
        pop  dx
        and  dl, 0Fh
        mov  BYTE PTR es:[bx], dl ; Wysłanie wartości
                                  ; koloru
        ret
EGAPixel ENDP
END
```

W pierwszej części programu przy wykorzystaniu przedstawionych wcześniej wzorów obliczany jest adres bajtu, zawierającego piksel. Podobnie jak w poprzednim przykładzie symbol **GMODE** używany jest w instrukcjach kompilacji warunkowej. Należy więc pamiętać o jego zdefiniowaniu.

Po obliczeniu adresu, wyznaczana jest maska. Operacja ta jest wykonywana przez podzielenie numeru kolumny przez 8 (liczba pikseli w bajcie) oraz przesunięcie bajtu maski (80h) w prawo o odpowiednią ilość pozycji. Jest ona następnie wysyłana do karty EGA przez porty indeksowy i danych.

Ostatnim krokiem jest pobranie danej, która znajduje się pod wyliczonym adresem, do rejestru zatrzaskowego. Następnie pod ten sam adres wpisywany jest kolor wyświetlanego piksela. Układ zatrzasku zmodyfikuje odpowiednie ramki i piksel pojawi się na ekranie.

PIKSELE NA KARCIE VGA

W karcie VGA istnieją trzy nowe tryby. Pierwsze dwa (17 i 18) są podobne do trybu 16 karty EGA. Procedura, która wyświetla piksel na karcie EGA, bez żadnych modyfikacji będzie działała również w trybie 18 karty VGA. Jedyną różnicą jest tylko zakres akceptowanych wartości numeru wiersza. Zostaje on zwiększony z 0-349 do 0-479.

W trybie 17 może zostać zastosowana większa część funkcji dla karty EGA. Należy tylko zastąpić procedurę maskowania i zapisywania bajtu taką samą procedurą dla karty CGA, pracującej w trybie 6.

Ostatni tryb (19) jest najprostszy do wykorzystania. Jego rozdzielczość wynosi 320x200 pikseli, a każdy piksel zapamiętywany jest na jednym bajcie. Oznacza to więc, że nie są potrzebne żadne operacje maskowania i przesuwania. Po prostu należy przesłać bajt do pamięci video. Wzór na obliczenie jego adresu jest następujący:

```
Przemieszczenie = 320 * y + x
```

A oto podprogram, który wyświetla pojedynczy piksel:

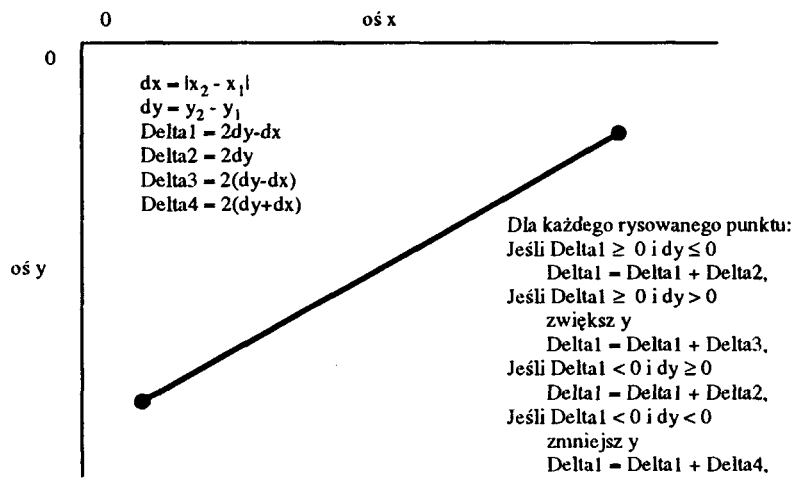
```
; Plik VGAPIX.ASM
; Wyświetlenie piksela w trybie 13h
        .MODEL  SMALL
        PUBLIC  VGAPixel
        .DATA
ScrBase DW   0A000h               ; Segment pamięci video
        .CODE
; AX = kolumna, BX = wiersz, DL = kolor
VGAPixel PROC
```

```
        mov   es, ScrBase          ; Segment pamięci video
        mov   cl, 6
        shl   bx, cl               ; bx = wiersz razy 64
        mov   cx, bx
        shl   bx, 1
        shl   bx, 1                ; bx = wiersz razy 256
        add   bx, cx               ; bx = wiersz razy 320
        add   bx, ax               ; Dodanie do łącznego
                                   ; wyniku
        mov   BYTE PTR es:[bx], dl ; Wysłanie wartości
                                   ; koloru
        ret
VGAPixel ENDP
END
```

W pierwszej części programu, przy wykorzystaniu przedstawionego wcześniej wzoru obliczany jest adres bajtu, zawierającego piksel. Adres umieszczany jest w rejestrach ES:BX. Ostatnia instrukcja zapisuje kolor piksela, który znajduje się w rejestrze DL do pamięci video, co powoduje jego wyświetlenie na ekranie.

Algorytm rysowania linii

Teraz, gdy już jest znany sposób wyświetlania pojedynczych pikseli, to można pokusić się o tworzenie bardziej ciekawych obiektów. Przykładem może być chociażby procedura z rozdziału 12, która rysuje okrąg. Poniższy przykład przedstawia algorytm rysowania linii. Jest on częściowo podobny do algorytmu tworzenia okręgu. Kreślenie linii odbywa się od jednego jej końca do drugiego, a kolejne jej punkty obliczane są przy zastosowaniu działań na liczbach całkowitych. Algorytm ten został wymyślony przez Breshenhama. Rysunek 15-5 przedstawia zasadę jego działania.



Rysunek 15-5. ***Algorytm Breshenhama rysowania linii.***

Algorytm ten nie wykorzystuje operacji dzielenia, a używane jest jedynie mnożenie przez liczby, które są potęgami dwójki. Dzięki temu jest on bardzo szybki. Jednakże za pomocą niego nie można narysować linii o nachyleniu większym od jedności (kąt nachylenia większy niż 45 stopni). Problem ten jest rozwiązany przez zamianę wartości współrzędnych x i y. Natomiast po wykonaniu obliczeń, przed narysowaniem kolejnego punktu, wartości te są ponownie odwracane. A oto cały program:

```
; Plik DRAWLINE.ASM
; Narysowanie linii przy użyciu algorytmu Breshenhama
          .MODEL  SMALL       ; Program będzie działał
                              ; w dowolnym modelu. Musi on
                              ; jednak być taki sam we
                              ; wszystkich modułach.
PUBLIC    DrawLine            ; Funkcja ta będzie widoczna
                              ; we wszystkich modułach
EXTRN     CGAPixel:PROC       ; Można użyć dowolnej
                              ; procedury, która wyświetla
                              ; piksel
          .DATA
Color     DB    ?             ; Kolor linii
X1        DW    ?             ; Początkowy punkt linii.
Y1        DW    ?
X2        DW    ?             ; Końcowy punkt linii
Y2        DW    ?
DiffX     DW    ?             ; Różnica pomiędzy początkowym
DiffY     DW    ?             ; a końcowym punktem
Delt1     DW    ?             ; Zmienne pomocnicze używane
Delt2     DW    ?             ; przy rysowaniu linii
Delt3     DW    ?
Delt4     DW    ?
Flipped   DB    0             ; Wartość prawdy, gdy
                              ; współrzędne x i y zostały
                              ; zamienione
; Argumenty funkcji:
; AX = Numer kolumny punktu początkowego
; BX = Numer wiersza punktu początkowego
; SI = Numer kolumny punktu końcowego
; DI = Numer wiersza punktu końcowego
; DL = Kolor linii
          .CODE
DrawLine PROC
          mov   Flipped, 0    ; Nie było zamiany
          mov   Color, dl     ; Zapamiętanie wartości koloru
          mov   X1, ax        ; Zapamiętanie współrzędnych
          mov   Y1, bx        ; punktu początkowego
          sub   ax, si        ; Odjęcie numeru kolumny
```

```
                                ; końcowej
        cwd                     ; Wyznaczenie wartości
                                ; bezwzględnej
        xor  ax, dx
        sub  ax, dx
        mov  DiffX, ax          ; Zapamiętanie tej różnicy
        mov  ax, bx
        sub  ax, di             ; Odjęcie numerów wierszy
        cwd                     ; Wyznaczenie wartości
                                ; bezwzględnej
        xor  ax, dx
        sub  ax, dx
        mov  DiffY, ax          ; Zapamiętanie tej różnicy
        cmp  ax, DiffX          ; Jeśli nachylenie < 1
        jle  NoFlip             ; nie jest dokonywana zamiana
                                ; współrzędnych
        mov  Flipped, 1         ; Ustawienie znacznika zamiany
        mov  DiffX, ax          ; Umieszczenie wartości DiffY
                                ; w DiffX
        mov  ax, X1             ; Zamiana X1 z Y1
        mov  bx, Y1
        mov  X1, bx
        mov  Y1, ax
        mov  ax, si             ; Zamiana SI z DI
        mov  si, di
        mov  di, ax
NoFlip:
        mov  X2, si             ; Zapamiętanie współrzędnych
        mov  Y2, di             ; punktu końcowego
        mov  ax, X1
        cmp  ax, X2             ; Czy X1 > X2
        jle  NoFlip2
        mov  ax, Y2             ; Zamiana współrzędnych y
        mov  bx, Y1
        mov  Y1, ax
        mov  Y2, bx
        mov  ax, X2             ; Zamiana współrzędnych x
        mov  bx, X1
        mov  X1, ax
        mov  X2, bx
NoFlip2:
        mov  ax, Y2             ; Ustalenie znaku DiffY
        sub  ax, Y1
        mov  DiffY, ax
        shl  ax, 1              ; Delt2 = DiffY * 2
        mov  Delt2, ax
```

```
        sub  ax, DiffX     ; Delt1 = DiffY * 2 - DiffX
        mov  Delt1, ax
        mov  ax, DiffY     ; Delt3 = (DiffY - DiffX) * 2
        sub  ax, DiffX
        shl  ax, 1
        mov  Delt3, ax
        mov  ax, DiffY     ; Delt4 = (DiffY + DiffX) * 2
        add  ax, DiffX
        shl  ax, 1
        mov  Delt4, ax
LineLp:
        cmp  Flipped, 0    ; Czy była zamiana
                           ; współrzędnych
        jne  NoFlip3
        mov  ax, X1        ; Współrzędne bieżącego punktu
        mov  bx, Y1
        jmp  SHORT DL1
NoFlip3:
        mov  ax, Y1
        mov  bx, X1
DL1:
        mov  dl, Color     ; Wyświetlenie punktu
        call CGAPixel      ; Wywołanie procedury, która
                           ; wyświetla piksel. Należy
                           ; tutaj użyć funkcji
                           ; odpowiedniej do zainstalo-
                           ; wanej karty graficznej.
        mov  ax, X1        ; Czy bieżący punkt jest
                           ; punktem końcowym
        cmp  ax, X2
        jge  DLDone
        inc  X1            ; Kolejny punkt
        cmp  Delt1, 0      ; Decyzja o zmianie wartości y
        jge  DL2
        cmp  DiffY, 0      ; Jeśli Delt1 < 0
        jge  DL3           ; i DiffY < 0
        dec  Y1            ; Zmniejszenie wartości y
                           ; o jeden
        mov  ax, Delt4     ; i dodanie do Delt1
                           ; wartości Delt4
        jmp  SHORT DL4
DL2:
        cmp  DiffY, 0      ; Jeśli Delt1 >= 0
        jle  DL3           ; i DiffY >= 0
        inc  Y1            ; Zwiększenie wartości y
                           ; o jeden
```

```
        mov  ax, Delt3      ; i dodanie do Delt1
                            ; wartości Delt3
        jmp  SHORT DL4
DL3:                        ; Jeśli nie jest zmieniana
                            ; wartość y dodanie do
        mov  ax, Delt2      ; Delt1 wartości Delt2
DL4:
        add  Delt1, ax
        jmp  SHORT LineLp
DLDone:
        ret
DrawLine ENDP
END
```

W pierwszym kroku procedura **DrawLine** ustawia wartości używanych zmiennych. **DiffX** i **DiffY** oznaczają liczbę pikseli w wierszach i kolumnach do punktu końcowego linii. Tak więc, jeśli wartość **DiffY** jest większa od **DiffX**, to nachylenie prostej jest większe od jedności; współrzędne muszą więc zostać zamienione. Zamiana ta jest oznaczana przez zapisanie jedynki do zmiennej **Flipped**.

Dwie kolejne instrukcje za etykietą **NoFlip** zapisują wartości kolumny i wiersza punktu końcowego w zmiennych **Y2** i **X2**. Następnie procedura sprawdza, czy punkt ten leży na prawo od punktu startowego. Jeśli nie, to ich współrzędne są zamieniane, tak więc w pozostałej części programu numery kolumn kolejnych punktów zwiększają się.

We fragmencie za etykietą **NoFlip2** obliczane są wartości zmiennych, które determinują momenty zmiany numeru wiersza. Są to zmienne **Delt1**, **Delt2**, **Delt3** i **Delt4**.

Etykieta **LineLp** rozpoczyna pętlę, rysującą kolejne punkty linii. W pierwszym kroku sprawdzane jest, czy ich współrzędne zostały zamienione. Jeśli tak, to przed narysowaniem punktu zostaje im przywrócone pierwotne znaczenie. Następnie wywoływany jest podprogram, który wyświetla pojedynczy piksel. W przykładzie została użyta procedura, która działa na karcie CGA. Można jednak użyć dowolnej innej funkcji, odpowiedniej do zainstalowanej karty. Po narysowaniu punktu procedura **DrawLine** porównuje numery kolumn punktu narysowanego i końcowego, ustalając moment zakończenia swojego działania.

Jeśli nie był to punkt końcowy, to numer kolumny zostaje zwiększony o jeden, a numer wiersza jest odpowiednio obliczany. W pierwszym teście sprawdzane jest, czy wartość **Delt1** jest większa od 0. Jeśli zarówno **Delt1** jak i **DiffY** są ujemne, wówczas numer ten jest zmniejszany o jeden, a do **Delt1** dodawana jest wartość **Delt3**. Jeśli obie zmienne są dodatnie, to numer wiersza jest zwiększany o jeden, a do **Delt1** dodawana jest wartość **Delt4**. Jeśli modyfikacja numeru wiersza nie była konieczna, wówczas do zmiennej **Delt1** dodawana jest wartość **Delt2**. Powyższe obliczenia ilustruje rysunek 15-5.

Po wykonaniu obliczeń, instrukcja **JMP**, która znajduje się przed etykietą **DLDone**, zamyka pętlę, a to powoduje, że rysowane są kolejne punkty linii.

Zabawa z liniami

Następny przykład wykorzystuje podprogram rysowania linii do tworzenia animacji. Wyświetla on kilka trójkątów, które poruszają się po ekranie w losowych kierunkach. W pro-

gramie istnieje kolejka, która może przechowywać położenia 32 ostatnio narysowanych figur. Jest ona zorganizowana jako bufor okrężny podobny do tego przedstawionego w rozdziale 14.
Z każdym rogiem trójkąta związany jest wektor, który określa następne położenie trójkąta. Po osiągnięciu krawędzi ekranu wektor ten przyjmuje pewną losową wartość.
Program generuje liczby losowe według reguły liczb względnie pierwszych. Zasada ta mówi, że ostatnia liczba losowa musi być pomnożona przez liczbę pierwszą (w tym wypadku przez 13), a rezultat podzielony przez liczbę pierwszą większą niż największa oczekiwana liczba losowa (257 spełnia ten warunek). Reszta z dzielenia stanowi kolejną wylosowaną liczbę.
Aby przy każdym uruchomieniu programu nie była generowana ta sama sekwencja liczb, procedura, która generuje liczby losowe wywoływana jest z inną wartością początkową. Wartość ta jest wartością licznika taktów zegara i jest pobierana z obszaru danych BIOS-u.

```
; Plik MOVELINE.ASM
         .MODEL  SMALL
EXTRN    DrawLine:PROC              ; Funkcja, która rysuje
                                    ; linię z poprzedniego
                                    ; przykładu
         .DATA
QSTRU    STRUC                      ; Struktura ta zawiera
                                    ; współrzędne trzech
X1       DW  ?                      ; wierzchołków trójkąta
Y1       DW  ?
X2       DW  ?
Y2       DW  ?
X3       DW  ?
Y3       DW  ?
QSTRU    ENDS
Head     DW  0                      ; Początek kolejki
Tail     DW  0                      ; Koniec kolejki
QCnt     DW  20                     ; Liczba trójkątów
                                    ; w kolejce
Cntr     DW  50                     ; Czas oczekiwania na
                                    ; zmianę długości kolejki
seed     DW  0                      ; Losowa wartość
                                    ; początkowa
Queue    QSTRU 32 DUP ( <> )        ; Tablica trójkątów
; Poniższe zmienne muszą być umieszczone obok siebie
DeltX1   DB  ?                      ; Zmiana w X dla punktu 1
DeltY1   DB  ?                      ; Zmiana w Y dla punktu 1
DeltX2   DB  ?                      ; Zmiana w X dla punktu 2
DeltY2   DB  ?                      ; Zmiana w Y dla punktu 2
DeltX3   DB  ?                      ; Zmiana w X dla punktu 3
DeltY3   DB  ?                      ; Zmiana w Y dla punktu 3
         .CODE
```

```
Main    PROC
        mov  ax, SEG DGROUP
        mov  ds, ax
        mov  ax, GMODE          ; Numer trybu graficznego
        int  10h
        mov  bx, OFFSET Queue   ; Utworzenie pierwszego
                                ; trójkąta
        mov  [bx].X1, 159       ; Umieszczenie go na
                                ; środku ekranu
        mov  [bx].Y1, 99
        mov  [bx].X2, 159
        mov  [bx].Y2, 99
        mov  [bx].X3, 159
        mov  [bx].Y3, 99
        mov  bx, OFFSET DeltX1
        mov  cx, 6
ILoop:                          ; Ustawienie początkowych
                                ; kierunków
        call Rand               ; Wylosowanie liczby
        and  ax, 7              ; z zakresu od 0 do 7
        sub  ax, 3              ; z zakresu od -3 do 4
        shl  ax, 1              ; z zakresu od -6 do 8
        mov  [bx], al           ; Zapisanie jej
        inc  bx
        loop ILoop
BgLoop:
        dec  Cntr               ; Zmniejszenie licznika
                                ; zmian
        jnz  BL1                ; Jeśli nie zero, nie ma
                                ; zmiany
        call Rand
        and  ax, 31             ; ax = od 0 do 31
        add  ax, 25             ; ax = od 25 do 56
        mov  Cntr, ax           ; Nowa wartość zmiennej
                                ; Cntr
        call Rand
        mov  cl, 25
        div  cl                 ; AH = od 0 do 24
        add  ah, 5              ; AH = od 5 do 29
        mov  al, ah
        cbw
        mov  QCnt, ax           ; Nowy rozmiar kolejki
BL1:
        mov  bx, Head           ; Pobranie numeru trójkąta
        mov  dl, 3              ; Ustawienie jego koloru
        call DrawTri
```

```
        push ds                 ; Przygotowanie do
                                ; kopiowania
        pop  es
        mov  si, Head           ; Pobranie bieżącego
                                ; trójkąta
        shl  si, 1              ; Razy 12 (rozmiar
                                ; struktury opisującej
        shl  si, 1              ; trójkąt)
        mov  ax, si
        shl  si, 1
        add  si, ax
        add  si, OFFSET Queue   ; Dodanie adresu tablicy
        mov  di, Head           ; Obliczenie numeru
                                ; następnego trójkąta
        inc  di
        and  di, 31             ; Zawinięcie na końcu
                                ; tablicy
        mov  Head, di           ; Zapisanie nowej wartości
        shl  di, 1
        shl  di, 1
        mov  ax, di
        shl  di, 1
        add  di, ax
        add  di, OFFSET Queue
        mov  cx, 12             ; Skopiowanie 12 bajtów
        cld
        rep  movsb
        sub  di, 12             ; Wskazanie na nowy trójkąt
        mov  bx, di
        call Adjust             ; Dodanie nowych wektorów
Hdlp:
        mov  ax, Head           ; Sprawdzenie, czy Head
                                ; jest prawidłowo
        sub  ax, Tail           ; oddalone od Tail
        jns  HdFirst
        add  ax, 32             ; Korekta wskaźnika po
                                ; zawinięciu
HdFirst:
        cmp  ax, QCnt           ; Jeśli różnica
                                ; jest > QCnt
        jb   NoErase
        mov  bx, Tail           ; Koniec kolejki
        mov  dl, 0              ; Czarny kolor
        call DrawTri            ; Skasowanie trójkąta
        mov  ax, Tail           ; Przesunięcie końca
                                ; kolejki
```

```
        inc  ax
        and  ax, 1Fh
        mov  Tail, ax
        jmp  SHORT Hdlp          ; Powtórzenie aż Tail
                                 ; będzie odległy od Head
                                 ; o QCnt pozycji
NoErase:
        mov  ah, 1               ; Test naciśnięcia
                                 ; klawisza
        int  16h
        jnz  TRDone              ; Zakończenie programu po
                                 ; naciśnięciu klawisza
        jmp  BgLoop
TRDone:
        mov  ah, 0               ; Odczytanie znaku
                                 ; z klawiatury
        int  16h
        mov  ax, 3               ; Przywrócenie trybu
                                 ; początkowego
        int  10h
        mov  ah, 4Ch             ; Funkcja kończąca program
        int  21h
Main    ENDP
; Narysowanie trójkąta
DrawTri PROC
        shl  bx, 1
        shl  bx, 1
        mov  ax, bx
        shl  bx, 1
        add  bx, ax
        mov  ax, [Queue+bx].X1 ; Pobranie współrzędnych
                               ; pierwszego punktu
        mov  cx, [Queue+bx].Y1
        mov  si, [Queue+bx].X2 ; Pobranie współrzędnych
                               ; drugiego punktu
        mov  di, [Queue+bx].Y2
        push dx
        push bx
        mov  bx, cx
        call DrawLine          ; Narysowanie pomiędzy
                               ; nimi prostej
        pop  bx
        pop  dx
        mov  ax, [Queue+bx].X2 ; Pobranie współrzędnych
                               ; drugiego punktu
        mov  cx, [Queue+bx].Y2
```

```
        mov  si, [Queue+bx].X3 ; Pobranie współrzędnych
                               ; trzeciego punktu
        mov  di, [Queue+bx].Y3
        push dx
        push bx
        mov  bx, cx
        call DrawLine          ; Narysowanie pomiędzy
                               ; nimi prostej
        pop  bx
        pop  dx
        mov  ax, [Queue+bx].X3 ; Pobranie współrzędnych
                               ; trzeciego punktu
        mov  cx, [Queue+bx].Y3
        mov  si, [Queue+bx].X1 ; Pobranie współrzędnych
                               ; pierwszego punktu
        mov  di, [Queue+bx].Y1
        mov  bx, cx
        call DrawLine          ; Narysowanie pomiędzy
                               ; nimi prostej
        ret
DrawTri ENDP
; Zwraca w ax losową liczbę
Rand    PROC
        push dx                ; Zachowanie rejestru DX
        push cx                ; i CX
        cmp  seed, 0           ; Czy potrzebna wartość
                               ; początkowa
        jne  SeedOK
        mov  ax, 40h           ; Segment danych BIOS-u
        mov  es, ax
        mov  ax, WORD PTR es:[6Ch]
        mov  seed, ax          ; Zapamiętanie nowej
                               ; wartości początkowej
SeedOK:
        mov  ax, seed          ; Pobranie wartości
                               ; początkowej
        mov  cl, 13            ; Mnożnik
        mul  cl
        sub  dx, dx            ; Wyzerowanie DX
        mov  cx, 257           ; Duża liczba pierwsza
        div  cx
        mov  seed, dx          ; Zapamiętanie reszty
        mov  ax, dx
        pop  cx                ; Odtworzenie rejestrów CX
        pop  dx                ; i DX
        ret
```

```
Rand      ENDP
; Modyfikacja współrzędnych wszystkich punktów trójkąta
; wskazywanego przez BX o wartości zawarte w zmiennej Delta
Adjust    PROC
          mov  si, OFFSET DeltX1 ; Wskazanie na wartość
                                 ; pierwszej poprawki
          mov  dx, 3             ; Liczba punktów
ADLoop:
          mov  cx, [bx]          ; Pobranie słowa wskazy-
                                 ; wanego przez BX
          mov  al, [si]          ; Pobranie poprawki
          cbw
          add  cx, ax            ; Nowy punkt
          jns  BigEnough1
          mov  WORD PTR [bx], 0  ; Ustawienie x na 0
          call FixDelt
          jmp  SHORT ADY
BigEnough1:
          cmp  cx, 319           ; Sprawdzenie prawej
                                 ; krawędzi ekranu
          jb   SmallEngh1
          mov  WORD PTR [bx], 319
          call FixDelt
          jmp  SHORT ADY
SmallEngh1:
          mov  [bx], cx
ADY:
          mov  cx, [bx+2]         ; Pobranie współrzędnej
          mov  al, [si+1]         ; Pobranie poprawki
          cbw
          add  cx, ax             ; Nowy punkt
          jns  BigEnough2
          mov  WORD PTR [bx+2], 0 ; Ustawienie y na 0
          call FixDelt
          jmp  SHORT ADNext
BigEnough2:
          cmp  cx, 199           ; Sprawdzenie dolnej
                                 ; krawędzi ekranu
          jb   SmallEngh2
          mov  WORD PTR [bx+2], 199
          call FixDelt
          jmp  SHORT ADNext
SmallEngh2:
          mov  [bx+2], cx
ADNext:
          add  bx, 4             ; Następny punkt
```

```
        add  si, 2              ; Następna poprawka
        dec  dx
        jnz  ADLoop
        ret
Adjust  ENDP
; Zmiana o losowe wartości poprawek wskazywanych przez
; rejestr SI
FixDelt PROC
        call Rand               ; Zmiana przyrostu wsp. x
        and  al, 7              ; AL = od 0 do 7
        sub  al, 4              ; AL = od -4 do 3
        js   FD1                ; Jeśli dodatnia
        inc  al                 ; AL = od -4 do 4
                                ; z wyjątkiem 0
FD1:
        shl  al, 1              ; AL = od -8 do 8
                                ; z wyjątkiem 0
        mov  [si], al           ; Zapamiętanie nowej
                                ; poprawki
        call Rand               ; Zmiana przyrostu wsp. y
        and  al, 7              ; AL = od 0 do 7
        sub  al, 4              ; AL = od -4 do 3
        js   FD2                ; Jeśli dodatnia
        inc  al                 ; AL = od -4 do 4
                                ; z wyjątkiem 0
FD2:
        shl  al, 1              ; AL = od -8 do 8
                                ; z wyjątkiem 0
        mov  [si+1], al         ; Zapamiętanie nowej
                                ; poprawki
        ret
FixDelt ENDP
        .STACK 1024
END     Main
```

Na początku programu ustawiany jest rejestr segmentu danych i odpowiedni tryb graficzny. Do jego wyboru został użyty symbol **GMODE**, który jest również wykorzystywany w procedurach, wyświetlających piksel na kartach CGA i EGA. Musi on zostać zdefiniowany przed podjęciem kompilacji programu. Następnie do kolejki wstawiany jest jeden trójkąt. Pętla **ILoop** ustawia wartości zmiennych, które zostaną wykorzystane do wyliczenia współrzędnych wierzchołków następnej figury. Odbywa się to przez wylosowanie liczby i zmianę jej wartości na daną z zakresu od -6 do 8.

Pętla **BgLoop** wykonuje wszystkie najważniejsze operacje programu. W pierwszym kroku sprawdza ona wartość zmiennej **Cntr** w celu ustalenia momentu zmiany rozmiaru kolejki. Jeśli akcja ta zostaje podjęta, to program generuje dwie losowe liczby, które stają się nowymi wartościami zmiennych **Cntr** i **QCnt**.

Jeśli długość kolejki nie musi być zmieniana, wówczas wykonywany jest skok do etykiety **BL1**. Tu następuje wywołanie procedury **DrawTri**, która wyświetla kolejny trójkąt. Po narysowaniu jego współrzędne zostają przekopiowane na następną pozycję w kolejce. Następnie są one modyfikowane według wartości zmiennej **Delta** przez procedurę **Adjust**. Po obliczeniu nowego trójkąta, główna pętla zamazuje figurę, która znajduje się na końcu kolejki. W ostatnim kroku badany jest stan klawiatury. Jeśli nie został wciśnięty klawisz, to proces powtarza się od początku.
Uruchomienie tego programu wymaga skompilowania kilku modułów: modułu, zawierającego ten właśnie program oraz modułów, zawierających procedury rysowania linii i punktu oraz ich skonsolidowania. Na przykład, stworzenie programu, który działa na karcie VGA, wymaga użycia komend:

```
tasm vgapix
tasm drawline
tasm /dGMODE=19 moveline
tlink moveline drawline vgapix
```

Podsumowanie

Chcąc wykorzystać grafikę, pierwszą rzeczą jaką należy zrobić, jest ustalenie typu zainstalowanej karty i wybranie odpowiedniego trybu jej pracy. Informacja ta pozwala ustalić położenie pamięci video i sposób jej zorganizowania. Następnie, korzystając z funkcji BIOS-u, należy włączyć wybrany tryb i można już rozpocząć wyświetlanie tekstu lub grafiki.
W trybach tekstowych każdy parzysty bajt pamięci video zawiera kod ASCII wyświetlanego znaku. Bajt występujący za nim określa atrybuty lub kolor tego znaku. Na większości kart zapis do pamięci może odbywać się w dowolnym momencie. Jednakże na niektórych kartach CGA operacja taka może spowodować powstanie efektu śnieżenia. Można go wyeliminować, dokonując dostępu do pamięci tylko podczas trwania impulsów odchylania poziomego lub pionowego.
Tryby graficzne są bardziej skomplikowane, chociażby ze względu na ich mnogość. W każdym z nich inaczej rozwiązane jest ułożenie pikseli w bajcie pamięci, ich kolory i organizacja pamięci. Oczywiście po napisaniu procedur, które wyświetlają piksele w każdym z tych trybów, można już użyć standardowych algorytmów rysowania linii, okręgów czy innych figur.
Przykładowe funkcje zamieszczone w tym rozdziale mogą współpracować zarówno ze sobą, jak i z wieloma innymi programami. Program **moveline** jest dobrym przykładem na ich wykorzystanie. Ponadto należy zwrócić w nim uwagę na:

- Umieszczenie na początku programu dyrektywy **EXTRN** dla każdej funkcji wykorzystywanej w nim.
- Zgodność modeli pamięci we wszystkich modułach, wchodzących w skład programu.
- Rejestr DS musi wskazywać na grupę **DGROUP**, a pozostałe rejestry przed wywołaniem każdej z funkcji muszą zawierać odpowiednie dla niej parametry.
- Nie można liczyć na to, że po wywołaniu funkcji wartości rejestrów pozostaną nie zmienione. Jeśli jest to konieczne przed wywołaniem procedury, to należy je przesłać na stos, a po jej wykonaniu pobrać z niego.

V Dodatki

- » Opcje programów TASM i TLink
- » Komendy Turbo Debuggera
- » Przerwania i funkcje BIOS-u i DOS-u
- » Konwersje znaków ASCII

Dodatek A Opcje programów TASM i TLink

Tabela A-1. Opcje Turbo Assemblera

Opcja	Opis
/A	Polecenie alfabetycznego uporządkowania segmentów przed zapisem ich do pliku wynikowego.
/B	Opcja jest rozpoznawana, ale nie wywołuje żadnej akcji.
/C	Umieszczenie w raporcie kompilacji podraportu odwołań skrośnych.
/D<sym>[=VAL]	Zdefiniowanie symbolu jako 0 lub VAL.
/E	Polecenie generowania rozkazów, które emulują operacje zmiennoprzecinkowe.
/H,/?	Polecenie wyświetlenia wykazu opcji.
/I<katalog>	Określenie nazwy katalogu zbiorów nagłówkowych.
/J<dyrektywa>	Określenie dyrektywy, która będzie wykonana przed podjęciem kompilacji.
/KH<numer>	Kompilacja programu, który zawiera maksymalnie podaną liczbę symboli. Wartość domniemana to 8182, a maksymalna 32768.
/KS<numer>	Kompilacja programu, który wymaga podanego obszaru pamięci dla łańcuchów. Asembler potrafi najczęściej samodzielnie określić rozmiar obszaru pamięci przeznaczonego na przechowanie łańcuchów. Opcja ta przydaje się wtedy, gdy kompilator zawiedzie.
/L	Polecenie generowania raportu kompilacji.
/LA	Polecenie generowania rozszerzonego raportu kompilacji.
/M<numer>	Określenie ilości przebiegów asemblera, niezbędnych do skompilowania wszystkich odwołań w przód. (tylko TASM 2.0)
/ML	Polecenie rozróżniania małych i dużych liter.
/MU	Polecenie przekształcenia wszystkich małych liter w duże.
/MV<numer>	Ustawienie maksymalnej długości symboli (tylko TASM 2.0).
/MX	Polecenie rozróżniania małych i dużych liter, ale tylko w obrębie symboli zewnętrznych.
/N	Polecenie wyeliminowania z raportu kompilacji informacji o tablicy symboli
/P	Polecenie sprawdzenia, czy w module źródłowym użyto rozkazów, które zawierają przedrostek segmentu CS. Ponieważ tylko niektóre segmenty wykonywalne mogą zawierać dane, to wystąpienie takiego przedrostka może spowodować powstanie błędu.

Opcja	Opis
/Q	Polecenie usuwania zbędnych rekordów z modułów wynikowych (tylko TASM 2.0).
/R	Polecenie generowania rzeczywistych rozkazów arytmetyki zmiennoprzecinkowej.
/S	Polecenie umieszczania segmentów w pliku wynikowym w kolejności ich wystąpienia w pliku źródłowym.
/T	Polecenie zaniechania komunikatu o poprawnej kompilacji
/W-[klasa]	Polecenie zaniechania generacji ostrzeżeń z podanej klasy.
/W+[klasa]	Polecenie generacji ostrzeżeń z podanej klasy.
ASS	Założenie, że segment jest 16-bitowy.
BRK	Potrzebne nawiasy.
ICG	Nieefektywny kod.
LCO	Przepełnienie wskaźnika pozycji.
OPI	Otwarta instrukcja IF.
OPP	Otwarta procedura.
OPS	Otwarty segment.
OVF	Nadmiar arytmetyczny.
PDC	Konstrukcja zależna od numeru przebiegu.
PRO	Użycie w trybie chronionym przedrostka CS.
/X	Polecenie włączania do raportu kompilacji nie kompilowanych gałęzi dyrektyw warunkowych.
/Z	Wyświetlanie linii kodu źródłowego, które spowodowały powstanie błędu.
/ZI	Umieszczenie w zbiorze wynikowym kompletnej informacji o numerach wierszy i symbolach modułu źródłowego.
/ZD	Umieszczenie w zbiorze wynikowym tylko informacji o numerach wierszy modułu źródłowego.

[...] element opcjonalny, <...> argument wymagany

Tabela A-2. Opcje konsolidatora.

Opcja	Opis
/3	Zezwolenie na segmenty 32-bitowy.
/C	Polecenie rozróżniania małych i dużych liter w symbolach zewnętrznych.
/D	Polecenie informowania o wystąpieniu tego samego symbolu w kilku bibliotekach.
/E	Polecenie zignorowania biblioteki rozszerzonej.
/I	Polecenie inicjacji segmentów.
/L	Polecenie włączenia do zbioru, który zawiera program wykonywalny, informacji o numerach linii programu źródłowego.
/M	Polecenie uzupełnienia mapy konsolidacji wykazem symboli publicznych.
/N	Polecenie zignorowania bibliotek domniemanych.
/O	Polecenie utworzenia pliku nakładkowego.
/S	Polecenie uzupełnienia mapy konsolidacji wykazem symboli publicznych oraz mapą segmentów.
/T	Utworzenie pliku typu COM.
/U	Polecenie włączenia do zbioru, który zawiera program wykonywalny, informacji uruchomieniowych.
/X	Polecenie niegenerowania mapy konsolidacji.
/YE	Polecenie użycia pamięci typu expanded.
/YX	Polecenie użycia pamięci typu extended.

Dodatek B Komendy Turbo Debuggera

Tabela B-1. Klawisze funkcyjne.

Klawisz	Funkcja
ALT-=	Rozpoczęcie rejestrowania makrodefinicji.
ALT-	Zakończenie rejestrowania makrodefinicji.
ALT-1-9	Aktywowanie okienka o podanym numerze.
ALT-B	Wyświetlenie menu globalnego Breakpoints.
ALT-D	Wyświetlenie menu globalnego Data.
ALT-F	Wyświetlenie menu globalnego File.
ALT-F1	Wyświetlenie poprzedniego wyjaśnienia.
ALT-F2	Zastawienie pułapki adresowej.
ALT-F3	Zamknięcie bieżącego okienka.
ALT-F4	Cofnięcie działania ostatniej śledzonej instrukcji.
ALT-F5	Wyświetlenie ekranu wyjściowego.
ALT-F6	Wyświetlenie ostatnio zamkniętego okienka.
ALT-F7	Wykonanie jednej instrukcji.
ALT-F8	Wykonanie aż do powrotu z procedury.
ALT-F9	Wykonanie do podanego adresu.
ALT-F10	Wyświetlenie lokalnego menu.
ALT-H	Wyświetlenie menu globalnego Help.
ALT-O	Wyświetlenie menu globalnego Options.
ALT-R	Wyświetlenie menu globalnego Run.
ALT-V	Wyświetlenie menu globalnego View.
ALT-W	Wyświetlenie menu globalnego Window.
ALT-X	Powrót do systemu DOS.
CTRL->	Zwiększenie adresu startowego o 1.
CTRL-<	Zmniejszenie adresu startowego o 1.
CTRL-A	Przesunięcie kursora do poprzedniego słowa.
CTRL-C	Przewinięcie o jeden ekran w dół.
CTRL-D	Przesunięcie kursora o jedną kolumnę w prawo.
CTRL-E	Przesunięcie kursora o linię w górę.
CTRL-F	Przesunięcie kursora do następnego słowa.
CTRL-F2	Ponowne załadowanie programu.
CTRL-F4	Wyznaczenie wartości wyrażenia.
CTRL-F5	Zmiana położenia lub rozmiaru okienka.

Tabela B-1. Klawisze funkcyjne. (*kontynuacja*)

Klawisz	Funkcja
CTRL-F7	Umieszczenie zmiennej w okienku obserwacyjnym.
CTRL-F8	Zastawienie/usunięcie pułapki.
CTRL-F9	Uruchomienie programu.
CTRL-F10	Wyświetlenie lokalnego menu.
CTRL-R	Przewinięcie o jeden ekran w górę.
CTRL-S	Przesunięcie kursora o jedną kolumnę w lewo.
CTRL-X	Przesunięcie kursora o linię w dół.
ESC	Zamknięcie okienka inspekcyjnego albo menu.
F1	Wyświetlenie wyjaśnienia.
F2	Zastawienie pułapki.
F3	Wybranie modułu źródłowego.
F4	Wykonanie do kursora.
F5	Powiększenie okienka do rozmiaru ekranu.
F6	Wyświetlenie następnego okienka.
F7	Wykonanie jednej instrukcji/rozkazu.
F8	Wykonanie j.w., ale z omijaniem procedur.
F9	Uruchomienie programu.
F10	Aktywowanie głównego menu.
INS	Zainicjowanie zaznaczania bloku tekstu.
SHIFT-F1	Wyświetlenie spisu wyjaśnień.
SHIFT-TAB	Przemieszczenie kursora do poprzedniego okienka.

Tabela B-2. Menu systemowe.

Komenda	Działanie
Restore Standard	Przywrócenie standardowego rozmieszczenia okienek.
Repaint Desktop	Przerysowanie całego ekranu uruchamiacza.
About	Wyświetlenie informacji o uruchamiaczu.

Tabela B-3. Menu File.

Komenda	Działanie
Open	Załadowanie uruchamianego programu.
Change Dir	Zmiana bieżącego katalogu.
Get Info	Wyświetlenie informacji o programie.
DOS Shell	Tymczasowe wywołanie systemu DOS.
Resident	Uczynienie Turbo Debuggera programem rezydentnym.
Symbol Load	Załadowanie tablicy symboli programu.
Table Relocate	Zmiana adresu bazowego tablicy symboli.
Quit	Powrót do systemu MS-DOS.

Tabela B-4. Menu View.

Komenda	Działanie
Breakpoints	Wyświetlenie okienka pułapek.
Stack	Wyświetlenie okienka wywołań.
Log	Wyświetlenie okienka komunikatów.
Watches	Wyświetlenie okienka obserwacyjnego.
Variables	Wyświetlenie okienka zmiennych.
Module	Wyświetlenie okienka modułów.
File	Wyświetlenie okienka zbiorów.
CPU	Wyświetlenie okienka procesora.
Dump	Wyświetlenie okienka pamięci.
Registers	Wyświetlenie okienka rejestrów.
Numeric Processor	Wyświetlenie okienka koprocesora.
Execution History	Wyświetlenie okienka wykonanych instrukcji.
Hierarchy	Wyświetlenie okienka informacji o obiektach i klasach programu.
Another	Wyświetlenie okienka dodatkowego:
Module	Modułu
Dump	Sterty
File	Pliku

Tabela B-5. Menu Run.

Komenda	Działanie
Run	Uruchomienie programu.
Go to Cursor	Wykonanie programu do miejsca ustawienia kursora.
Trace Into	Wykonanie jednego wiersza źródłowego albo jednego rozkazu.
Step Over	Wykonanie jednego wiersza źródłowego albo jednego rozkazu bez wchodzenia do wywoływanych procedur.
Execute to	Wykonanie programu do podanego adresu.
Until Return	Wykonanie programu do instrukcji powrotu z procedury.
Animate	Wykonywanie programu bez udziału użytkownika.
Back Trace	Cofnięcie wykonania ostatniej instrukcji.
Instruction Trace	Wykonanie jednej instrukcji.
Arguments	Podanie argumentów wywołania programu.
Program Reset	Ponowne załadowanie programu.

Tabela B-6. Menu Breakpoints

Komenda	Działanie
Toggle	Zastawienie/usunięcie pułapki w miejscu ustawienia kursora.
At	Ustawienie pułapki pod podanym adresem.
Changed Memory Global	Zastawienie pułapki, w którą program wpadnie, gdy nastąpi zmiana zawartości podanego obszaru pamięci.
Expression True Global	Zastawienie pułapki, w którą program wpadnie, gdy podane wyrażenie stanie się prawdziwe.
Delete All	Usunięcie wszystkich pułapek.

Tabela B-7. Menu Data

Komenda	Działanie
Inspect	Wyświetlenie okienka inspekcyjnego, dotyczącego podanej zmiennej.
Evaluate/Modify	Wyznaczenie wartości wyrażenia.
Add Watch	Umieszczenie zmiennej w okienku obserwacyjnym.
Function Return	Ujawnienie rezultatu funkcji.

Tabela B-8. Menu Options

Komenda	Działanie
Language	Wybranie języka, w którym będą zapisywane wyrażenia.
Macros	Wyświetlenie podmenu, dotyczącego makrodefinicji:
Create	Rozpoczęcie rejestrowania makrodefinicji.
Stop Recording	Zakończenie rejestrowania makrodefinicji.
Remove	Usunięcie makrodefinicji.
Delete All	Usunięcie wszystkich makrodefinicji.
Display Options	Określenie sposobu wyświetlania danych.
Path for Source	Podanie katalogu, w którym są poszukiwane moduły źródłowe.
Save Options	Zapamiętanie parametrów uruchamiacza.
Restore Options	Odtworzenie z pliku parametrów uruchamiacza.

Tabela B-9. Menu Window

Komenda	Działanie
Zoom	Powiększenie okienka do rozmiaru ekranu.
Next	Uaktywnienie następnego okienka.
Next Pane	Wyróżnienie następnej ramki aktywnego okienka.
Size/Move	Zmiana położenia lub rozmiarów okienka.
Iconize/Restore	Znaczne zmniejszenie okienka lub przywrócenie mu normalnych wymiarów.
Close	Zamknięcie aktywnego okienka.
Undo Close	Otworzenie ostatnio zamkniętego okienka.
Dump Pane to Log	Skopiowanie aktywnej ramki do okienka komunikatów.
User Screen	Wyświetlenie ekranu wyjściowego.
Open window list	Wyświetlenie listy okienek, które mogą stać się aktywne.
Window Pick	Wyświetlenie listy wszystkich otwartych okienek.

Tabela B-10. Menu Help

Komenda	Działanie
Indeks	Wyświetlenie spisu wyjaśnień.
Previous Topic	Wyświetlenie poprzedniego wyjaśnienia.
Help on Help	Opis używania systemu pomocy.

Dodatek C *Przerwania i funkcje BIOS-u i DOS-u*

W rozdziale tym zaprezentowany jest sposób działania przerwań w systemie MS-DOS. Termin przerwanie oznacza procedurę zawieszenia wykonywania bieżącego programu, zapamiętania jego stanu, przełączenia procesora na obsłużenie innego procesu i, na koniec, przywrócenia poprzedniego stanu, dzięki czemu przerwany program może być kontynuowany od miejsca zawieszenia.

Instrukcje są wykonywane przez procesor w kolejności ich występowania w pamięci w kierunku większych adresów. Zarządzanie tym procesem nadzorują zawartości pary rejestrów CS:IP. Rejestr CS jest adresem bazowym segmentu pamięci; IP zawiera przemieszczenie w tym segmencie. Po wykonaniu każdej instrukcji procesor zwiększa rejestr IP o jeden lub kilka bajtów zależnie od rozmiaru wykonanego rozkazu. Tak więc, para CS:IP zawsze wskazuje na następną instrukcję do wykonania.

Instrukcja, która zakłóca ten stan rzeczy, to INT, wykonywana z jednobajtowym argumentem. Zmusza ona procesor do zapisania na stosie stanu znaczników oraz wartości rejestrów CS i IP. Podana liczba wskazuje na element struktury, która znajduje się w pamięci podstawowej zwanej tablicą wektorów przerwań (Interrupt Vector Table - IVT). Każdy jej element składa się z 4 bajtów, które razem stanowią adres, który będzie przesłany do rejestrów CS:IP procesora.Pod tym adresem znajduje się pierwsza instrukcja procedury, która ma wykonać określone zadanie. Procedura ta zwana jest procedurą obsługi przerwania (Interrupt Service Routine - ISR), a jej ostatnim rozkazem powinien być rozkaz IRET. Powoduje on pobranie danych ze stosu i umieszczenie ich w rejestrach IP, CS i znaczników, przywracając pierwotny stan procesora.

Tablica wektorów przerwań zawiera do 256 adresów procedur ich obsługi. Mogą one zostać wywołane instrukcją INT, której argument jest numerem żądanej procedury. Dobrze znany przykład to rozkaz INT 21h. Procesor zapisuje wtedy bieżące wartości znaczników oraz rejestrów CS i IP, pobiera z odpowiedniego elementu (21h) tablicy IVT adres i wykonuje pierwszą wskazywaną przez niego instrukcję. Po napotkaniu rozkazu IRET ze stosu pobierane są trzy słowa i umieszczane w rejestrach IP, CS oraz znaczników.

W procesorach rodziny 8088 przerwania podzielone są na kilka klas. Jedna z nich to przerwania sprzętowe. Przerwania sprzętowe to takie, które powodują pojawienie się napięcia na jednej z dwóch końcówek procesora, NMI (Non Maskable Interrupt - Przerwanie niemaskowalne) lub INTR (Interrupt). Tak więc istnieją dwa rodzaje przerwań sprzętowych.

Sygnał na końcówce NMI pojawia się tylko w wyniku poważnej awarii sprzętu, np. wykrycia uszkodzenia pamięci. Nie ma możliwości przewidzenia chwili wystąpienia takiej sytuacji, tak więc procedura obsługi tego przerwania podejmuje najlepszą akcję, którą jest przerwanie działalności całości systemu. Dzięki temu można uniknąć znacznych szkód (na przykład zapisania danych z uszkodzonego obszaru pamięci na dysk).

Sygnał na końcówce INTR pochodzi od układu kontroli przerwań 8259 (Peripheral Interface Controller). Jeśli znacznik zezwolenia na przerwanie (IF) jest ustawiony, to procesor je

przyjmuje. W przeciwnym razie jest ono ignorowane, a procesor kontynuuje wykonywanie bieżącego programu.
Inną klasę przerwań stanowią przerwania programowe. Istnieje ich kilka rodzajów. Jeśli procesor zmuszony jest wykonać dzielenie przez zero, to wysyła przerwanie. Jeśli znacznik pracy krokowej (TF) jest ustawiony, wówczas procesor generuje przerwanie po wykonaniu każdej instrukcji. Jeśli znacznik nadmiaru (OF) jest ustawiony, to procesor również generuje przerwanie.
Instrukcja INT może być użyta do wywołania dowolnego przerwania. Dzięki niej programista posiada pełną kontrolę nad komputerem, ale niektóre przerwania nie powinny być zmieniane. Poniżej opisane są różne przerwania oraz ich własności.

Przerwania i funkcje BIOS-u

Pierwsze szesnaście przerwań , INT 0 do INT 0Fh ma specjalne znaczenie. Firma Intel rezerwuje je dla procesora, co powinno zapewnić im nienaruszalność. Jednakże IBM przedefiniował je do użytku w komputerach IBM PC (i stąd w systemie MS-DOS). W poniższym zestawieniu obie definicje tych przerwań są rozróżniane, lecz przyporządkowanie przez IBM traktowane jest jako to, którego należy się spodziewać.
Następne szesnaście przerwań, INT 10h do INT 1Fh, obsługiwanych jest przez BIOS, a przerwania DOS-u rozpoczynają się od numeru 20h (często jednak mówiąc o przerwaniach BIOS-u ma się na myśli numery od 0 do 1Fh).
W rzeczywistości należy przede wszystkim używać funkcji dostępnych przez przerwanie INT 21h; ich działanie jest dobrze zdefiniowane. Jeśli jednak istnieje konieczność skorzystania z usług BIOS-u, to należy pamiętać, że ich zachowanie zmienia się w zależności od producenta. Jednakże zaletą funkcji BIOS-u jest o wiele większa efektywność, niż odpowiadających im funkcji przerwania INT 21h. Należy pamiętać, że bez dokładnej wiedzy o działaniu przerwań o numerach od 0 do 0Fh, ich użycie nie jest wskazane.

Przerwanie 00h Dzielenie przez zero

Przerwanie to jest generowane przez instrukcję dzielenia przez zero.

Przerwanie 01h Praca krokowa

Przerwanie to jest generowane po wykonaniu każdej instrukcji, o ile ustawiony jest znacznik pracy krokowej (Trap Flag).

Przerwanie 02h Przerwanie niemaskowalne

Przerwanie to sygnalizowane jest pojawieniem się sygnału na końcówce NMI procesora i oznacza wystąpienie błędu fatalnego.

Przerwanie 03h Punkt kontrolny

Przerwanie to jest wykorzystywane przede wszystkim przez programy uruchomieniowe. W kodzie programu umieszczana jest instrukcja INT 03h, a procedura obsługi może wyświetlać stan procesora czy zawartość różnych obszarów pamięci.

Przerwanie 04h Nadmiar

Przerwanie to generowane jest przez instrukcję INTO. Jeśli ustawiony jest znacznik nadmiaru (Overflow Flag) przed wykonaniem rozkazu INTO, to procesor traktuje go wtedy jako instrukcję INT 04h.

Przerwanie 05h Wydruk zawartości ekranu

Przerwanie to przez firmę Intel zostało przeznaczone do obsługi wyjątku przekroczenia zakresów tablicy. Natomiast w systemie MS-DOS generowane jest ono po wciśnięciu klawiszy Shift-PrtSc i powoduje wydrukowanie znaków ASCII, znajdujących się na ekranie.

Przerwanie 06h Zarezerwowane

Przerwanie to zdefiniowane jest przez firmę Intel do obsługi zgłoszenia nielegalnego rozkazu.

Przerwanie 07h Zarezerwowane

Przerwanie to zdefiniowane jest przez firmę Intel do obsługi zgłoszenia niedostępności koprocesora.

Przerwanie 08h Zegar Czasu Rzeczywistego

Przerwanie to jest generowane przez końcówkę IRQ0 układu 8259. Ma ono najwyższy priorytet wśród wszystkich możliwych ośmiu przerwań. Procedura jego obsługi wywołuje przerwanie INT 1Ch, które standardowo inicjowane jest jako puste (pojedyncza instrukcja IRET), ale w wielu programach badających zależności czasowe, jest często przechwytywane.

Przerwanie 09h Klawiatura

Przerwanie to zdefiniowane jest przez firmę Intel jako zarezerwowane. W systemie MS-DOS generowane jest ono po każdym wciśnięciu i zwolnieniu klawisza.

Przerwanie 0Ah (Różnie)

Przerwanie to zdefiniowane jest przez firmę Intel do obsługi zgłoszenia sygnału DMA0. W komputerach klasy AT używane jest do obsługi przerwania, które pochodzi z dodatkowego układu 8259 do obsługi sprzętowych żądań IRQ8 do IRQ15.

Przerwanie 0Bh (Różnie)

Przerwanie to zdefiniowane jest przez firmę Intel do obsługi zgłoszenia sygnału DMA1. W komputerach klasy AT dostarcza funkcji obsługujących porty COM2 i COM4. Natomiast w komputerach klasy PC dostarcza funkcji, które obsługują porty COM1 i COM3. W komputerach klasy PS/2 jest ono zarezerwowane (jego zastosowanie nie jest udokumentowane).

Przerwanie 0Ch (Różnie)

Przerwanie to zdefiniowane jest przez firmę Intel do obsługi instrukcji INTO. W komputerach klasy AT dostarcza funkcji, które obsługują porty COM1 i COM3. Natomiast w kom-

puterach klasy PC dostarcza funkcji, które obsługują porty COM2 i COM4. W komputerach klasy PS/2 jest ono zarezerwowane.

Przerwanie 0Dh (Różnie)

Przerwanie to zdefiniowane jest przez firmę Intel jako INT1. W komputerach klasy AT dostarcza funkcji, które obsługują port LPT2. Natomiast w komputerach klasy PC, wyposażonych w dysk twardy (PC XT), dostarcza funkcji, które go obsługują. Podobne usługi oferowane są przez to przerwanie w komputerze PS/2 Model 30. Na pozostałych maszynach klasy PS/2 jest ono zarezerwowane.

Przerwanie 0Eh Sterownik dysków

Przerwanie to zdefiniowane jest przez firmę Intel jako INT2. Na komputerach pracujących pod kontrolą systemu operacyjnego MS-DOS, obsługuje ono sterownik dysków elastycznych. ze względu na poprawki dokonane w pozostałych funkcjach BIOS-u, używanie go nie jest wskazane.

Przerwanie 0Fh Układ wyjściowy LPT1

Przerwanie to zdefiniowane jest przez firmę Intel jako INT3. Na komputerach AT i PC obsługuje ono porty drukarki, aczkolwiek używanie go nie jest wskazane. W komputerach klasy PS/2 jest ono zarezerwowane.

Przerwanie 10h Obsługa Ekranu

Przerwanie 10h udostępnia bogaty zbiór starannie opracowanych funkcji. Każda z nich może być wywołana przez umieszczenie odpowiedniej wartości w rejestrze AH. W wypadku, gdy dana funkcja składa się z kilku podfunkcji, ich numer podaje się w rejestrze AL. Ich parametry przekazywane są w pozostałych rejestrach. Mogą to być, na przykład kod wyświetlanego znaku, adres łańcucha czy atrybuty wyświetlania.

AH = 00h Wybór trybu pracy monitora

Funkcja ta powoduje ustawienie trybu pracy wybranej karty graficznej i wyczyszczenie ekranu. Żądany tryb przekazywany jest w rejestrze AL. Wartości od AL = 80h do AL = FFh mają takie samo znaczenie, jak wartości AL = 00 do AL = 7Fh, ale nie powodują wyczyszczenia ekranu.

Wejście: AH = 00h

Podanie poniższych wartości powoduje zapis do pamięci video od adresu 0B800h i uaktywnienie jednej z kart CGA, EGA, MCGA lub VGA:

AL = 00 lub 80h, tekst, 40 x 25, monochromatyczny
AL = 01 lub 81h, tekst, 40 x 25, 16 kolorów
AL = 02 lub 82h, tekst, 80 x 25, monochromatyczny
AL = 03 lub 83h, tekst, 80 x 25, 16 kolorów
AL = 04 lub 84h, grafika, 320 x 200, 4 kolory
AL = 05 lub 85h, grafika, 320 x 200, monochromatyczny
AL = 06 lub 86h, grafika, 640 x 200, monochromatyczny

Podanie poniższej wartości powoduje zapis do pamięci video od adresu 0B000h i uaktywnienie jednej z kart EGA, MDA lub VGA:

AL = 07 lub 87h, tekst, 80 x 25, monochromatyczny

Podanie poniższych wartości powoduje zapis do pamięci video od adresu 0B800h i uaktywnienie karty PCjr lub Tandy 1000:

AL = 08 lub 88h, grafika, 160 x 200, 16 kolorów
AL = 09 lub 89h, grafika, 320 x 200, 16 kolorów
AL = 0Ah lub 8Ah, grafika, 640 x 200, 4 kolory

Poniższe wartości zarezerwowane są do wewnętrznego użycia przez BIOS karty EGA:

AL = 0Bh lub 8Bh, zarezerwowana
AL = 0Ch lub 8Ch, zarezerwowana

Podanie poniższych wartości powoduje zapis do pamięci video od adresu 0A000h i uaktywnienie karty EGA lub VGA:

AL = 0Dh lub 8Dh, grafika, 320 x 200, 16 kolorów
AL = 0Eh lub 8Eh, grafika, 640 x 200, 16 kolorów
AL = 0Fh lub 8Fh, grafika, 640 x 350, monochromatyczny
AL = 10h lub 90h, grafika, 640 x 350, 16 kolorów

Podanie poniższej wartości powoduje zapis do pamięci video od adresu 0A000h i uaktywnienie karty MCGA lub VGA:

AL = 11h lub 91h, grafika, 640 x 480, monochromatyczny

Podanie poniższej wartości powoduje zapis do pamięci video od adresu 0A000h i uaktywnienie tylko karty VGA:

AL = 12h lub 92h, grafika, 640 x 480, 16 kolorów

Podanie poniższej wartości powoduje zapis do pamięci video od adresu 0A000h i uaktywnienie karty MCGA lub VGA:

AL = 13h lub 93h, grafika, 320 x 200, 256 kolorów

Wartości rejestru AL od 14h do 7Fh nie wszędzie są jednakowo wykorzystywane. Są one używane przez producentów do ich własnych potrzeb. Wybranie jednej z tych wartości powinno więc zostać poprzedzone lekturą odpowiedniej dokumentacji.

Wyjście: bez znaczenia

AH = 01h Definiowanie rozmiaru kursora

Funkcja ta używana jest w trybie tekstowym do określenia kształtu i szybkości migotania kursora.

Wejście: AH = 01h

Rejestry CH i CL sterują kształtem i szybkością migotania kursora. Bity od 0 do 4 rejestru CH określają numer linii, w której zaczyna się kursor, a bity 5 i 6 ustalają szybkość migotania (00 normalna, 01 kursor niewidoczny, 10 kursor wolny, 11 kursor szybki). Bity

od 0 do 4 rejestru CL określają numer linii, w której kończy się kursor. Jego kształt zależy więc od wartość bitów w rejestrach CH i CL.

Wyjście: bez znaczenia

AH = 02h Ustawienie pozycji kursora

Funkcja ta używana jest w trybie tekstowym do określenia pozycji kursora na ekranie.

Wejście: AH = 02h

Rejestr BH zawiera numer strony od 0 do 7, zależnie od trybu
Rejestry DH i DL zawierają numery wiersza i kolumny pozycji kursora

Wyjście: bez znaczenia

AH = 03h Odczytanie pozycji kursora

Funkcja ta używana jest do odczytania pozycji kursora na ekranie, jego kształtu i szybkości migotania.

Wejście: AH = 03h
Wyjście: Rejestr BH zawiera numer strony od 0 do 7
Rejestry DH i DL zawierają numery wiersza i kolumny pozycji kursora
Rejestr CH określa szybkość migotania i numer początkowej linii kursora
Rejestr CL określa numer końcowej linii kursora

AH = 04h Odczytanie pozycji pióra świetlnego

Funkcja ta używana jest do odczytania bieżącej pozycji pióra świetlnego. Nie działa ona na komputerach PS/2.

Wejście: AH = 04h
Wyjście: Rejestry DH i DL zawierają bieżące numery wiersza i kolumny
Rejestr CH lub CX, zależnie od trybu, zawiera numer wiersza ekranu
Rejestr BX zawiera numer kolumny ekranu
Rejestr AH zawiera wartość 1 lub 0, która oznacza aktywność pióra

AH = 05h Wybranie strony

Funkcja ta używana jest do wybrania numeru wyświetlanej strony.

Wejście: AH = 05h
AL = 0 do 7 dla trybów 00h, 01h, 07h, 0Dh: EGA, VGA
AL = 0 do 3 dla trybu 0Eh: EGA, VGA
AL = 0 do 1 dla trybów 0Fh, 10h: EGA, VGA
AL = 0 do 7 dla trybów 00h, 01h, 02h, 03h: MCGA
AL = 0 do 7 dla trybów 00h, 01h: CGA
AL = 0 do 3 dla trybów 02h, 03h: CGA

Inne karty jak PCjr. czy Cordata mogą używać innych wartości rejestru AL. Wszelkie niezbędne dane powinny się znajdować w ich dokumentacji.

Wyjście: bez znaczenia

AH = 06h Przewijanie strony w górę

Funkcja ta jest używana do przewinięcia tekstu na ekranie o podaną liczbę linii. Dolne wiersze wypełniane są liniami pustymi. Rozmiar i położenie przewijanej części ekranu określają rejestry CH,CL i DH,DL.

Wejście: AH = 06h
Rejestr AL zawiera ilość linii do przewinięcia; zero oznacza cały ekran
Rejestr BH zawiera wartość atrybutu pustych linii
Rejestry CH i CL zawierają numery wiersza i kolumny lewego górnego rogu przewijanego okna
Rejestry DH i DL zawierają numery wiersza i kolumny prawego dolnego rogu przewijanego okna

Wyjście: bez znaczenia

AH = 07h Przewijanie strony w dół

Funkcja ta działa podobnie do funkcji o numerze 6, z tym że przewija ona tekst okienka ekranu w dół.

Wejście: AH = 07h
Rejestr AL zawiera ilość linii do przewinięcia; zero oznacza cały ekran
Rejestr BH zawiera wartość atrybutu pustych linii
Rejestry CH i CL zawierają numery wiersza i kolumny lewego górnego rogu przewijanego okna
Rejestry DH i DL zawierają numery wiersza i kolumny prawego dolnego rogu przewijanego okna

Wyjście: bez znaczenia

AH = 08h Odczytanie znaku i atrybutu w miejscu ustawienia kursora

Funkcja ta jest używana do odczytania znaku (i jego atrybutu), który znajduje się w miejscu ustawienia kursora.

Wejście: AH = 08h
Rejestr BH zawiera numer strony

Wyjście: Rejestr AH zawiera atrybut znaku
Rejestr AL zawiera kod ASCII znaku

AH = 09h Zapis znaku i atrybutu w miejscu ustawienia kursora

Funkcja jest używana do zmiany znaku wyświetlanego w miejscu ustawienia kursora.

Wejście: AH = 09h
Rejestr AL zawiera kod ASCII wyświetlanego znaku
Rejestr BH zawiera numer strony
Rejestr BL zawiera atrybuty nowego znaku
Rejestr CX określa ilość powtórzeń tej operacji
Rejestr CX określa ilość powtórzeń tej operacji.

W trybie tekstowym powoduje to wyświetlenie na kolejnych pozycjach identycznych znaków. Jeśli jest to konieczne wyświetlanie jest przenoszone do następnej linii. Natomiast w trybie graficznym wiersz nie jest zmieniany. W obu wypadkach pozycja kursora nie zmienia się.
Jeśli najstarszy bit rejestru BL będzie ustawiony (wartość 80h lub większa), to bieżąca i nowa zawartość ekranu zostaną przed wyświetleniem poddane operacji XOR.

Wyjście: bez znaczenia

AH = 0Ah Zapis znaku w miejscu ustawienia kursora

Funkcja jest używana do zmiany wyświetlanego znaku z zachowaniem jego atrybutu w miejscu ustawienia kursora.

Wejście: AH = 0Ah
Rejestr AL zawiera kod ASCII wyświetlanego znaku
Rejestr BH zawiera numer strony
Rejestr BL określa kolor znaku w trybie graficznym
Rejestr CX określa ilość powtórzeń tej operacji
Rejestry CX i BL mają takie samo znaczenie jak w funkcji 9.

Wyjście: bez znaczenia

AH = 0Bh Ustawienie palety kolorów

Funkcja ta jest używana do wybrania aktywnej palety kolorów lub wybrania z niej wyświetlanego koloru.

Wejście: AH = 0Bh
Rejestr BH zawiera identyfikator palety kolorów, 0 lub 1
Rejestr BL zawiera daną, która jest interpretowana zależnie od wartości BH
Jeśli rejestr BH zawiera 0, dana w BL określa kolory tła i ramki. Jeśli BH zawiera 1, dana w BL określa wybieraną paletę.

Wyjście: bez znaczenia

AH = 0Ch Wyświetlenie punktu

Funkcja ta powoduje wyświetlenie punktu w określonej pozycji. Zależnie od trybu można podać również jego kolor.

Wejście: AH = 0Ch
Rejestr AL określa kolor punktu
Rejestr BH zawiera numer strony graficznej
Rejestr DX zawiera numer wiersza (0 do 199 lub 349)
Rejestr CX zawiera numer kolumny (0 do 319 lub 639, zależnie od trybu)

Wyjście: bez znaczenia

AH = 0Dh Odczytanie koloru punktu

Funkcja ta powoduje odczytanie koloru podanego punktu.

Wejście: AH = 0Dh
Rejestr BH zawiera numer strony graficznej

Rejestr DX zawiera numer wiersza (0 do 199 lub 349)
Rejestr CX zawiera numer kolumny (0 do 319 lub 639)

Wyjście: Rejestr AL zawiera kolor podanego punktu.

AH = 0Eh Zapis znaku w trybie dalekopisowym

Funkcja wyświetla jeden znak w kodzie ASCII, a następnie przesuwa kursor na następną pozycję. Znaki sterujące traktowane są według definicji kodu ASCII i nie powodują wyświetlania związanych z nimi znaków graficznych.

Wejście: AH = 0Eh
Rejestr AL zawiera kod ASCII wyświetlanego znaku
Rejestr BH zawiera numer strony
Rejestr BL w trybie graficznym określa kolor znaku

Wyjście: bez znaczenia

AH = 0Fh Odczyt aktualnego trybu

Funkcja zwraca wartości, opisujące bieżący tryb pracy karty.

Wejście: AH = 0Fh

Wyjście: Rejestr AH zawiera daną, która określa liczbę kolumn na ekranie
Rejestr AL zawiera daną, która określa bieżący tryb
Rejestr BH zawiera numer bieżącej strony

AH = 10h Ustawienie rejestru palety kolorów

Funkcja pozwala określić paletę kolorów karty EGA, VGA, PS/2, PCjr. i Tandy 1000.

Wejście: AH = 10h
Rejestr AL może zawierać wartości 0, 1, lub 2, które określają: rejestr, paletę, ramkę lub wszystkie rejestry palety
AL = 0, BL zawiera rejestr palety, BH kolor
AL = 1, BH zawiera kolor
AL = 2, rejestry ES:DX wskazują 17-bajtowy obszar, który zawiera listę kolorów
AL = 3, na kartach EGA, VGA lub MCGA zanegowanie wartości bitów migania i rozjaśnienia
BL = 0 lub 1 określa intensywność rozjaśnienia lub miganie
Wartości rejestru AL powyżej 7 wybierają różne funkcje związane z odczytywaniem, zapisywaniem czy wybieraniem rejestrów palety. Wymagają one podania specjalnych kombinacji rejestrów, które mogą ponadto zależeć od typu karty.

Wyjście: Brak dla Al = od 0 do 3;
dla pozostałych funkcji może ono mieć jakieś znaczenie

Funkcje o numerach od AH = 11h do AH = 1Ch są przeznaczone dla kart EGA, VGA i nowszych modeli komputerów PC i PS/2. Wartości od AH = 1Dh do FFh są zarezerwowane lub używane w szczególny sposób zależny od aplikacji i konfiguracji systemu.

Przerwanie 11h Test konfiguracji sprzętu

Przerwanie pozwala stwierdzić, które z części systemu zostały zainstalowane i są dostępne.

Wejście: bez znaczenia

Wyjście: Rejestr AX zawiera słowo, którego bity określają istnienie pewnych zasobów sprzętowych (jeśli nie podano inaczej, to wartość jeden oznacza, że dane urządzenie jest zainstalowane; zero oznacza jego brak)

Bit	Znaczenie
0	Stacje dysków
1	Koprocesor 8087
2, 3	Wartość 11b oznacza, że na płycie głównej zainstalowane jest co najmniej 64k pamięci RAM; komputery PS/2: bit 2 oznacza obecność urządzenia wskaźnikowego, bit 3 nie jest używany
4, 5	Początkowy tryb video: 01b oznacza 40 x 25 kolor; 10b oznacza 80 x 25 kolor; 11b oznacza 80 x 25 monochromatyczny
6, 7	Liczba stacji dysków (bez znaczenia jeżeli bit 0 jest równy zero); 00b oznacza 1 stację; 01b oznacza 2 stacje; 10b oznacza 3 stacje; 11b oznacza 4 stacje
8	Nie używany (PCjr: oznacza obecność układu DMA)
9-11	Liczba portów szeregowych
12	Układ wejściowy manipulatora (PS/2: nie używany)
13	Nie używany (PS/2: oznacza obecność wewnętrznego modemu)
14, 15	Liczba wyjść równoległych

Przerwanie 12h Odczyt rozmiaru pamięci

Przerwanie zwraca rozmiar ciągłej pamięci liczony w blokach po 1K.

Wejście: bez znaczenia

Wyjście: Rejestr AX zawiera rozmiar dostępnej pamięci bez wliczania pamięci EMS czy pamięci rozszerzonej

Przerwanie 13h Obsługa dysków

Funkcje tego przerwania służą do sterowania i obsługi stacji dysków. Funkcje AH = 00h do 05h działają na wszystkich komputerach, które pracują w systemie MS-DOS. Funkcje od 06h wzwyż definiowane są różnie dla każdego typu maszyny. Ogólnie rejestr AH zawiera numer funkcji lub status układu; AL związany jest z liczbą sektorów; CH zawiera numer ścieżki, CL numer sektora; DH przechowuje numer głowicy (0 lub 1), a DL numer dysku (00h do 7Fh identyfikuje stację dysków, 80h do FFh określa dysk twardy). ES:BX może wskazywać bieżący bufor dyskowy.

AH = 00h Inicjacja systemu dyskowego

Funkcja używana jest do zainicjowania pracy sterownika dysków po wystąpieniu błędu we/wy.

Wejście: AH = 00h
Rejestr DL określa numer dysku; jeśli jego najstarszy bit jest ustawiony, to podany jest numer dysku twardego.

Wyjście: CF = 0 - udane zakończenie operacji
CF = 1 - błąd
Rejestr AH zawiera bajt stanu (Patrz AH = 01h)

AH = 01h Odczyt statusu dysku

Funkcja zwraca status ostatnio wykonanej operacji dyskowej.

Wejście: AH = 01h

Wyjście: Rejestr AH zawiera bajt stanu o znaczeniu:
00h = operacja prawidłowa
01h = nielegalny rozkaz dla sterownika dysków
02h = nie znaleziono znacznika adresu: zły sektor
03h = próba zapisu na zabezpieczoną dyskietkę
04h = nie znaleziono sektora
05h = błąd inicjacji dysku twardego
06h = nie ma dyskietki w mechaniźmie
07h = błędny parametr twardego dysku
08h = błąd w kanale DMA
09h = przekroczenie granicy adresowej dla kanału DMA
0Ah = błędny sektor na twardym dysku
0Bh = błędna ścieżka na twardym dysku
0Ch = niepoprawna ścieżka
10h = błąd sumy kontrolnej CRC
11h = poprawiony ECC
20h = błąd sprzętowy sterownika dysków
40h = błąd pozycjonowania głowicy
80h = przekroczony czas operacji
AAh = niegotowy twardy dysk
BBh = nie zdefiniowany błąd twardego dysku

AH = 02h Odczyt sektorów

Funkcja odczytuje do pamięci dane z jednego lub kilku sektorów.

Wejście: AH = 02h
Rejestr AL zawiera liczbę odczytywanych sektorów (od 1 do 9)
Rejestr CH określa numer ścieżki (od 0 do 39)
Rejestr CL określa numer początkowego sektora (od 1 do 9)
Rejestr DH zawiera numer głowicy (0 lub 1)
Rejestr DL określa numer dysku (od 0 do 3)

Rejestry ES:BX wskazują bieżący bufor dyskowy

Wyjście: CF = 0 - udane zakończenie operacji
CF = 1 - błąd
Rejestr AH = 0 lub status operacji (Patrz funkcja AH = 01h)
Rejestr AL = liczba odczytanych sektorów

AH = 03h Zapis sektorów

Funkcja zapisuje dane z bufora pamięci wskazywanego przez rejestry ES:BX, do sektorów dyskowych, określonych przez rejestry AL, CH, CL, DH i DL.

Wejście: AH = 03h
Rejestr AL zawiera liczbę zapisywanych sektorów
Rejestr CH określa numer ścieżki (od 0 do 39)
Rejestr CL określa numer początkowego sektora (od 1 do 9)
Rejestr DH zawiera numer głowicy (0 lub 1)
Rejestr DL określa numer dysku (od 0 do 3)
Rejestry ES:BX wskazują bieżący bufor dyskowy

Wyjście: CF = 0 - udane zakończenie operacji
CF = 1 - błąd
Rejestr AH zawiera status operacji
Rejestr AL = liczba zapisanych sektorów

AH = 04h Weryfikacja sektorów

Funkcja porównuje sumę kontrolną CRC sektora z sumą kontrolną wyliczoną na podstawie przechowywanych w nim danych.

Wejście: AH = 04h; Rejestry AL, CH, CL, DH i DL zawierają standardowe informacje

Wyjście: CF = 0 - udane zakończenie operacji
CF = 1 - błąd
Rejestr AH zawiera status operacji

AH = 05h Formatowanie ścieżki

Funkcja ta powoduje formatowanie ścieżek. Nie powinna ona być wykonywana bez pewności, że operacja ta jest niezbędna. Niewłaściwe jej użycie może doprowadzić do zniszczenia tablicy FAT, a tym samym do utraty wszystkich danych na dysku.
Funkcje o numerach AH = 06h i wyższych różnie się zachowują na różnych typach maszyn. Dokładny ich opis można znaleźć w dokumentacji.

Przerwanie 14h Obsługa portu szeregowego

Funkcje tego przerwania pozwalają inicjować, odczytywać, zapisywać dane oraz testować port szeregowy. Funkcje o numerach 04h i wyższych różnie się zachowują na różnych typach maszyn. Ogólnie rejestr DX określa numer portu: 0, 1, 2 lub 3, które odpowiadają urządzeniom COM1, COM2, COM3 i COM4. W komputerach PC i starszych typach PC/XT obsługiwane są tylko COM1 i COM2.

AH = 00h Inicjacja portu szeregowego

Funkcja ta używana jest do inicjacji portu szeregowego, wskazywanego przez rejestr DX, według wartości parametru zawartego w AL.

Wejście: AH = 00h
Rejestr DX określa port
Rejestr AL zawiera bajt, który określa parametry portu

Bit	Znaczenie
0, 1	wartość 10b określa słowo siedmiobitowe wartość 11b określa słowo ośmiobitowe
2	wartość 0b lub 1b określa 1 lub dwa bity stopu
3, 4	wartość 00b lub 10b określa brak sprawdzania parzystości, wartości 01b lub 11b określają parzystość parzystą lub nieparzystą
5-7	wartości od 000b do 111b określają prędkość transmisji w bodach jako 110, 150, 300, 600, 1200, 2400, 4800 i 9600

Wyjście: Rejestr AH zawiera bajt statusowy portu

Bit	Znaczenie
0	Układ gotowy
1	Błąd zgubienia znaku
2	Błąd parzystości
3	Błąd ramki
4	Stwierdzono przerwę transmisji
5	Pusty rejestr buforowy nadawczy
6	Pusty rejestr przesuwny nadawczy
7	Błąd przekroczenia czasu

Rejestr AL zawiera bajt stanu modemu

Bit	Znaczenie
0	Linia Clear-To-Send (CTS) zmieniła stan
1	Linia Data-Set-Ready (DSR) zmieniła stan
2	Linia Ring-Indication (RI) zmieniła stan
3	Linia Received-Line-Signal-Detector zmieniła stan
4	Stan linii Clear-To-Send (CTS)
5	Stan linii Data-Set-Ready (DSR)
6	Stan linii Ring-Indication (RI)
7	Stan linii Received-Line-Signal-Detector

AH = 01h Wysłanie znaku do portu szeregowego

Funkcja wysyła jeden znak do portu szeregowego wskazywanego przez rejestr DX.

Wejście: AH = 01h
Rejestr AL zawiera przesyłany bajt
Rejestr DX określa port

Wyjście: Rejestr AH zawiera stan operacji: sukces lub błąd, gdy jego 7 bit jest wyzerowany lub ustawiony; jeśli bit 7 jest ustawiony, to bity od 0 do 6 określają przyczynę wystąpienia błędu; ich znaczenie jest opisane w funkcji AH = 00h

AH = 02h Odczyt znaku z portu szeregowego

Funkcja pobiera jeden znak z portu szeregowego i umieszcza go w rejestrze AL.

Wejście: AH = 02h
Rejestr DX określa port

Wyjście: Rejestr AH zawiera stan operacji: sukces lub błąd, gdy jego 7 bit jest wyzerowany lub ustawiony; jeśli bit 7 jest ustawiony, to bity od 0 do 6 określają przyczynę wystąpienia błędu; ich znaczenie jest opisane w funkcji AH = 00h
Rejestr AL zawiera odczytany bajt

AH = 03h Pobranie statusu portu szeregowego

Funkcja pobiera status portu wskazywanego przez rejestr DX. Działa ona podobnie do funkcji AH = 00h, z tym, że nie wykonuje żadnej inicjacji portu ani też w żaden inny sposób nie wpływa na stan jego rejestrów.
Funkcje o numerach AH = 04h i wyższych różnie się zachowują na różnych typach maszyn. Dokładny ich opis można znaleźć w dokumentacji.

Przerwanie 15h Obsługa magnetofonu

Funkcje tego przerwania podzielone są na kilka kategorii. Funkcje AH = 00h do 03h związane są z obsługą magnetofonu kasetowego, urządzenia teraz już rzadko używanego. Do tego przerwania zostało dodanych wiele różnych funkcji. Są one jednak mało znane i są raczej charakterystyczne dla danej maszyny lub oprogramowania.

Przerwanie 16h Klawiatura

AH = 00h Odczytanie znaku z bufora klawiatury

Funkcja zwraca kod ASCII i numer kolejnego znaku z bufora klawiatury.

Wejście: AH = 00h

Wyjście: Rejestr AH zawiera numer znaku pobranego z bufora klawiatury
Rejestr AL zawiera kod ASCII tego znaku

AH = 01h Pobranie stanu bufora klawiatury

Funkcja sprawdza, czy w buforze klawiatury istnieje jeszcze nie przetworzony znak.

Wejście: AH = 01h

Wyjście: Znacznik ZF = 0 lub 1, wskazując obecność lub brak znaku w buforze klawiatury
Rejestr AH zawiera numer znaku
Rejestr AL zawiera kod ASCII znaku

AH = 02h Odczyt stanu klawiszy sterujących

Funkcja zwraca bajt stanu klawiszy sterujących.

Wejście: AH = 02h

Wyjście: Rejestr AL zawiera bajt, którego bity oznaczają stan klawiszy sterujących

Bit	Znaczenie
0	Naciśnięty prawy klawisz Shift
1	Naciśnięty lewy klawisz Shift
2	Naciśnięty klawisz Ctrl
3	Naciśnięty klawisz Alt
4	Aktywny stan Scroll-Lock
5	Aktywny stan Num-Lock
6	Aktywny stan Caps-Lock
7	Aktywny stan Insert

Funkcje o numerach AH = 04h i wyższych różnie się zachowują na różnych typach maszyn. Dokładny ich opis można znaleźć w dokumentacji.

Przerwanie 17h Obsługa drukarki

Funkcje tego przerwania pozwalają inicjować, odczytywać, zapisywać dane oraz testować port drukarki. Ogólnie rejestr AL zawiera drukowany znak, DX określa drukarkę (0 do 2), jej stan zwracany jest w rejestrze AH.

AH = 00h Wysłanie znaku do drukarki

Funkcja wysyła jeden znak do drukarki.

Wejście: AH = 00h
Rejestr AL zawiera drukowany znak
Rejestr DX określa drukarkę (0 do 2)

Wyjście: Rejestr AH zawiera stan drukarki po wykonaniu funkcji

Bit	Znaczenie
0	Przekroczenie czasu obsługi, drukarka nie jest podłączona, nie jest włączona, lub nie działa
1, 2	Nie używane
3	Błąd wejścia/wyjścia
4	Wybranie drukarki
5	Brak papieru

Bit	Znaczenie
6	Potwierdzenie przyjęcia znaku
7	Drukarka nie jest zajęta

AH = 01h Inicjacja drukarki

Funkcja jest używana do inicjacji drukarek IBM lub Epson i zgodnych z nimi dwoma bajtami o wartościach 08h i 0Ch (niektóre pozostałe drukarki mogą błędnie zareagować).

Wejście: AH = 01h
Rejestr DX określa drukarkę (0 do 2)

Wyjście: Rejestr AH zawiera stan drukarki po wykonaniu funkcji (Patrz funkcja AH = 00h)

AH = 02h Odczyt stanu drukarki

Funkcja zwraca aktualny stan drukarki.

Wejście: AH = 02h
Rejestr DX określa drukarkę (0 do 2)

Wyjście: Rejestr AH zawiera bajt stanu drukarki (Patrz funkcja AH = 00h)

Przerwanie 18h Wywołanie interpretera języka BASIC

Przerwanie to pozwala wywołać zapisany w pamięci ROM interpreter języka BASIC.

Przerwanie 19h Uruchomienie programu ładującego system operacyjny

Przerwanie to powoduje restart komputera. Do pamięci pobierana jest zawartość zerowego sektora dysku i od tego miejsca kontynuowane jest wczytywanie systemu operacyjnego. Zawartość pamięci RAM nie ulega zmianie.

Przerwanie 1Ah Obsługa zegara

Funkcje tego przerwania pozwalają na odczytanie i ustawienie czasu dnia liczonego jako liczba taktów, które upłynęły od ostatniej północy (18.2065 taktów na sekundę).

AH = 00h Odczytanie czasu

Funkcja zwraca aktualny czas zapisany w zegarze systemowym.

Wejście: AH = 00h

Wyjście: Rejestr AL zawiera wartość większą od zera, jeśli funkcja nie była wywoływana od ostatniej północy; oznacza więc on przepełnienie dobowe
Rejestry CX i DX zawierają starsze i młodsze słowo licznika zegara

AH = 01h Ustawienie czasu

Funkcja przesyła do zegara systemowego nową wartość, reprezentującą liczbę taktów od ostatniej północy.

Wejście: AH = 01h
Rejestry CX i DX zawierają starsze i młodsze słowo nowego licznika zegara

Funkcje o numerach AH = 02h i wyższych różnie się zachowują na różnych typach maszyn. Dokładny ich opis można znaleźć w dokumentacji.

Przerwanie 1Bh Naciśnięcie klawiszy Ctrl-Break

Dla większości użytkowników różnica między klawiszami Control-Break i Control-C nie jest zauważalna. Przerwanie to umożliwia rozdzielenie tych dwóch kombinacji. Każde wciśnięcie klawiszy Control-Break powoduje natychmiastowe wygenerowanie przez klawiaturę przerwania 1Bh, co oczywiście powoduje zawieszenie wykonywania przez procesor bieżącego zadania. Znaczenie tego przerwania zależy od jego procedury obsługi.
Natomiast kombinację Control-C DOS rozpoznaje, jak każdy inny znak, odbierając ją z bufora klawiatury. Po zinterpretowaniu tej kombinacji DOS ustawia specjalny znacznik, wykonuje normalną sekwencję zadań, a na końcu sprawdza stan tego znacznika. Dopiero wtedy uruchamia procedurę, która obsługuje pojawienie się tej kombinacji.
Zasadnicza różnica polega więc na tym, że przerwanie INT 1Bh pozwala na natychmiastowe przerwanie działania systemu.
Programista ma do wyboru wiele możliwości obsługi błędów. Kombinacja Control-C pozwala systemowi na dokończenie rozpoczętej operacji (na przykład zapisywania danych do pliku) przed wywołaniem procedury obsługi błędu. Kombinacja Control-Break jest szybsza i umożliwia zamieszczenie bardziej ostrej procedury obsługi, lecz niesie ze sobą ryzyko przerwania operacji DOS-u. Czasami dobrze jest więc po prostu jako procedurę obsługi przerwania INT 1Bh umieścić tylko instrukcję IRET lub podprogram, który ustawi znacznik Control-C. Wtedy aktualne zadanie będzie się wykonywało tak długo, aż DOS nie wykryje tej sytuacji.

Przerwanie 1Ch Przerwanie zegarowe

Przerwanie to wywoływane jest przez INT 08h. Standardową procedurę jego obsługi stanowi instrukcja IRET. Jednak programy, badające zależności czasowe, często instalują własne procedury obsługi. Ważne jest, by przed zmienieniem wektora przerwania zapamiętać poprzedni podprogram jego obsługi i zawsze, przed wykonaniem nowego, przekazywać do niego sterowanie. Innymi słowy, jeśli wywoływane jest przechwycone przerwanie, sterowanie przekazywane jest do nowej procedury. Jednym z jej pierwszych zadań powinno być wywołanie oryginalnego podprogramu. Wtedy jego instrukcja IRET spowoduje ponowne odzyskanie sterowania i możliwość wykonania swego zadania. Ostatni rozkaz procedury, czyli IRET, spowoduje ponowne uruchomienie przerwanego programu.
Ponieważ standardową obsługę tego przerwania stanowi sama instrukcja IRET, to kusząca może wydawać się myśl o zaniechaniu wywoływania poprzedniego programu obsługi. Jednakże w wypadku, gdy inna aplikacja przychwyciła już wcześniej to przerwanie, to zaniedbanie doprowadzi do ignorowania jej procedury. Efekt może być taki, że każdorazowe wywołanie takiego programu unieruchamia pozostałe, działające wcześniej zadania. Należy więc pamiętać, by zawsze przekazywać sterowanie poprzedniej procedurze obsługi.
Procedura obsługi przerwania INT 1Ch nie powinna być długa. Przerwanie to jest bowiem wywoływane przez INT 08h, które ma najwyższy priorytet wśród przerwań sprzętowych. Jeśli więc program obsługi będzie wykonywał się dłużej niż 55 milisekund, to po upływie tego czasu zostanie on przerwany przez INT 08h, które wywoła INT 1Ch, a przerwanie

ponownie uruchomi zbyt długą procedurę obsługi. Rezultatem będzie oczywiście zawieszenie się systemu.

Przerwanie 1Dh Tablica parametrów video

Wektor przerwania 1Dh nie wskazuje na kod. Wskazuje on natomiast tablicę, która zawiera początkowe wartości dla układu sterownika video 6845. Dlatego też nie należy wykonywać instrukcji, wywołującej to przerwanie. Spowoduje to bowiem załadowanie rejestrów CS:IP pierwszymi dwiema wartościami tablicy, rezultat takiego działania jest nie do przewidzenia. Wartości te nie powinny być również w żadnym przypadku modyfikowane. Jeśli bowiem układ 6845 otrzyma niepoprawne dane, to może niemal natychmiast zniszczyć kartę graficzną.
Wszelkie eksperymenty są ze wszech miar nie wskazane.

Przerwanie 1Eh Tablica parametrów dyskowych

Wektor przerwania 1Eh nie wskazuje na kod. Wskazuje on natomiast tablicę, która zawiera początkowe wartości dla układu sterownika dysków. Do tego przerwania odnoszą się również takie same uwagi jak do poprzedniego. Nie należy więc wykonywać instrukcji, która wywołuje to przerwanie, ani modyfikować żadnej z wartości w tablicy; rezultatem może być zupełna utrata danych.

Przerwanie 1Fh Tablica wzorów znaków

Wektor przerwania 1Fh nie wskazuje na kod. Wskazuje on natomiast tablicę, która zawiera dane, definiujące kształty znaków o wartościach kodu ASCII od 128 do 255. Każdy znak określany jest przez wartości bitów kolejnych 8 bajtów, pierwszy definiuje postać pierwszej jego linii, drugi, drugiej itd., aż do bajtu ósmego, definiującego linię ostatnią. Każdy znak ma więc rozmiar 8 x 8 pikseli.
Ponieważ wektor tego przerwania nie wskazuje kodu, to nie należy go wywoływać. Modyfikacja wartości w tej tablicy ma wpływ na postać znaków wyświetlanych na ekranie. Jeśli zostanie utracona kontrola nad tym procesem, wówczas konieczne będzie wyzerowanie komputera.

Przerwania systemowe MS-DOS, INT 20h do INT 0FFh

Elementy tablicy wektorów przerwań począwszy od 20h definiowane są przez MS-DOS. Są one bardzo różnych typów. Jedne wskazują na struktury danych, inne na kod dla procesora, część jest już przestarzała, część nie udokumentowana itd. Ponadto niektóre z nich mogą nie zachowywać się zgodnie z definicją, zależnie od producenta, wersji systemu operacyjnego, czy zainstalowanego oprogramowania.
Poniższe zestawienie oparte jest na informacjach, które udało się nam uzyskać od firm IBM, Microsoft i innych wytwórców oprogramowania, od wielu ludzi, którzy działają w The Waite Group i których spotkaliśmy w ciągu ostatnich dziesięciu lat. Informacja ta powinna zostać potraktowana jako przewodnik. Implementacje systemu MS-DOS różnią się w poszczególnych wersjach, zależą również od wytwórców. Programy rezydentne także często modyfikują działanie niektórych części systemu. Funkcje, które oficjalnie nazywane są "nie udokumentowane", "zarezerwowane" lub "nie używane" powinny być więc traktowane

z dużą dozą sceptycyzmu. Ich użycie powinno zostać poprzedzone wielorakimi testami na wszystkich typach systemów, na których program ma działać.

Przerwanie 20h Zakończenie programu

Przerwanie to przeznaczone jest do wewnętrznego użytku przez system MS-DOS. W programach użytkowych nie powinno być stosowane. Do poprawnego zakończenia należy używać funkcji 4Ch.

Wejście: CS = adres segmentowy przedrostka programu PSP

Wyjście: bez znaczenia

Przerwanie 21h Funkcje systemowe

Przerwanie 21h składa się z prawie 100 funkcji, które dostarczają wielu usług, takich jak otwieranie i zamykanie, odczyt i zapis plików, wejście z klawiatury, wyświetlanie na ekranie, pobieranie i ustawianie daty oraz zapewnia ono dostęp do kilku użytecznych zmiennych.
Ogólnie schemat wywoływania funkcji przedstawia się następująco. W rejestrze AH należy umieścić numer żądanej funkcji. Jej argumenty natomiast należy przekazać w pozostałych rejestrach. Wartości rejestrów, które ulegają zmianie, mogą zostać przed wywołaniem funkcji zapisane na stosie.
Następnie wykonuje się instrukcję INT 21h. Następuje wówczas wybranie na podstawie wartości rejestru AH odpowiedniej funkcji i jej wywołanie. Może ona zakończyć się pomyślnie lub spowodować powstanie błędu. W tym drugim wypadku najczęściej ustawiany jest znacznik przeniesienia, a kod błędu wpisywany jest do rejestru AX. Dobrą praktyką jest wykonywanie po każdej instrukcji INT 21h rozkazu JC, który przekazuje sterowanie do podprogramu obsługi zaistniałego błędu.
Uwaga: Jeśli nie podano inaczej, to wszystkie funkcje reagują na kombinacje klawiszy Control-Break i Control-C. Generowane jest wtedy przerwanie 23h.
Uwaga: Działanie niektórych funkcji, szczególnie tych przesyłających znaki na konsolę, nie da się przewidzieć, jeśli dane urządzenie zostało przekierowane. Należy być szczególnie ostrożnym przy ich stosowaniu w podprogramach zależnych od wykrycia znacznika końca pliku (EOF) lub, które mogą przesyłać dane, nie mieszczące się na dysku.

AH = 00h Zakończenie programu [Wszystkie wersje]

Funkcja ta jest starszą wersją podprogramu, który powoduje zakończenie bieżącego programu. W obecnych programach nie należy jej używać. Zamiast niej powinna być wywoływana funkcja 4Ch.
Funkcja ta zwalnia całą pamięć przydzieloną programowi, opróżnia wszystkie bufory plików (należy pamiętać o poprawnym zamknięciu bloków FCB przed jej wywołaniem, inaczej grozi to utratą nie zapisanych jeszcze danych).
Funkcja na podstawie bieżącego przedrostka PSP przywraca wektory przerwań INT 22h (zakończenie programu, PSP przemieszczenie 0Ah), INT 23h (Control-Break, PSP przemieszczenie 0Ch) i INT 24h (Błąd krytyczny, PSP przemieszczenie 0Eh). Sterowanie jest przekazywane pod adres umieszczony w PSP pod adresem 0Ah.

Wejście: CS = adres segmentowy przedrostka programu PSP

Wyjście: Nie ma

Funkcje związane: AH = 31h, AH = 4Ch

Przerwania związane: INT 20h

AH = 01h Odczyt znaku z konsoli z echem [Wszystkie wersje]

Funkcja używana jest do odczytania znaku z klawiatury (STDIN). Jeśli nie jest on dostępny, to funkcja oczekuje na wciśnięcie klawisza. Następnie znak ten zostaje wyświetlony na ekranie (echo), a jego kod ASCII zwracany jest w rejestrze AL. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.
Jeśli został wciśnięty klawisz z rozszerzonego zbioru znaków, to AL = 0. Dlatego też zawsze należy badać jego wartość. W wypadku wykrycia zera, funkcja ta powinna zostać ponownie wywołana, by odczytać kod znaku rozszerzonego.
Jeśli urządzenie STDIN zostało przekierowane, to działania funkcji nie da się przewidzieć. Nie wykrywa ona kombinacji Control-C, chyba że zmienna BREAK ma stan ON, ani znaku końca pliku EOF.

Wejście: Nie ma

Wyjście: Jeśli przy pierwszym wywołaniu AL > 0, AL = zwykły znak ASCII
Jeśli przy pierwszym wywołaniu AL = 0, w celu pobrania rozszerzonego kodu znaku ASCII należy wywołać funkcję 01h ponownie

Funkcje związane: AH = 06h, AH = 07h, AH = 08h, AH = 3Fh

AH = 02h Wysłanie znaku na konsolę [Wszystkie wersje]

Funkcja używana jest od wysłania znaku na ekran konsoli (STDOUT). Rejestr DL zawiera kod ASCII wyświetlanego znaku.

Wejście: DL = znak, który będzie wysłany na ekran [tylko wersja 1] lub do STDOUT [wersje 2, 3, 4]

Wyjście: Nie ma

Funkcje związane: AH = 06h, AH = 09h, AH = 40h

AH = 03h Odczyt znaku z portu dodatkowego [Wszystkie wersje]

Funkcja używana jest do odczytania znaku z urządzenia STDAUX (Standard Auxiliary Device, standardowo jest to COM1). Jeśli nie jest on dostępny, to funkcja oczekuje na jego pojawienie się. Jego kod ASCII zwracany jest w rejestrze AL. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.
Ogólnie urządzenie STDAUX nie ma wbudowanego mechanizmu buforowania, a MS-DOS nie dostarcza funkcji, która bada dostępność następnego znaku. Funkcja ta nie zapewnia żadnego zabezpieczenia przed nie skończonym oczekiwaniem. Zadaniem programisty jest napisanie procedury, która przeprowadzałaby odpowiedni test i wykonywałaby przesłanie do pamięci. Ponieważ żądania wielu aplikacji wymagają szybkiej reakcji na wykrycie pojawienia się znaku, to funkcja ta nie jest zbyt często stosowana. Najczęściej komunikacja z portem szeregowym odbywa się bez udziału BIOS-u czy MS-DOS-u. Sposób ten przedstawiony jest w rozdziale 14.

Wejście: Nie ma

Wyjście: AL = znak odczytany z pierwszego portu szeregowego [tylko wersja 1] lub z STDAUX [wersje 2, 3, 4]

Funkcje związane: AH = 3Fh

Przerwania związane: INT 14h

AH = 04h Wysłanie znaku do portu dodatkowego [Wszystkie wersje]

Funkcja używana jest do wysłania znaku do urządzenia STDAUX (Standard Auxiliary Device, standardowo jest to COM1). Jeśli nie jest ono gotowe, to funkcja oczekuje na możliwość przeprowadzenia operacji. Kod ASCII wysyłanego znaku powinien znajdować się w rejestrze DL. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.
Podobnie, jak funkcja 3, funkcja ta nie pozwala na określenie stanu urządzenia wyjściowego; wykorzystanie tylko tej funkcji nie gwarantuje powodzenia transmisji.

Wejście: DL = znak wysyłany na wyjście STDAUX

Wyjście: Nie ma

Funkcje związane: AH = 40h

Przerwania związane: INT 14h

AH = 05h Wysłanie znaku do drukarki [Wszystkie wersje]

Funkcja używana jest do wysłania znaku do standardowego portu drukarki (LPT1 lub PRN). Jeśli nie jest ona gotowa, wówczas funkcja oczekuje na możliwość przeprowadzenia operacji. Kod ASCII wysyłanego znaku powinien znajdować się w rejestrze DL. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.
Podobnie, jak funkcje 3 i 4, funkcja ta nie pozwala na określenie stanu urządzenia wyjściowego. Przed jej wykorzystaniem należy się upewnić o gotowości urządzenia do przeprowadzenia transmisji.

Wejście: DL = znak wysyłany na wyjście STDPRN

Wyjście: Nie ma

Funkcje związane: AH = 40h

AH = 06h Bezpośrednia komunikacja z konsolą [Wszystkie wersje]

Funkcja jest używana do odczytania lub wysłania każdego znaku ASCII (tzn. dokładna reprezentacja wartości bajtu, łącznie ze znakami sterującymi i pochodzącymi z rozszerzonego zbioru znaków) na konsolę (STDIN i STDOUT, ogólnie klawiatura i ekran). Oczywiście funkcja ta nie reaguje na wciśnięcie klawiszy Control-C.
W poniższym opisie przyjęto, że urządzenia STDIN i STDOUT nie zostały przekierowane. Aby odczytać znak z STDIN, należy do rejestru DL przesłać wartość 0FFh; jeśli znacznik zera jest wyzerowany, to odczytany znak znajduje się w AL. Jeśli znacznik zera jest ustawiony, wówczas urządzenie wejściowe nie przekazało żadnego znaku. Podobnie, jak w funkcji 1, AL = 00 oznacza, że został odczytany znak rozszerzony (przyjmując, że znacznik zera jest wyzerowany). Jego kod zostanie pobrany przy następnym wywołaniu tej funkcji.

Zapis znaku do urządzenia STDOUT odbywa się przez umieszczenie jego kodu w rejestrze DL (ale nie 0FFh) i wywołanie INT 21h.

Wejście: Jeśli DL <> 0FFh, znak z rejestru DL wysyłany jest do urządzenia STDOUT; w przeciwnym razie bezpośrednio odczytywane jest wejście konsoli

Wyjście: Bez znaczenia - przy bezpośrednim zapisie na konsolę; dla bezpośredniego odczytu ZF = 1 oznacza, że nie jest dostępny żaden znak, w przeciwnym razie - AL = kod znaku

Funkcje związane: AH = 01h,AH = 02h,AH = 07h,AH = 08h,AH = 3Fh,AH = 40h

AH = 07h Bezpośrednie odczytanie znaku z konsoli bez jego echa [Wszystkie wersje]

Funkcja jest używana do odczytania znaku z urządzenia STDIN (klawiatura) bez jego wysyłania do STDOUT (ekran). Jeśli znak nie jest dostępny, to funkcja oczekuje na wciśnięcie klawisza. Odczytany znak umieszczany jest w rejestrze AL. Podobnie, jak w funkcjach 1 i 6, jeśli AL = 00, drugie wywołanie tej funkcji zwraca kod rozszerzonego znaku. Funkcja ta nie reaguje na wciśnięcie klawiszy Control-C.

Wejście: Nie ma

Wyjście: AL = odebrany z STDIN znak

Funkcje związane: AH = 01h,AH = 06h,AH = 08h,AH = 3Fh,AH = 40h

AH = 08h Odczytanie znaku z konsoli bez jego echa [Wszystkie wersje]

Funkcja jest używana do odczytania znaku z urządzenia STDIN (klawiatura) bez jego wysyłania do STDOUT (ekran). Jeśli nie jest on dostępny, to funkcja oczekuje na wciśnięcie klawisza. Odczytany znak umieszczany jest w rejestrze AL. Podobnie, jak w funkcjach 1, 6 i 7, jeśli AL = 00, drugie wywołanie tej funkcji zwraca kod rozszerzonego znaku. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.

Wejście: Nie ma

Wyjście: AL = znak odebrany z STDIN

Funkcje związane: AH = 01h,AH = 06h,AH = 08h,AH = 3Fh,AH = 40h

AH = 09h Wysłanie na konsolę łańcucha znaków [Wszystkie wersje]

Funkcja jest używana do wysłania na konsolę (ekran) łańcucha znaków. Należy się upewnić, że ostatnim znakiem ciągu jest znak "$" (24h). W łańcuchu mogą wystąpić dowolne znaki ASCII od 00 do 0FFh (ale po pierwszym wystąpieniu znaku "$" działanie funkcji jest kończone, a znak ten nie jest przesyłany). Adres ciągu zawierają rejestry DS:DX. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.

Wejście: DS:DX = wskaźnik łańcucha znaków zakończonego znakiem "$"

Wyjście: Nie ma

Funkcje związane: AH = 02h, AH = 06h, AH = 40h

AH = 0Ah Buforowane wejście z konsoli z echem [Wszystkie wersje]

Funkcja ta używana jest to odczytania z urządzenia STDIN (klawiatura) wiersza tekstu. Odczytywane znaki wysyłane są do STDOUT (ekran). Funkcja kończy swoje działanie po odczytaniu znaku powrotu karetki (0Dh). Funkcja ta reaguje na wciśnięcie klawiszy Control-C. Rejestry DS:DX zawierają adres bufora. Jego pierwszy bajt powinien zawierać rozmiar bufora. Procedura odczytuje z klawiatury znaki i wstawia je do bufora począwszy od jego trzeciego bajtu. Swoją działalność kończy po wykryciu znaku powrotu karetki (0Dh). Znak ten jest również wyświetlany na ekranie i umieszczany w buforze. W drugim jego bajcie zostaje natomiast umieszczon liczba zapisanych w nim znaków.
W buforze może zostać umieszczonych o jeden mniej znaków niż jest to podane w jego pierwszym bajcie. Pozostałe znaki są gubione, a użytkownik jest powiadamiany o tym fakcie przez sygnał dźwiękowy.

Wejście: DS:DX = wskaźnik bufora wejściowego

Wyjście: Nie ma

Funkcje związane: AH = 01h,AH = 06h,AH = 07h,AH = 08h,AH = 3Fh,AH = 40h

AH = 0Bh Sprawdzenie stanu standardowego wejścia [Wszystkie wersje]

Funkcja jest używana do sprawdzenia stanu standardowego wejścia (klawiatura), tzn. czy w jej buforze są jeszcze nie przetworzone znaki. Funkcja zwraca w rejestrze AL wartość 0FFh, jeśli jeden lub kilka znaków czeka na odebranie; jeśli AL=00, to żaden znak nie jest dostępny. Wartość AL = 0FFh będzie zwracana tak długo, aż znaki nie zostaną odczytane jedną z funkcji 1, 6, 7, 8, 0Ah lub 3Fh. Funkcja ta reaguje na wciśnięcie klawiszy Control-C.

Wejście: Nie ma

Wyjście: AL = 0FFh jeśli dostępny jest znak z urządzenia STDIN; jeśli nie AL <> 0FFh

Funkcje związane: AH = 01h,AH = 06h,AH = 07h,AH = 08h,AH = 3Fh,AX = 4406h

AH = 0Ch Wyczyszczenie bufora klawiatury i wywołanie funkcji [Wszystkie wersje]

Funkcja ta powoduje opróżnienie bieżącej zawartości bufora STDIN (klawiatura), a następnie wywołuje jedną z funkcji 01h, 06h, 07h, 0Ah do pobrania nowych danych. Numer tej funkcji należy podać w rejestrze AL. Przy wyborze funkcji 0Ah należy pamiętać również o określeniu bufora wejściowego. Sposób zwracania wartości oraz reakcja na wciśnięcie klawiszy Control-C zależy od wartości rejestru AL.

Wejście: AL = numer jednej z funkcji przerwania INT 21h (01h, 06h, 07h lub 0Ah); wartości pozostałych rejestrów określone są przez funkcję wybraną w AL; jeśli AL zawiera wartość inną niż podane, to działanie funkcji ogranicza się tylko do opróżnienia bufora.

Wyjście: AL = znak (chyba że wywołano funkcję 0Ah); pozostałe rejestry zdefiniowane są przez wybraną funkcję

Funkcje związane: AH = 01h,AH = 06h,AH = 07h,AH = 08h,AH = 0Ah

AH = 0Dh Opróżnienie buforów dyskowych [Wszystkie wersje]

Funkcja ta używana jest do przepisania na dysk danych, znajdujących się w buforach systemowych. Nie powoduje ona zamknięcia plików, ani też nie modyfikuje zawartości katalogu. Do wykonania operacji zamknięcia plików powinna być wywołana funkcja 10h lub 3Eh. W systemie DOS 3.3 istnieje preferowana funkcja 68h, która automatycznie modyfikuje zawartość katalogu i tablicy FAT. Funkcja ta, z powodu swoich ograniczonych możliwości, powinna być traktowana tylko jako szybki sposób przesłania danych z pamięci na dysk przez procedury, obsługujące ważne lub nagłe sytuacje. Na przykład, może być zastosowana w podprogramie, który obsługuje wykrycie kombinacji klawiszy Control-C.

Wejście: Nie ma

Wyjście: Nie ma

Funkcje związane: AH = 10h, AH = 3Eh, AH = 68h

Przerwania związane: INT 25h, INT 26h

AH = 0Eh Wybór dysku [Wszystkie wersje]

Funkcja pozwala określić, który z dysków ma być dyskiem bieżącym. Jego numer należy umieścić w rejestrze DL (0 = A:, 19h = Z:). Funkcja w rejestrze AL zwraca liczbę dysków obecnych w systemie (minimalna liczba to dwa); Gdy w systemie istnieje tylko jedna stacja dysków elastycznych, to MS-DOS traktuje ją zarówno jako A: jak i B:.
MS-DOS w wersji 2.X pozwala na istnienie do 3Fh dysków, MS-DOS w wersji 3.X pozwala na istnienie do 1Ah dysków, MS-DOS w wersjach 3.X i 4.X zwraca minimalnie pięć dysków logicznych, chyba że zostanie to zmienione komendą LASTDRIVE w pliku CONFIG.SYS.

Wejście: DL = numer dysku (0 = A:,..., 19h = Z:)

Wyjście: AL = liczba dysków logicznych w systemie (0 = A:,..., 19h = Z:)

Funkcje związane: AH = 19h

Przerwania związane: INT 25h, INT 26h

AH = 0Fh Otwarcie pliku [Wszystkie wersje]

Funkcja ta pozwala na współpracę z plikami danych za pomocą bloków FCB (File Control Block). Jednakże preferowane jest raczej używanie deskryptorów plików niż bloków FCB. Blok FCB jest strukturą danych, która istnieje w systemie w celu zapewnienia zgodności z aplikacjami projektowanymi dla wcześniejszych wersji systemu MS-DOS (wersje 1.X), które w większości są przekształconymi programami napisanymi pod systemem CP/M (CP/M jest starszym systemem operacyjnym, teraz już zupełnie nie używanym).
Blok FCB to struktura danych, zawierająca pola, które określają nazwę dysku, pliku, jego rozszerzenie, rozmiar, czas i datę ostatniej modyfikacji oraz wiele innych danych, służących do identyfikacji położenia pliku oraz jego stanu.
Funkcji tej to należy używać do otwarcia istniejącego pliku, który nie został jeszcze otworzony, a który znajduje się na podanej ścieżce na bieżącym dysku. Jeśli jest on już otwarty lub też nie istnieje w wyspecyfikowanym katalogu, to funkcja zwracając w rejestrze AL wartość 0FFh kończy się błędem. W wypadku pomyślnego zakończenia operacji, pola

File Size, File Date i File Time wypełniane są informacją z katalogu, a wartości pól Current Record i Random Record ustawiane są na 0.

Wejście: DS:DX = adres nie otwartego bloku FCB

Wyjście: AL = 00h jeśli plik został otwarty pomyślnie; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 28h,
preferowaną funkcją, która otwiera plik jest AH = 3Dh

AH = 10h Zamknięcie pliku [Wszystkie wersje]

Przed wywołaniem tej funkcji rejestry DS:DX powinny zwierać adres nie otwartego bloku FCB. Instrukcja INT 21h wywołuje podprogram, który przegląda dysk, poszukując pliku wyspecyfikowanego w tym bloku. Jeśli operacja ta zakończy się pomyślnie, to na podstawie stanu bloku FCB, aktualizowane są dane w katalogu i funkcja zwraca w rejestrze AL wartość 0. W wypadku wystąpienia błędu rejestr AL zawiera daną 0FFh. Należy tu wspomnieć, że dobrą praktyką przy projektowaniu programów jest dokonanie niezbędnych operacji na pliku i jak najszybsze jego zamknięcie. Unika się w ten sposób zniszczenia danych o pliku, które to dane mogą powstać na skutek błędu w programie.

Wejście: DS:DX = adres otwartego bloku FCB

Wyjście: AL = 00h jeśli plik został zamknięty pomyślnie; AL = 0FFh w razie błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 28h,
preferowaną funkcją, która zamyka plik jest AH = 3Eh

AH = 11h Poszukiwanie pierwszego pliku [Wszystkie wersje]

Funkcja ta przeprowadza przegląd bieżącego katalogu na bieżącym dysku w celu znalezienia pierwszej nazwy pliku, która to nazwa odpowiada podanemu w nie otwartym bloku FCB wzorcowi. Jego adres podawany jest w rejestrach DS:DX. Jeśli operacja ta zakończy się pomyślnie, to w obszarze operacji dyskowych DTA tworzony jest duplikat tego bloku, a rejestr AL przyjmuje wartość 0. W wypadku wystąpienia błędu rejestr AL zawiera daną 0FFh. Kolejne wyszukiwania należy już wykonywać za pomocą funkcji AH = 12h.
Przed wywołaniem funkcji AH = 11h, należy najpierw odpowiednio wypełnić pola nie otwartego bloku FCB. Zgodnie z potrzebami należy podać nazwę bieżącego dysku oraz nazwę i rozszerzenie poszukiwanego pliku. Mogą one zawierać znaki maskujące. Pozostałe pola powinny zawierać wartość zero.
Do wyszukiwania plików ukrytych, systemowych czy katalogów należy, ustawiając wartość atrybutu na odpowiednio 02h, 04h lub 10h, użyć rozszerzonej struktury FCB. Każdy atrybut kodowany jest na jednym bicie, toteż wartość 06h oznacza wyszukiwanie plików ukrytych i systemowych, a 16 dodatkowo jeszcze katalogów. Odszukanie etykiety dysku wymaga podania wartości 08h.
Jeśli obszar operacji dyskowych DTA nie zostanie określony przez użytkownika, to funkcja ta wykorzysta obszar o przemieszczeniu 80h, który znajduje się w przedrostku PSP. Funkcja AH = 2Fh zwraca adres bieżącego obszaru DTA.

Wejście: DS:DX = adres nie otwartego bloku FCB

Wyjście: AL = 00h jeśli plik został znaleziony; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 28h, prferowaną funkcją, która rozpoczyna wyszukiwanie plików jest AH = 4Eh

AH = 12h Poszukiwanie kolejnego pliku [Wszystkie wersje]

Funkcja ta powinna być używana po wywołaniu funkcji AH = 11h. Jej działanie jest również bardzo podobne. Rejestry DS:DX powinny zawierać adres bloku FCB. Jeśli operacja ta zakończy się pomyślnie, to podobnie jak w funkcji AH = 11h, inicjowany jest bieżący blok FCB, a rejestr AL przyjmuje wartość 0. W wypadku wystąpienia błędu rejestr AL zawiera daną 0FFh.
W programie należy sprawdzić, czy aktualnie znaleziona nazwa jest tą, o którą chodzi. Jeśli nie jest to żądany plik, wówczas proces wyszukiwania można powtarzać, aż do spełnienia oczekiwanego warunku lub aż do otrzymania wartości AL = 0FFh oznaczającej, że w bieżącym katalogu nie ma już więcej plików, których nazwy pasują do podanego wzorca.

Wejście: DS:DX = adres bloku FCB zwrócony przez poprzednie wywołanie funkcji, która znajduje pierwszy lub kolejny plik

Wyjście: AL = 00h jeśli plik został znaleziony; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowaną funkcją, która kontynuuje wyszukiwanie plików jest AH = 4Fh

AH = 13h Usunięcie pliku [Wszystkie wersje]

Funkcja ta powoduje usunięcie pliku, którego nazwa i rozszerzenie znajdują się w nie otwartym bloku FCB. Jeśli zawierają one znaki maskujące, to usunięte zostaną wszystkie pliki, które pasują do tak określonego wzorca. Rejestry DS:DX powinny zawierać adres bloku FCB. Jeśli operacja ta zakończy się pomyślnie, wówczas rejestr AL przyjmuje wartość 0. W wypadku wystąpienia błędu rejestr AL zawiera daną 0FFh, co oznacza, że w bieżącym katalogu nie odnaleziono żadnego pliku, którego nazwa pasowałaby do podanego wzorca.

Wejście: DS:DX = adres nie otwartego bloku FCB

Wyjście: AL = 00h jeśli plik został skasowany; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowaną funkcją, która usuwa pliki, jest AH = 41h

AH = 14h Odczyt sekwencyjny [Wszystkie wersje]

Funkcja powoduje odczyt danych z pliku. Rejestry DS:DX powinny wskazywać otwarty blok FCB. Dane są odczytywane z kolejnych rekordów. Rekordy zaś są przechowywane w blokach.
Rozmiar rekordu wyznaczany jest na podstawie pola bloku FCB o nazwie Record Size Field. Zakłada się, że plik zawiera więcej niż jeden rekord oraz więcej niż jeden blok rekordów. Numery odczytywanych rekordów są ustalane na podstawie wartości pól Current Block i Current Record. Po wykonaniu operacji są one inkrementowane.
Po pomyślnym wykonaniu operacji, odczytany rekord umieszczany jest w bieżącym obszarze DTA, a rejestr AL zawiera wartość 0. Jeśli dane pliku zostały już wyczerpane, to AL przyjmie wartość 1, która oznacza, że w obszarze DTA nie zostały umieszczone żadne nowe

dane. Jeśli obszar DTA nie jest właściwy (zbyt mały, przekracza granicę segmentu), wówczas funkcja zwraca AL = 02h i nie jest podejmowana żadna operacja odczytu. Jeśli w trakcie odczytu zostanie napotkany znacznik końca pliku (EOF: znak 1Ah = Control-Z), to rejestr AL zawierał będzie wartość 03, a pobrane dane zostaną umieszczone w obszarze DTA. Pozostała część rekordu zostanie wypełniona zerami.

Wejście: DS:DX = adres otwartego bloku FCB

Wyjście: AL = kod zakończenia operacji
00h = dane zostały prawidłowo skompletowane
01h = dane pliku zostały już wyczerpane; koniec pliku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA
03h = skompletowana część danych; napotkany znak EOF

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowaną funkcją, która odczytuje dane z pliku, jest AH = 3Fh

AH = 15h Zapis sekwencyjny [Wszystkie wersje]

Funkcja powoduje przepisanie danych z bieżącego obszaru DTA do pliku określonego w bloku FCB. Jego adres powinien znajdować się w rejestrach DS:DX. Przy realizowaniu tej operacji wykorzystywane są te same pola bloku, jakich używała funkcja AH = 14h tzn. Record Size, Current Record, Current Block. Po wykonaniu funkcji wartości dwóch ostatnich pól są inkrementowane.
Podobnie jak funkcja AH = 14h, również i ta funkcja zwraca wartości, które oznaczają kod zakończenia operacji. Wartość AL = 0 oznacza jej pomyślny przebieg, AL = 1 oznacza przepełnienie dysku, a AL = 2 oznacza nieprawidłowy obszar DTA.

Wejście: DS:DX = adres otwartego bloku FCB

Wyjście: AL = kod zakończenia operacji
00h = dane zostały prawidłowo zapisane
01h = przepełnienie dysku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowaną funkcją, która zapisuje dane do pliku, jest AH = 40h

AH = 16h Utworzenie pliku [Wszystkie wersje]

Przed wywołaniem funkcji rejestry DS:DX powinny wskazywać nie otwarty blok FCB. Funkcja powoduje ona otwarcie pliku o nazwie zawartej w polu FileName bloku FCB lub utworzenie nowego. Uwaga: jeśli plik o podanej nazwie istnieje, to zmiennym, które go opisują zostaną przypisane wartości początkowe (np. jego długość zostanie ustawiona na zero), a to spowoduje zniszczenie jego poprzedniej zawartości. Funkcja ta może również utworzyć plik ukryty lub podkatalog, ale wtedy wymagane jest użycie rozszerzonej struktury FCB.

Wejście: DS:DX = adres nie otwartego bloku FCB

Wyjście: AL = 00h jeśli plik został utworzony; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowane funkcje to AH = 3Ch, Utworzenie pliku; AH = 5Ah,

Utworzenie pliku tymczasowego; AH = 5Bh Utworzenie nowego pliku

AH = 17h Zmiana nazwy pliku [Wszystkie wersje]

Przed użyciem tej funkcji należy tak zmodyfikować blok FCB, by zawierał on dwie nazwy plików. Pierwsza znajduje się w standardowej pozycji. Druga powinna wystąpić tuż za nią, o adresie 11h względem początku bloku. (Funkcja ta nie działa z rozszerzonym blokiem FCB). Po jej wywołaniu podejmowane jest w bieżącym katalogu wyszukiwanie pliku podanego w pierwszej nazwie. Jeśli operacja ta zakończy się pomyślnie, to nazwa ta zostanie zastąpiona drugą, a to spowoduje przemianowanie pliku. Funkcja zwraca wtedy w rejestrze AL wartość 0. W wypadku błędu AL zawiera daną 0FFh.

Wejście: DS:DX = adres zmodyfikowanego bloku FCB (nowa nazwa zaczyna się w polu, które zawiera numer aktualnego bloku)

Wyjście: AL = 00h jeśli plik został przemianowany; AL = 0FFh w wypadku błędu

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowana funkcja, która zmienia nazwę pliku, to AH = 3Ch

AH = 18h Nie udokumentowana

Nie używana przez DOS (stare wywołanie zgodne ze standardem CP/M).

AH = 19h Pobranie bieżącego dysku [Wszystkie wersje]

Funkcja używana jest do określenia bieżącego dysku. Numer od 0 do 19h zwracany w rejestrze AL identyfikuje dyski od a do Z.

Wejście: Nie ma

Wyjście: AL = numer bieżącego dysku (0 = A:,...,25 = Z:)

AH = 1Ah Ustawienie bufora operacji dyskowych [Wszystkie wersje]

Funkcja pozwala na zmianę bufora operacji dyskowych DTA. Musi on znajdować się w takim miejscu pamięci, aby nie leżał na granicy segmentów. Funkcja nie zwraca wartości, oznaczającej poprawność jej zakończenia. Błędy pojawią się dopiero przy próbach korzystania z nieprawidłowo zdefiniowanego bufora. Jego rozmiar może być właściwie dowolny, o ile jest mniejszy niż 64k i nie przekracza granicy segmentu. Odpowiednio do jego definicji muszą być również ustawiane zmienne w bloku FCB, jak na przykład Record Length. Obszar DTA, który jest zbyt mały, by pomieścić rekord pliku, powoduje powstawanie błędów przy operacjach czytania i pisania; zbyt duży powoduje marnotrawstwo pamięci. Jeśli bufor ten nie zostanie ustalony przez użytkownika, to system przyjmie domniemany obszar o długości 80h bajtów, który znajduje się w przedrostku programu PSP.

Wejście: DS:DX = wskaźnik na nowy obszar DTA

Wyjście: Nie ma

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;

AH = 1Bh Informacja o dysku bieżącym [Wszystkie wersje]

Funkcja zwraca informacje dotyczące bieżącego dysku. Rejestr CX zawiera liczbę bajtów w sektorze, liczba sektorów w grupie zawarta jest w rejestrze AL, a liczba grup na dysku umieszczona jest w DX. Ponadto bajt FAT ID określa typ nośnika. Może to być dyskietka jedno- lub dwu-stronna z 8, 9 lub 15 sektorami na ścieżce lub też twardy dysk

Wejście: Nie ma

Wyjście: DS:BX = wskaźnik na bajt, który zawiera FAT ID
DX = 0FFh DS, 8 sektorów
0FEh SS, 8 sektorów
0FDh DS, 9 sektorów
0FCh SS, 9 sektorów
0F9h DS, 15 sektorów
0F8h dysk stały
DX = liczba grup na dysku
AL = liczba sektorów w grupie
CX = liczba bajtów w sektorze

Funkcje związane: AH = 36h Ustalenie ilości wolnego obszaru na dysku; AH = 1Ch Informacja o wskazanym dysku

AH = 1Ch Informacja o wskazanym dysku [Wszystkie wersje]

Funkcja zwraca informacje o dysku wskazanym w rejestrze DL. Działa ona bardzo podobnie do funkcji AH = 1Bh, która udostępnia dane o dysku bieżącym.

Wejście: DL = numer dysku (0 = dysk bieżący, 1 = A:,...,26 = Z:)

Wyjście: DS:BX = wskaźnik na bajt zawierający FAT ID
DX = liczba grup na dysku
AL = liczba sektorów w grupie
CX = liczba bajtów w sektorze

Funkcje związane: AH = 36h; AH = 1Bh

Przerwanie związane: INT 13h

AH = 1Dh Nie udokumentowana

Nie używana przez DOS (stare wywołanie zgodne ze standardem CP/M).

AH = 1Eh Nie udokumentowana

Nie używana przez DOS (stare wywołanie zgodne ze standardem CP/M).

AH = 1Fh Pobranie bloku parametrów dysku

Funkcja ta nie jest udokumentowana, co oznacza, że działanie jej może być różne na różnych wersjach i implementacjach pochodzących od różnych producentów. W rejestrach DS:DX zwraca ona adres bloku parametrów dysku bieżącego (Drive Parameter Block - DPB).

Wejście: DL = 0

Wyjście: CF = 0, para DS:DX zawiera adres bloku DPB dla bieżącego dysku
CF = 1, AX zawiera kod błędu

AH = 21h Odczyt swobodny [Wszystkie wersje]

Funkcja powoduje odczytanie z pliku wskazanego rekordu i przesłanie go do bieżącego obszaru DTA. Rejestry DS:DX powinny wskazywać otwarty blok FCB z odpowiednio wypełnionymi polami Current Block, Record Size, Current Record, Relative Record.

Wejście: DS:DX = adres otwartego bloku FCB

Wyjście: AL = kod zakończenia operacji
00h = dane zostały prawidłowo skompletowane
01h = dane pliku zostały już wyczerpane; koniec pliku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA
03h = skompletowana część danych; napotkany znak EOF

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowaną funkcją, która odczytuje dane z pliku, jest AH = 3Fh

AH = 22h Zapis swobodny [Wszystkie wersje]

Funkcja powoduje zapis do pliku danych umieszczonych w aktualnym obszarze DTA. Podobnie, jak w funkcji AH = 21h, w bloku FCB należy ustawić odpowiednie wartości pól, w szczególności pole Relative Record.

Wejście: DS:DX = adres otwartego bloku FCB

Wyjście: AL = kod zakończenia operacji
00h = dane zostały prawidłowo zapisane
01h = przepełnienie dysku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowaną funkcją, która zapisuje dane do pliku, jest AH = 40h

AH = 23h Pobranie rozmiaru pliku [Wszystkie wersje]

Funkcja zwraca rozmiar pliku identyfikowanego przez nie otwarty blok FCB, wskazywany przez rejestry DS:DX. Przed jej wykonaniem należy również ustawić żądaną wartość pola Record Size (przemieszczenie 0Eh). Funkcja pobiera daną z pola File Size (przemieszczenie 0Ch) i oblicza wartość, którą umieści w polu Relative Record przez podzielenie rozmiaru pliku przez rozmiar jego rekordu, zaokrąglając wynik w górę do najbliższej wartości całkowitej.

Wejście: DS:DX adres nie otwartego bloku FCB

Wyjście: Jeśli AL = 00h, pole Relative Record = liczba rekordów w pliku
Jeśli AL = 0FFh, plik nie został znaleziony

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;
preferowaną funkcją to AH = 42h

AH = 24h Określenie rekordu przy dostępie swobodnym [Wszystkie wersje]

Funkcja ustawia wartość pola Relative Record w otwartym bloku FCB wskazywanym przez rejestry DS:DX. Przed jej wywołaniem należy również ustawić odpowiednie wartości pól Current Block (0Ch) i Current Record (20h). Na ich podstawie zostanie wyliczona nowa wartość pola Relative Record.

Wejście: DS:DX adres otwartego bloku FCB

Wyjście: Nie ma

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowana funkcja to AH = 42h

AH = 25h Ustawienie wektora przerwań [Wszystkie wersje]

Funkcja instaluje procedurę obsługi podanego przerwania, zmieniając odpowiedni element tablicy wektorów przerwań. Rejestr AL powinien zawierać jego numer, a DS:DX adres procedury ISR. Ostrzeżenie: nie należy bezpośrednio operować na tablicy IVT; MS-DOS śledzi zapisane w niej wartości i bezpośredni zapis może mieć nieoczekiwany skutek.

Wejście: AL = numer przerwania
DS:DX = adres nowej procedury obsługi przerwania

Wyjście: Nie ma

Funkcje związane: AH = 35h

AH = 26h Utworzenie nowego przedrostka programu [Wszystkie wersje]

Funkcja powoduje utworzenie nowego bloku PSP w segmencie pamięci podanym w rejestrze DX. Jest ona już przestarzała. Należy raczej użyć funkcji 4Bh.

Wejście: DX:0 = adres nowego bloku PSP

Wyjście: Nie ma

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowana funkcja to AX = 4B00h, ładowanie i uruchomienie programu

AH = 27h Swobodny odczyt bloku rekordów [Wszystkie wersje]

Funkcja powoduje odczytanie z pliku jednego lub kilku rekordów i przesłanie ich do bieżącego obszaru DTA. Rejestry DS:DX powinny wskazywać otwarty blok FCB, a rejestr CX liczbę pobieranych rekordów. W czasie wykonywania operacji pola Current Block, Current Record i Relative Record bloku FCB są odpowiednio aktualizowane. Obszar DTA powinien mieć dostateczną pojemność na pomieszczenie wszystkich pobranych rekordów.

Wejście: DS:DX = adres otwartego bloku FCB
CX = liczba odczytywanych rekordów

Wyjście: AL = kod zakończenia operacji

00h = dane zostały prawidłowo skompletowane
01h = dane pliku zostały już wyczerpane; koniec pliku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA
03h = skompletowana część danych; napotkany znak EOF
CX = liczba rzeczywiście odczytanych rekordów

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowaną funkcją, która odczytuje dane z pliku, jest AH = 3Fh

AH = 28h Swobodny zapis bloku rekordów [Wszystkie wersje]

Funkcja powoduje zapis do pliku jednego lub kilku rekordów z bieżącego obszaru DTA. Rejestry DS:DX powinny wskazywać otwarty blok FCB, a rejestr CX liczbę przesyłanych rekordów. Pierwszy rekord zostanie zapisany w pliku, w miejsce wskazywane przez pole Relative Record. W czasie wykonywania operacji pola Current Block, Current Record i Relative Record bloku FCB są odpowiednio aktualizowane. Rejestr CX zawiera liczbę faktycznie przetransmitowanych rekordów, a AL określa stan zakończenia funkcji.

Wejście: DS:DX = adres otwartego bloku FCB
CX = liczba zapisywanych rekordów

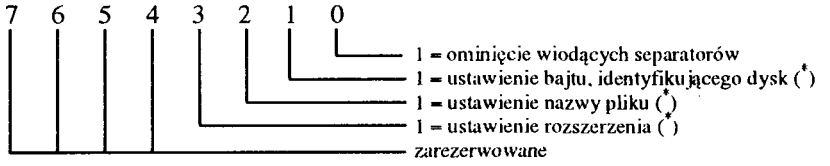
Wyjście: AL = kod zakończenia operacji
00h = dane zostały prawidłowo zapisane
01h = przepełnienie dysku
02h = operacja odczytu została odrzucona; zbyt mały obszar DTA
CX = liczba rzeczywiście zapisanych rekordów

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h; preferowaną funkcją, która zapisuje dane do pliku, jest AH = 40h

AH = 29h Analiza nazwy zbioru [Wszystkie wersje]

Funkcja przeprowadza analizę łańcucha znaków, pozwalając na wykrycie w nim składowych ścieżki dostępu do zbioru, takich jak dysk:nazwa_zbioru.rozszerzenie. Adres ciągu znaków zawierają rejestry DS:SI. Natomiast rejestry ES:DI wskazują na wystarczająco duży obszar pamięci, by pomieścić strukturę FCB. Sterowanie analizą odbywa się poprzez 4 najmłodsze bity rejestru AL; cztery najstarsze muszą być wyzerowane.
Funkcja zwraca w rejestrze AL daną, która informuje o wystąpieniu w nazwie znaków maskujących czy nieprawidłowej specyfikacji dysku. Jeśli operacja zakończy się pomyślnie, to para DS:SI zawiera adres pierwszego bajtu za poprawną specyfikacją dostępu do pliku, a jego nie otwarta struktura FCB wskazywana jest przez ES:DI.

Wejście: DS:SI = adres analizowanego łańcucha
ES:DI = adres bloku FCB
AL = sterowanie przebiegiem analizy



(*) = tylko, gdy część ta występuje w analizowanym łańcuchu

Wyjście: DS:SI = adres pierwszego bajtu za przeanalizowanym łańcuchem
ES:DI = adres wypełnionego bloku FCB
AL = kod zakończenia funkcji
00h = nie wykryto żadnych znaków wieloznacznych
01h = wykryto znaki wieloznaczne
0FFh = niepoprawna specyfikacja

Funkcje związane: AH = od 10h do 17h, AH = 21h do 24h, AH = 26h do 28h;

AH = 2Ah Pobranie daty [Wszystkie wersje]

Funkcja zwraca aktualną datę systemową.

Wejście: Nie ma

Wyjście: CX = rok (1980 do 2099)
DH = miesiąc (1 do 12)
DL = dzień (1 do 31)
AL = dzień tygodnia (0 = Niedziela)

Funkcje związane: AH = 2Bh, AH = 2Ch, AH = 2Dh

Przerwania związane: INT 1Ah

AH = 2Bh Ustawienie daty [Wszystkie wersje]

Funkcja powoduje ustawienie nowej daty w systemie.

Wejście: CX = rok (1980 do 2099)
DH = miesiąc (1 do 12)
DL = dzień (1 do 31)

Wyjście: AL = 0 jeśli data była podana poprawnie; AL = 0FFh w wypadku błędu

Funkcje związane: AH = 2Ah, AH = 2Ch, AH = 2Dh

Przerwania związane: INT 1Ah

AH = 2Ch Pobranie czasu [Wszystkie wersje]

Funkcja zwraca czas odczytany z zegara systemowego. Nie należy polegać na dokładności danej z rejestru DL. W niektórych systemach setne sekundy nie są w ogóle mierzone; w innych generowana jest tylko ich przybliżona wartość na podstawie generowanego, co 18.2 sekundy, przerwania INT 1Ch.

Wejście: Nie ma

Wyjście: CH = godziny (0 do 23)
CL = minuty (0 do 59)
DH = sekundy (0 do 59)
DL = setne sekundy (0 do 99)

Funkcje związane: AH = 2Ah, AH = 2Bh, AH = 2Dh

Przerwania związane: INT 1Ah

AH = 2Dh Ustawienie czasu [Wszystkie wersje]

Funkcja powoduje ustawienie nowego czasu w systemie.

Wejście: CH = godziny (0 do 23)
CL = minuty (0 do 59)
DH = sekundy (0 do 59)
DL = setne sekundy (0 do 99)

Wyjście: AL = 0 jeśli czas był podany poprawnie; AL = 0FFh w wypadku błędu

Funkcje związane: AH = 2Ah, AH = 2Bh, AH = 2Ch

Przerwania związane: INT 1Ah

AH = 2Eh Ustawienie/Zerowanie znacznika weryfikacji [Wszystkie wersje]

Funkcja służy do sterowania weryfikacją danych po każdej operacji zapisu na dysk. Wartość 0 załadowana do AL wyłącza ją, a 1 przywraca ponownie.

Wejście: AL = 00h co powoduje wyłączenie weryfikacji
AL = 01h co powoduje włączenie weryfikacji

Wyjście: Nie ma

Funkcje związane: AH = 15h, AH = 16h, AH = 22h, AH = 39h do 3Ch, AH = 54h, AX = 5701h, AH = 5Ah, AH = 5Bh

Przerwania związane: INT 13h, INT 26h

AH = 2Fh Pobranie adresu obszaru operacji dyskowych [Wersje 2, 3, 4]

Funkcja zwraca w parze rejestrów ES:BX adres bieżącego obszaru operacji dyskowych DTA. Powinna być wywołana przed podjęciem operacji zapisu/odczytu, które są przeprowadzane przy użyciu bloków FCB.

Wejście: Nie ma

Wyjście: ES:BX = adres bieżącego obszaru operacji dyskowych DTA

Funkcje związane: AH = 1Ah oraz funkcje zapisu i odczytu, działające na blokach FCB

AH = 30h Pobranie numeru wersji systemu MS-DOS [Wersje 2, 3, 4]

Funkcja zwraca numer wersji systemu operacyjnego. Jego pierwszą część zawiera rejestr AL, a drugą AH. Dodatkowo BH może zawierać numer seryjny producenta, a para BL:CX 24-bitowy numer seryjny użytkownika.

Wejście: Nie ma

Wyjście: AL = Pierwsza część numeru wersji (na lewo od kropki)
AH = Druga część numeru wersji (na prawo od kropki)
BX,CX = 0000

Uwaga: AX = 0, jeśli jest to MS-DOS wersja 1.X

AH = 31h Zakończenie programu bez usuwania go z pamięci [Wersje 2, 3, 4]

Funkcja ta pozwala na zainstalowanie rezydentnego kodu i przekazanie sterowania innemu programowi. Oczywiście program rezydentny musi posiadać pewien system aktywujący go, który powinien uwzględniać pewne nieprzewidywalne stany systemu np. podczas wykonywania funkcji przerwania INT 21h, odczytu lub zapisu na dysk lub wystąpienia błędu krytycznego. Istnieją trzy szerokie klasy aplikacji, które używają tej funkcji: programy instalujące swe własne procedury obsługi przerwań, programy typu "Hot-Key" aktywowane wciśnięciem pewnej kombinacji klawiszy i program działające w tle np. programy drukowania z buforami. Każda z nich ma swoje wady i zalety.
Ogólnie tego typu program powinien być projektowany jako część składowa procesu ładowania się systemu. Powinien więc przejrzeć struktury systemowe, zapamiętując ważne informacje, takie jak adres znacznika Critical-Break czy adresy procedur obsługi błędów krytycznych.
Tuż po załadowaniu program rezydentny działa jako program pierwszoplanowy. W tym czasie MS-DOS traktuje jego PSP, DTA, FCB i inne zmienne sterujące jako bieżące struktury danych. Przy wywołaniu funkcji AH = 31h, należy w rejestrze DX określić, jak wiele paragrafów pamięci powyżej adresu PSP ma zostać zarezerwowane dla programu.
Funkcja ta powoduje zwrócenie sterowania do poprzedniego programu, najczęściej jest to COMMAND.COM. Struktury programu takie jak PSP, kod, dane są nie naruszone, ale MS-DOS już nie traktuje ich jako bieżących. W efekcie, redukując obszar pamięci dostępny dla innych programów, stał się on częścią systemu operacyjnego.
Przy późniejszej aktywacji, program rezydentny zaczyna swe życie niejako na ślepo. MS-DOS traktuje przecież bieżący PSP, FCB, DTA i inne struktury sterujące jako należące do aktualnego programu pierwszoplanowego. Program rezydentny powinien być tak zaprojektowany, by nie została naruszona poprawność ani danych systemowych, ani danych programu pierwszoplanowego.

Wejście: AL = kod powrotu
DX = liczba paragrafów pamięci, które staną się rezydentne

Wyjście: Nie ma

Funkcje związane: AH = 4Ch

Przerwania związane: INT 27h

AH = 32h Odnalezienie bloku parametrów dla podanego dysku

Funkcja nie udokumentowana.
Funkcja ta działa tak jak funkcja AH = 1Fh, z tym że dodatkowo można określić dysk (0 to dysk bieżący, 1 to A: 1Ah to Z:). Jego numer należy przesłać do rejestru DL.

Wejście: DL = Numer dysku (0 = bieżący, 1 - 1Ah = A: - Z:)

Wyjście: Jeśli CF = 0,
AL = 00, poprawny dysk
AL = 0FFh, niepoprawny dysk
para DS:DX zawiera adres bloku DPB

Funkcje związane: AH = 1Fh, AH = 36h preferowana

AH = 33h Pobranie/Ustawienie sposobu reakcji na Control-C (Control-Break) [Wersje 2, 3, 4]

Funkcja jest używana do ustawienia lub pobrania sposobu reakcji na wciśnięcie klawiszy Control-C. Wartość AL = 0 powoduje odczytanie bieżącego stanu, a AL = 1 powoduje jego ustawienie zgodnie z wartością rejestru DL. Istnieje możliwość zastąpienia procedury obsługi tego zdarzenia własnym podprogramem.

Wejście: AL = 00h, pobranie bieżącego stanu
AL = 01h, ustawienie określonego sposobu reakcji na Control-C
DL = 00h, wyłączenie sprawdzania
DL = 01h, włączenie sprawdzania

Wyjście: AL = 0FFh jeśli błąd (AL nie jest równe ani 0 ani 1)
DL = 00h jeśli sprawdzanie jest wyłączone
DL = 01h jeśli sprawdzanie jest włączone

Funkcje związane: AH = 34h

Przerwania związane: INT 23h (nie należy go bezpośrednio wywoływać), INT 24h

AH = 34h Zwrócenie adresu znacznika sekcji krytycznej (Nie udokumentowana) [Znana w wersji 3.X]

Funkcja w rejestrach ES:BX zwraca adres znacznika sekcji krytycznej. Jeśli nie jest wykonywana żadna sekcja krytyczna, to wartość tego słowa jest równa zeru. Ustawienie tego znacznika oznacza, że jest wykonywana funkcja przerwania INT 21h i stan systemu jest nieokreślony. Dobrą praktyką jest włączenie sprawdzania stanu tego znacznika przy instalowaniu procedur obsługi przerwań lub programów rezydentnych, szczególnie wtedy, gdy korzystają one z plików lub funkcji INT 21h.
Ostrzeżenie: Funkcja AH = 34h nie powinna być wywoływana z procedur ISR oraz przez programy rezydentne. Ustalenie adresu znacznika powinno być częścią procedur instalujących je, czyli jeszcze przed przekazaniem sterowania. Po reaktywowaniu w podprogramie należy więc sprawdzić tylko wartość słowa pod zapamiętanym adresem. Ważne jest natomiast, by, jeśli znacznik jest ustawiony, nie używać wywołań funkcji INT 21h. Oczywiście ostrzeżenie to dotyczy również funkcji AH = 34h.

Wejście: Nie ma

Wyjście: ES:BX zawiera adres znacznika sekcji krytycznej

Funkcje związane: AH = 33h

Przerwania związane: INT 24h

AH = 35h Pobranie wektora przerwania [Wersje 2, 3, 4]

Funkcja zwraca w rejestrach ES:BX adres procedury obsługi przerwania o numerze podanym w AL. Adres ten nie powinien być odczytywany bezpośrednio.

Wejście: AL = numer wektora przerwania

Wyjście: ES:BX = adres procedury obsługi przerwania

Funkcje związane: AH = 25h

AH = 36h Ustalenie ilości wolnego miejsca na dysku [Wersje 2, 3, 4]

Funkcja zwraca informacje o dysku wskazanym w rejestrze DL. Rejestr BX zawiera liczbę wolnych grup na dysku. Rejestr CX zawiera liczbę bajtów w sektorze, liczba sektorów w grupie zawarta jest w rejestrze AX, a liczba grup na dysku umieszczona jest w DX.

Wejście: DL = numer dysku (0 = dysk bieżący, 1 = A:,...,26 = Z:)

Wyjście: BX = liczba wolnych grup
DX = liczba grup na dysku
CX = liczba bajtów w sektorze
Jeśli AX = 0FFFFh, podano niepoprawny numer dysku
Jeśli AX <> 0FFFFh, AX = liczba sektorów w grupie

AH = 37h Sprawdzenie lub zmiana znaku opcji (Nie udokumentowana)

Funkcja służy do sprawdzenia lub zmiany znaku, który w komendzie dla systemu określa opcję. Standardowo jest to znak slash ("/"). Rejestr AL powinien zawierać wartość 0 przy sprawdzaniu i 1 przy ustawianiu znaku umieszczonego w DL.

Wejście: AL = 0, pobranie
AL = 1, ustawienie nowego znaku z rejestru DL

Wyjście: Jeśli CF = 0
AL = 0, wtedy DL zawiera bieżący znak opcji
AL = 1, ustawienie
Jeśli CF = 1, AX zawiera kod błędu

AH = 38h Pobranie informacji o kraju [Wersje 2, 3, 4]

Przed wywołaniem tej funkcji rejestrom AL, BX, i DX należy przypisać poprawne wartości. Jeśli DX zawiera daną 0FFFFh, to funkcja zmienia bieżący kraj na podstawie danych zawartych w pozostałych rejestrach. Jeśli DX < 0FFFFh, wówczas funkcja zwraca informację o aktualnie ustawionym kraju. Jeśli CF = 0, BX zawiera kod kraju. Jeśli CF = 1, AX zawiera kod błędu. Dodatkowa informacja zwracana przez funkcję dostępna jest w strukturze danych wskazywanej przez rejestry DS:DX.

Wejście: AL = 00, pobranie informacji o kraju
AL = 01h do 0FFh oznacza kod kraju mniejszy niż 0FFh
AL = 0FFh dla kodów większych niż 0FFh
BX = kod kraju, jeśli AL = 0FFh
DS:DX = adres 34-bajtowej struktury danych
DX = 0FFFFh oznacza ustawienie nowego kraju [wersje 3 i 4]

Wyjście: Jeśli CF = 0, BX = kod kraju
Jeśli CF = 1, AX = kod błędu

Funkcje związane: AH = 65h

AH = 39h Stworzenie podkatalogu (MKDIR) [Wersje 2, 3, 4]

Przed wywołaniem funkcji rejestry DS:DX powinny zawierać adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu. Funkcja kończy się niepomyślnie, jeśli podano niepoprawną ścieżkę, jeśli katalog nie jest pusty, lub jeśli ścieżka określa katalog główny lub bieżący.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu

Wyjście: Jeśli CF = 1
Błędy: AX = 3, ścieżka dostępu nie znaleziona
AX = 5, niedozwolony dostęp

Funkcje związane: AH = 3Ah, AH = 3Bh

AH = 3Ah Usunięcie podkatalogu (RMDIR) [Wersje 2, 3, 4]

Przed wywołaniem funkcji rejestry DS:DX powinny zawierać adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu do usuwanego katalogu. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu

Wyjście: Jeśli CF = 1
Błędy: AX = 3, ścieżka dostępu nie znaleziona
AX = 5, niedozwolony dostęp
AX = 10h, próba usunięcia bieżącego katalogu

Funkcje związane: AH = 39h, AH = 3Bh

AH = 3Bh Zmiana bieżącego katalogu (CHDIR) [Wersje 2, 3, 4]

Przed wywołaniem funkcji rejestry DS:DX powinny zawierać adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu do katalogu. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0. W przeciwnym razie CF = 1, a AX zawiera wartość 3.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu

Wyjście: Jeśli CF = 1
Błędy: AX = 3, ścieżka dostępu nie znaleziona lub ścieżka określa plik, a nie katalog

Funkcje związane: AH = 39h, AH = 3Ah

AH = 3Ch Utworzenie pliku [Wersje 2, 3, 4]

Funkcja ta powoduje otworzenie nowego pliku na dysku i katalogu określonym ścieżką dostępu, której adres zawierają rejestry DS:DX. Rejestr CX powinien zawierać atrybuty pliku. Jeśli w ścieżce dostępu nie jest określony dysk lub katalog, to operacja będzie wykonana na dysku bieżącym w bieżącym katalogu. Funkcja nie sprawdza istnienia pliku o podanej nazwie. Jeśli taki plik już istnieje zawartość starego pliku jest bezpowrotnie tracona, chyba że atrybuty poprzedniego pliku nie pozwalają na przeprowadzenie na nim takiej operacji. Jeśli funkcja zakończy działalność pomyślnie, wówczas w rejestrze AX

umieszczany jest uchwyt nowego pliku, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu
CX = 00, plik zwykły
01, tylko do czytania
02, ukryty
04, systemowy
08, etykieta dysku
10h, nazwa podkatalogu
20h, podlegające archiwizacji

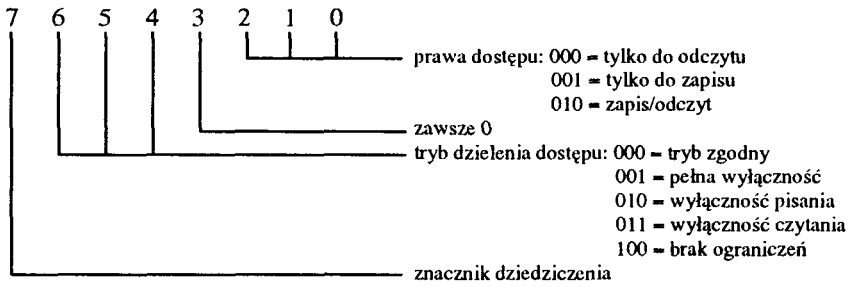
Wyjście: Jeśli CF = 0, AX = uchwyt pliku
Jeśli CF = 1
Błędy: AX = 3, nie znaleziona ścieżka dostępu
AX = 4, za dużo otwartych plików
AX = 5, katalog jest już pełen, katalog z podaną nazwą już istnieje, atrybuty istniejącego pliku o podanej nazwie nie pozwalają na jego utworzenie

Funkcje związane: AH = 3Dh, AH = 3Eh, AH = 3Fh, AH = 40h, AH = 4Eh

AH = 3Dh Otwarcie pliku [Wersje 2, 3, 4]

Funkcja ta powoduje otwarcie pliku na dysku i katalogu określonym ścieżką dostępu, której adres zawierają rejestry DS:DX. Rejestr AL powinien zawierać prawa dostępu do pliku. Jeśli operacja zakończy się pomyślnie, to w rejestrze AX umieszczany jest uchwyt pliku, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu
AL = tryb otwarcia:



Wyjście: Jeśli CF = 0, AX = uchwyt pliku
Jeśli CF = 1
Błędy: AX = 1, niepoprawny numer funkcji
2, nie znaleziono pliku
3, nie znaleziona ścieżka dostępu
4, za dużo otwartych plików
5, zabroniony dostęp
0Ch, niepoprawny dostęp

Uwaga: w wersji DOS 2.X nie jest możliwe otwieranie plików w sieci.

Funkcje związane: AH = 3Ch, AH = 3Eh, AH = 3Fh, AH = 40h, AH = 4Eh

AH = 3Eh Zamknięcie pliku [Wersje 2, 3, 4]

Funkcja zamyka plik, którego uchwyt znajduje się w rejestrze BX. Jeśli operacja zakończyła się pomyślnie, to na dysk zapisywane są wszystkie bufory z nim związane, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera wartość 6, która oznaczającą błędny uchwyt.

Wejście: BX = uchwyt pliku

Wyjście: Jeśli CF = 1

Błędy: AX = 6, błędny uchwyt pliku

Funkcje związane: AH = 3Ch, AH = 3Dh, AH = 3Fh, AH = 40h, AH = 4Eh

AH = 3Fh Odczyt z pliku lub urządzenia [Wersje 2, 3, 4]

Funkcja powoduje odczyt ciągu bajtów z pliku lub urządzenia. Urządzenie jest identyfikowane poprzez uchwyt, który znajduje się w rejestrze BX. Liczba transmitowanych bajtów powinna zostać umieszczona w rejestrze CX, a bufor, w którym zostaną one umieszczone, wskazywany jest parą DS:DX. Jego rozmiar musi odpowiadać wielkości odczytywanego bloku. Jeśli operacja zakończy się pomyślnie, to w rejestrze AX umieszczana jest liczba rzeczywiście odczytanych bajtów, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu.

Wejście: BX = uchwyt pliku
CX = liczba bajtów do odczytania
DS:DX = adres bufora

Wyjście: Jeśli CF = 0, AX = liczba rzeczywiście odczytanych bajtów
Jeśli CF = 1
AX = 5, niemożliwy odczyt
AX = 6, błędny uchwyt pliku

Funkcje związane: AH = 3Ch, AH = 3Dh, AH = 3Eh, AH = 40h, AH = 4Eh

AH = 40h Zapis do pliku lub urządzenia [Wersje 2, 3, 4]

Funkcja działa podobnie jak funkcje o numerach od AH = 3Ch do AH = 3Fh. Uchwyt pliku należy umieścić w rejestrze BX. Liczba transmitowanych bajtów powinna zostać umieszczona w rejestrze CX, a bufor, w którym się one znajdują, wskazywany jest parą DS:DX. Jeśli operacja zakończy się pomyślnie, to w rejestrze AX umieszczana jest liczba rzeczywiście zapisanych bajtów, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera kod błędu. Jeśli CF = 0, to wartości rejestrów AX i CX powinny być sobie równe; jeśli tak nie jest, to znaczy, że wystąpił błąd, mimo że CF = 0.

Wejście: BX = deskryptor pliku
CX = liczba bajtów do zapisania
DS:DX = adres bufora

Wyjście: Jeśli CF = 0, AX = liczba rzeczywiście zapisanych bajtów

Jeśli CF = 1
AX = 5, niemożliwy zapis
AX = 6, błędny deskryptor pliku

Funkcje związane: AH = 3Ch do AH = 3Fh, AH = 42h, AH = 4Eh

AH = 41h Usunięcie pliku [Wersje 2, 3, 4]

Funkcja powoduje usunięcie z katalogu pliku o nazwie określonej łańcuchem znaków ASCIIZ o adresie DS:DX. Nazwa ta musi być nazwą pliku, a nie podkatalogu. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0. W przeciwnym razie AX zawiera kod błędu.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu do pliku

Wyjście: Jeśli CF = 1
AX = 2, nie znaleziono ścieżki dostępu lub pliku
AX = 5, ścieżka dostępu określa katalog lub plik tylko do odczytu

Funkcje związane: AH = 43h

AH = 42h Przesunięcie wskaźnika pliku (LSEEK) [Wersje 2, 3, 4]

System operacyjny przechowuje dla każdego pliku 32-bitowy wskaźnik jego pozycji, czyli miejsca wykonania następnej operacji zapisu lub odczytu. Dzięki tej funkcji można pobrać lub zmienić aktualną jego pozycję dla pliku identyfikowanego przez uchwyt, zawarty w rejestrze BX. Rejestr AL określa rodzaj przesuwania, a para CX:DX powinna określać ilość bajtów przesunięcia. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0, a rejestry DX:AX zawierają nową wartość wskaźnika pozycji pliku. Jeśli przed wywołaniem w AL, CX i DX znajdowała się wartość 0, wówczas rejestry DX:AX będą zawierały aktualną wartość tego wskaźnika. W przeciwnym razie CF = 1 i AX zawiera kod błędu.

Wejście: CX:DX = zakres przesunięcia w bajtach
AL = tryb przesuwania
00 = od początku pliku
01 = od bieżącej pozycji pliku
02 = od końca pliku
BX = uchwyt pliku

Wyjście: Jeśli CF = 0, DX:AX = nowa wartość wskaźnika pozycji
Jeśli CF = 1
AX = 1, wartość AL poza zakresem
AX = 6, błędny uchwyt pliku

Funkcje związane: AH = 40h

AH = 43h Zmiana atrybutów pliku (CHMOD) [Wersje 2, 3, 4]

Funkcja umożliwia pobranie lub zmianę atrybutów pliku o nazwie wskazywanej przez rejestry DS:DX.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który definiuje ścieżkę dostępu
AL = 00h, pobranie atrybutów

AL = 01h, ustawienie atrybutów
CX = 00, plik zwykły
01, tylko do czytania
02, ukryty
04, systemowy
08, etykieta dysku
10h, nazwa podkatalogu
20h, podlegający archiwizacji

Wyjście: Jeśli CF = 0 i AL = 00h,
CL = atrybuty pliku
Jeśli CF = 1
AX = 1, wartość AL poza zakresem
AX = 3, ścieżka dostępu nie znaleziona
AX = 5, nie można zmienić atrybutów (katalog lub etykieta dysku)

Funkcje związane: AH = 41h

AH = 44h Sterowanie urządzeniami we/wy (IOCTL)

Funkcja ta składa się z szeregu podfunkcji, które pozwalają sterować urządzeniem, którego uchwyt znajduje się w rejestrze BX. Funkcje, które dokonują operacji zapisu i odczytu, używają bufora wskazywanego przez parę DS:DX. Jeśli operacja zakończy się sukcesem, to funkcja zwraca CF = 0, a AX zawiera kod zakończenia wywołanej funkcji. W przeciwnym razie CF =1 i AX zawiera kod błędu. Sterowanie urządzeniami znajduje się poza zakresem niniejszego rozdziału. W technicznej dokumentacji systemu MS-DOS można znaleźć szczegóły działania wszystkich podfunkcji.

00h = pobranie informacji o urządzeniu [wersje 2, 3, 4]

01h = zmiana trybu pracy urządzenia [wersje 2, 3, 4]

02h = odczyt z urządzenia znakowego [wersje 2, 3, 4]

03h = zapis do urządzenia znakowego [wersje 2, 3, 4]

04h = odczyt z urządzenia blokowego [wersje 2, 3, 4]

05h = zapis do urządzenia blokowego [wersje 2, 3, 4]

06h = sprawdzenie stanu urządzenia wejściowego [wersje 2, 3, 4]

07h = sprawdzenie stanu urządzenia wyjściowego [wersje 2, 3, 4]

08h = sprawdzenie, czy urządzenie blokowe jest wymienialne [wersje 3 i 4]

09h = sprawdzenie, czy urządzenie jest lokalne czy zdalne [wersje 3.1 3.2, 3.3, 4]

0Ah = czy deskryptor odnosi się do urządzenia lokalnego czy do zdalnego [wersje 3.1 3.2, 3.3, 4]

0Bh = określenie ilości powtórzeń przy błędzie współdzielenia plików [wersje 3 i 4]

0Ch = informacje sterujące dla klasy urządzeń (uchwyt) [wersje 3.3, 4]

0Dh = ogólne polecenia dla urządzeń blokowych [wersje 3.2, 3.3, 4]

0Eh = określenie urządzenia logicznego [wersje 3.2, 3.3, 4]

0Fh = ustawienie urządzenia logicznego [wersje 3.2, 3.3, 4]

AH = 45h Duplikacja uchwytu (DUP) [Wersje 2, 3, 4]

Funkcja powoduje utworzenie drugiego uchwytu dla otwartego pliku, którego uchwyt znajduje się w rejestrze BX. Jeśli operacja zakończyła się pomyślnie, to CF = 0 i AX zawiera nowo utworzony deskryptor. Od tego momentu operacje na pliku mogą odbywać się za pośrednictwem jednego z nich. Jednakże działanie na jednym z nich ma wpływ na zmienne sterujące drugiego (np. wskaźnik pozycji pliku). W wypadku wystąpienia błędu, CF = 1 a AX zawiera jego kod.

Wejście: BX = uchwyt istniejącego pliku

Wyjście: Jeśli CF = 0, AX = nowy zduplikowany uchwyt pliku
Jeśli CF = 1
AX = 4, zbyt dużo otwartych plików
AX = 6, błędny uchwyt pliku lub plik nie jest otwarty

Funkcje związane: AH = 3Ch do AH= 40h, AH = 42h, AH = 46h, AH = 4Eh

AH = 46h Wymuszona zmiana uchwytu pliku (FORCDUP) [Wersje 2, 3, 4]

Funkcja ta działa bardzo podobnie jak funkcja AH = 45h, z tym że w rejestrze CX podaje się wartość, która stanie się nowym uchwytem pliku.

Wejście: BX = uchwyt istniejącego pliku
CX = żądany uchwyt

Wyjście: Jeśli CF = 1
AX = 4, zbyt dużo otwartych plików
AX = 6, błędny uchwyt pliku lub plik nie jest otwarty

Funkcje związane: AH = 3Ch do AH= 40h, AH = 42h, AH = 45h, AH = 4Eh

AH = 47h Pobranie nazwy bieżącego katalogu [Wersje 2, 3, 4]

Funkcja zwraca ścieżkę dostępu do bieżącego katalogu. Rejestry DS:SI powinny wskazywać na 64-bajtowy bufor, a DL powinien określać numer dysku (0 dysk bieżący, 1 to A: itd.). Jeśli operacja zakończy się pomyślnie, to rejestry DS:SI wskazują bufor, w którym została umieszczona ścieżka dostępu do bieżącego katalogu, a znacznik CF = 0. W przeciwnym razie CF = 1, a AX zawiera wartość 0Fh, niepoprawny numer dysku.

Wejście: DS:SI = adres 64-bajtowego bufora
DL = numer dysku (0 = dysk bieżący, 1 = A:,...,26 = Z:)

Wyjście: DS:SI = adres, pod którym została umieszczona pełna ścieżka dostępu
Jeśli CF = 1
AX = 0Fh, niepoprawny numer dysku

Uwaga: Zwrócona ścieżka nie zawiera nazwy dysku oraz wiodącego znaku "\"

Funkcje związane: AH = 36h

AH = 48h Przydzielenie bloku pamięci [Wersje 2, 3, 4]

Funkcja powoduje przydzielenie programowi bloku pamięci. Rejestr BX powinien określać jego rozmiar w 16-bajtowych paragrafach. Jeśli operacja zakończy się pomyślnie, to CF = 0 a rejestr AX zawiera segmentowy adres przydzielonego bloku pamięci. W przeciwnym razie CF = 1, BX zawiera liczbę dostępnych paragrafów pamięci, a AX zawiera kod błędu.

Wejście: BX = żądany rozmiar bloku pamięci wyrażony w paragrafach

Wyjście: Jeśli CF = 0, AX:0 = adres przydzielonego bloku
Jeśli CF = 1
AX = 7, zniszczona wewnętrzna struktura pamięci systemowej
AX = 8, brak pamięci
BX = rozmiar największego wolnego bloku pamięci

Funkcje związane: AH = 49h, AH = 4Bh

AH = 49h Zwolnienie przydzielonego bloku pamięci [Wersje 2, 3, 4]

Funkcja ta powoduje zwolnienie przydzielonego funkcją AH = 48h bloku pamięci. Jego adres segmentowy powinien zawierać rejestr ES. Funkcja nie oznacza w jakiś specjalny sposób swego poprawnego zakończenia. W wypadku błędu CF = 1, a AX zawiera jego kod.

Wejście: ES = segment przydzielonego bloku pamięci

Wyjście: Jeśli CF = 1
AX = 7, zniszczona wewnętrzna struktura pamięci systemowej
AX = 8, ES wskazuje na obszar pamięci nie przydzielony funkcją AH = 48h

Funkcje związane: AH = 48h, AH = 4Ah, AH = 4Bh

AH = 4Ah Zmiana rozmiaru przydzielonego bloku pamięci (SETBLOCK) [Wersje 2, 3, 4]

Funkcja służy do zwiększenia lub zmniejszenia rozmiaru bloku pamięci przydzielonego funkcją AH = 48h. Rejestr ES powinien zawierać jego adres, a BX wartość, określającą nowy jego rozmiar. Jeśli operacja zakończy się pomyślnie, to CF = 0. W przeciwnym razie CF = 1, BX zawiera liczbę dostępnych paragrafów pamięci, a AX zawiera kod błędu.

Wejście: ES:0 = adres przydzielonego bloku pamięci, który ma być zmodyfikowany

Wyjście: Jeśli CF = 1
BX = maksymalny możliwy rozmiar bloku
AX = 7, zniszczona wewnętrzna struktura pamięci systemowej
8, brak pamięci
9, ES wskazuje na obszar pamięci nie przydzielony funkcją AH = 48h

Funkcje związane: AH = 48h, AH = 49h, AH = 4Bh

AH = 4Bh Załadowanie i uruchomienie programu (EXEC) [Wersje 2, 3, 4]

Funkcja powoduje załadowanie i uruchomienie programu. Po jej wywołaniu funkcja ładuje program do pamięci oraz tworzy jego blok PSP w oparciu o podany blok parametrów. Należy się upewnić, że istnieje wystarczająca ilość pamięci do pomieszczenia programu i jego danych oraz struktur sterujących MS-DOS-u (PSP, FCB, DTA, bloki sterowania pamięcią, itd.).

Przed wywołaniem funkcji należy przygotować pewne struktury danych. Rejestry DS:DX powinny wskazywać łańcuch znaków ASCIIZ, który określa nazwę uruchamianego programu. Para ES:BX zawiera adres 14-bajtowego bloku parametrów. Składa się on z czterech elementów: słowa zawierającego numer segmentu, w którym znajdują się zmienne środowiskowe, podwójne słowa będącego adresem linii komendy, która zostanie umieszczona w PSP, i dwóch podwójnych słów wskazujących na dwa bloki FCB, które zostaną również umieszczone w nowym PSP.

Po pomyślnym przeprowadzeniu tej operacji, CF = 0, a nowy program przejmuje sterowanie, wykonuje się, a po swym zakończeniu sterowanie zostanie przekazane do pierwszej instrukcji za wywołaniem tej funkcji. W wypadku wystąpienia błędu, funkcja zwraca CF = 1, a AX zawiera jego kod.

Wejście: DS:DX = adres nazwy pliku, w którym znajduje się uruchamiany programu
AL = określenie zadania
00h = załadowanie i wykonanie programu
03h = załadowanie nakładki
ES:BX = adres bloku parametrów
Jeśli AL = 00h

```
seg_env  dw ?        ; segment zmiennych środowiskowych
cmd_ptr  dd ?        ; adres linii komendy
fcb1_ptr dd ?        ; adres pierwszego FCB
fcb2_ptr dd ?        ; adres drugiego FCB
```

Jeśli AL = 03h

```
seg_load dw ?        ; segment, pod który należy
                     ; program załadować
rel_fact dw ?        ; współczynnik realokacji
```

Wyjście: Jeśli CF = 0, nie ma
Jeśli CF = 1
AX = 7, uszkodzona wewnętrzna struktura pamięci systemowej
8, brak pamięci
9, ES wskazuje niedostępny obszar pamięci
BX zawiera liczbę dostępnych paragrafów pamięci

Funkcje związane: AH = 48h do AH = 4Ah, AH = 4Ch

AH = 4Ch Zakończenie procesu (EXIT) [Wersje 2, 3, 4]

Funkcja ta jest zalecanym sposobem zakończenia programu i powoduje zwrócenie przez niego sterowania. Jest ono przejmowane przez program, który spowodował uruchomienie kończonego procesu. Zwykle jest to COMMAND.COM. Ponadto funkcja ta przekazuje do niego kod zakończenia programu potomnego. Przed jej wywołaniem należy go umieścić w rejestrze AL. Wartość ta może następnie być testowana, np. komendą IF w plikach wsadowych.

Wejście: AL = kod powrotu (wartość zmiennej ERRORLEVEL w plikach wsadowych)

Wyjście: Nie ma

Funkcje związane: AH = 31h, AH = 48h do 4Bh, AH = 4Dh

AH = 4Dh Pobranie kodu powrotu procesu potomnego [Wersje 2, 3, 4]

Funkcja zwraca kod zakończenia procesu potomnego. W rejestrze AH umieszczany jest stan jego powrotu, a AL zawiera wartość, którą przekazał proces w trakcie swego zakończania.

Wejście: Nie ma

Wyjście: AL = kod powrotu zwrócony przez podproces
AH = stan powrotu
00h = zakończenie normalne
01h = zakończenie spowodowane naciśnięciem Ctrl-Break
02h = zakończenie spowodowane wystąpieniem błędu krytycznego
03h = zakończenie programu, który pozostaje jako rezydentny

Funkcje związane: AH = 48h do 4Ch

AH = 4Eh Poszukiwanie pierwszego odpowiadającego pliku (FINDFIRST) [Wersje 2, 3, 4]

Funkcja przegląda wskazany katalog w poszukiwaniu pierwszego pliku, który odpowiada podanej nazwie. Adres nazwy wskazują rejestry DS:DX, a w rejestrze CX należy umieścić atrybuty poszukiwanego zbioru. Jeśli CX = 0, to funkcja rozpoznaje tylko pliki zwykłe. Jeśli CX = 1, wówczas funkcja wykrywa pliki zwykłe i tylko do odczytu. Jeśli CX = 2, to dodatkowo wyszukiwane są pliki ukryte itd.
Jeśli operacja zakończy się pomyślnie, wówczas funkcja zwraca CF = 0, i wypełnia bieżący obszar DTA siedmioma elementami o łącznym rozmiarze 43 bajtów. W przeciwnym razie CF = 1, a rejestr AX zawiera kod błędu.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który opisuje nazwę pliku
CX = atrybuty poszukiwanego pliku

Wyjście: Jeśli CF = 1
AX = 2, nie znaleziono pliku
18, nie ma pliku o podanej specyfikacji
Jeśli CF = 0, w bieżącym obszarze DTA umieszczana jest struktura o postaci:

```
reserved db 21 dup (?)          ; zarezerwowane
```

```
Attrib db ?           ; atrybuty pliku
Time   dw ?           ; czas ostatniej modyfikacji pliku
Date   dw ?           ; data ostatniej modyfikacji pliku
Size   dd ?           ; rozmiar pliku
Name   db 13 dup (?) ; nazwa pliku
```

Funkcje związane: AH = 4Fh

AH = 4Fh Poszukiwanie kolejnego odpowiadającego pliku (FINDNEXT) [Wersje 2, 3, 4]

Funkcja przegląda wskazany katalog w poszukiwaniu kolejnego pliku, który odpowiada poddanej nazwie. Jeśli operacja powiedzie się, to wypełniany jest bieżący obszar DTA, podobnie jak w funkcji AH = 4Eh. W wypadku wystąpienia błędu, CF = 1, a AX zawiera kod błędu.

Wejście: adres obszaru DTA zwrócony przez funkcję FINDFIRST lub poprzednie wywołanie funkcji FINDNEXT

Wyjście: takie samo jak przy wywołaniu funkcji FINDFIRST

Funkcje związane: AH = 4Eh

AH = 50h Ustawienie bloku PSP (Nie udokumentowana)

Funkcja traktuje wartość znajdującą się w rejestrze BX jako adres segmentu bloku PSP. Należy się upewnić, czy pod tym adresem znajduje się poprawnie stworzony przedrostek PSP.

Wejście: BX = adres nowego bloku PSP

Wyjście: CF = 0, sukces
CF = 1, poważne kłopoty

Funkcje związane: AH = 51h, AH = 55h

AH = 51h Pobranie adresu bieżącego bloku PSP (Nie udokumentowana)

Funkcja w rejestrze BX zwraca adres bieżącego bloku PSP.

Wejście: Nie ma

Wyjście: BX zawiera adres segmentu bieżącego bloku PSP

Funkcje związane: AH = 50h, AH = 55h

AH = 52h Pobranie adresu listy struktur sterujących (Nie udokumentowana)

W systemie MS-DOS istnieje lista, która przechowuje adresy wszystkich ważnych struktur systemowych takich jak tablica plików (System File Table - SFT), pierwszy blok zarządzania pamięcią (Memory Control Block - MCB), adresy programów obsługi urządzeń itd. Funkcja w rejestrach ES:BX zwraca adres tej listy.

Wejście: Nie ma
Wyjście: ES:BX zawierają adres listy

AH = 53h Nie udokumentowana

AH = 54h Pobranie stanu znacznika weryfikacji [Wersje 2, 3, 4]

Funkcja pozwala na sprawdzenie, czy po każdej operacji zapisu na dysk jest przeprowadzana weryfikacja. Wartość 1 zwrócona w rejestrze AL oznacza że tak, wartość 0, że nie.

Wejście: Nie ma

Wyjście: AL = 00h jeśli weryfikacja jest wyłączona; AL = 01h jeśli jest ona włączona

Funkcje związane: AH = 2Eh

AH = 55h Utworzenie bloku PSP (Nie udokumentowana)

Funkcja powoduje utworzenie nowego bloku PSP w segmencie o adresie podanym w DX. Ta nowa struktura zawiera wartości, które są kopią bieżącego bloku PSP.

Wejście: DX zawiera adres segmentu nowego bloku PSP

Wyjście: Nie ma

Funkcje związane: AH = 50h, AH = 51h

AH = 56h Zmiana nazwy pliku [Wersje 2, 3, 4]

Funkcja pozwala na zmianę nazwy pliku w katalogu określonym nazwą o adresie DS:DX. Rejestry ES:DI powinny zawierać adres jego nowej nazwy. Operacja musi dotyczyć tego samego dysku.

Wejście: DS:DX = wskaźnik starej nazwy [dysk:ścieżka/nazwa_pliku]
ES:DI = wskaźnik nowej nazwy [dysk:ścieżka/nazwa_pliku]

Wyjście: CF = 1
AX = 2, błędna ścieżka lub nie znaleziono pliku
5, zabroniony dostęp, DS:DX określa katalog, ES:DI wskazuje na nazwę istniejącego już pliku, nie można otworzyć pliku o nazwie wskazywanej przez ES:DI
17, niezgodność numerów dysków

AH = 57h Pobranie czasu i daty ostatniej modyfikacji pliku [Wersje 2, 3, 4]

Funkcja pozwala na pobranie lub zmianę czasu i daty ostatniej modyfikacji pliku, którego uchwyt znajduje się w rejestrze BX. Pobranie danych wskazywane jest przez umieszczenie wartości 0 w rejestrze AL. Po wykonaniu funkcji data i czas zostanie przesłany do rejestrów DX i CX. Ustawienie nowych wartości odbywa się przez przesłanie ich do rejestrów CX i DX i wywołanie funkcji.
Jeśli operacja zakończy się pomyślnie, to CF = 0; jeśli nie, wówczas CF = 1 i AX zawiera kod błędu.

Wejście: BX = uchwyt pliku
Jeśli AL = 0, pobranie czasu i daty
Jeśli AL = 1, CX zawiera nowy czas, a DX nową datę ustawianą w pliku

Wyjście: Jeśli CF = 0, CX zawiera czas, a DX datę ostatniej modyfikacji pliku
Jeśli CF = 1
AX = 1, AL zawiera błędną wartość
6, błędny deskryptor lub plik nie jest otwarty

AH = 58h Pobranie/Ustawienie strategii przydziału pamięci

Funkcja pozwala na pobranie lub zmianę strategii, jaką stosuje system operacyjny do przydzielania programowi bloków pamięci. Rozstrzyga o tym wartość umieszczona w rejestrze AL. Przy operacji ustawiania, rodzaj nowej strategii powinien być umieszczony w rejestrze BX. W wypadku pomyślnego zakończenia funkcji, CF = 0, a AX zawiera typ przydziału bloków. W przeciwnym razie CF = 1, a AX zawiera kod błędu.

Wejście: AL = 0, pobranie strategii
AL = 1, zmiana strategii
BX = 0, pierwszy wolny (pierwszy wolny blok o najniższym adresie)
1, najlepiej pasujący (najmniejszy wolny blok)
2, ostatni wolny (wolny blok o największym adresie)

Wyjście: CF = 0, AL = 0, nie ma wartości zwrotnych
CF = 0, AL = 1, AX zawiera typ strategii (0, 1, 2)
CF = 1, AX = 1, błędna wartość AL lub BX

AH = 59h Pobranie rozszerzonego kodu błędu [Wersje 3 i 4]

Funkcja powinna być wykonywana po wykryciu błędu. Funkcja pozwala na ustalenie możliwych przyczyn powstania błędu i sposobu dalszego postępowania. Przed wykonaniem funkcji należy na stos przesłać wartości rejestrów, a do BX wpisać wartość 0. Informacja zostanie zwrócona w rejestrach BH, BL i CH. W niektórych wypadkach jest ona jasna i niedwuznaczna np. BL = 5, natychmiastowe zakończenie. W innych konieczne jest przeprowadzenie dodatkowych badań sprzętowych lub zasobów systemowych.

Wejście: BX = 0000h

Wyjście: AX = rozszerzony kod błędu
BH = klasa błędu
1, brak zasobu
2, błąd przejściowy
3, błędne uprawnienia
4, błąd wewnętrzny systemu
5, błąd sprzętowy
6, błąd systemu
7, błąd programu użytkownika
8, nie znaleziono pliku
9, zły format
10, zablokowany zasób
11, uszkodzony nośnik dysku
12, inna
BL = zalecane działanie
1, powtórzenie, potem pytanie o decyzję użytkownika

2, powtórzenie z opóźnieniem
3, poprawienie wprowadzonych danych
4, normalne zakończenie programu
5, natychmiastowe zakończenie programu
6, zignorowanie
7, powtórzenie po interwencji użytkownika
CH = miejsce wystąpienia błędu
1, nieznane
2, urządzenie blokowe o dostępie swobodnym
3, sieć
4, urządzenie znakowe
5, pamięć RAM
CL, DX, SI, DI, ES, DS wartości tych rejestrów są niszczone

AH = 5Ah Utworzenie pliku tymczasowego [Wersje 3 i 4]

Funkcja pozwala na szybkie i bezpieczne utworzenie nowego pliku. System operacyjny zapewnia, że jego nazwa nie będzie się pokrywała z żadną z nazw plików już istniejących. Rejestry DS:DX powinny zawierać adres łańcucha znaków ASCIIZ, który określa dysk i katalog, w którym plik zostanie utworzony. Łańcuch ten powinien być zakończony znakiem slash ("\"), za którym powinno być jeszcze 13 bajtów wolnej pamięci. Rejestr CX powinien określać atrybuty tworzonego pliku.
Jeśli operacja zakończy się pomyślnie, CF = 0, a AX zawiera uchwyt utworzonego pliku, którego nazwa została dołączona na końcu ścieżki wskazywanej przez DS:DX.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który określa dysk i katalog, w którym plik zostanie utworzony. Łańcuch ten powinien być zakończony znakiem slash ("\"), za którym powinno być jeszcze 13 bajtów wolnej pamięci.
CX = atrybuty pliku

Wyjście: Jeśli CF = 0, AX = uchwyt pliku; DS:DX = adres łańcucha znaków ASCIIZ, który określa kompletną nazwę pliku
Jeśli CF = 1
AX = 3, błędna ścieżka
5, zabroniony dostęp

Funkcje związane: AH = 5Bh

AH = 5Bh Utworzenie nowego pliku [Wersje 3 i 4]

Funkcja powoduje utworzenie pliku o nazwie wskazywanej przez rejestry DS:DX i przypisanie mu atrybutów, które znajdują się w rejestrze CX. Jeśli operacja zakończy się pomyślnie, to CF = 0, a AX zawiera uchwyt utworzonego pliku. W wypadku wystąpienia błędu, CF = 1, a AX zawiera jego kod.

Wejście: DS:DX = adres łańcucha znaków ASCIIZ, który określa nazwę pliku
CX = atrybuty pliku

Wyjście: Jeśli CF = 0, AX = uchwyt pliku
Jeśli CF = 1
AX = 3, błędna ścieżka

5, zabroniony dostęp

Funkcje związane: AH = 5Bh

AH = 5Ch Zablokowanie/Odblokowanie dostępu do pliku [Wersje 3 i 4]

Funkcji tej należy używać tylko wtedy, gdy został zainstalowany program SHARE.EXE. Pozwala ona zablokować lub odblokować obszar pliku o przesunięciu podanym w rejestrach CX:DX i długości określonej w SI:DI. Funkcja ta powinna być wykorzystywana tylko w środowisku sieciowym.

Wejście: AL = 00h, zablokowanie dostępu do pliku; AL = 01h, odblokowanie dostępu do pliku
BX = uchwyt pliku
CX = starsze słowo przesunięcia
DX = młodsze słowo przesunięcia
SI = starsze słowo długości
DI = młodsze słowo długości

Wyjście: Jeśli CF = 1, AX = kod błędu

AH = 5Dh Nie udokumentowana

AX = 5E00h SIEĆ: Udostępnienie nazwy lokalnego komputera w sieci [Wersje 3.1, 3.2, 3.3, 4]

Funkcja zwraca pełną nazwę sieciową lokalnego stanowiska roboczego.

Wejście: DS:DX = adres 16-bajtowego bufora, w którym zostanie umieszczona nazwa

Wyjście: Jeśli CF = 0, DS:DX = adres nazwy komputera
Jeśli CF = 1, AX = kod błędu
Jeśli CH = 0, nazwa komputera nie jest znana w sieci
Jeśli CH <> 0, nazwa ta jest znana; CL = numer karty sieciowej

AX = 5E02h SIEĆ: Określenie ciągu znaków, które sterują drukarką sieciową [Wersje 3.1, 3.2, 3.3, 4]

Funkcja ta definiuje ciąg znaków, które mają poprzedzać treść każdego pliku tekstowego, wysłanego do drukarki sieciowej. Nagłówek ten może zawierać znaki sterujące.

Wejście: BX = pozycja w liście zasobów, która określa drukarkę
CX = długość łańcucha sterującego (maksymalnie 64 bajty)
DS:SI = adres bufora, który zawiera łańcuch sterujący

Wyjście: Jeśli CF = 1, AX = kod błędu

AX = 5E03h SIEĆ: Pobranie ciągu znaków, które sterują drukarką sieciową [Wersje 3.1, 3.2, 3.3, 4]

Funkcja pobiera ciąg znaków sterujących dla drukarki sieciowej. Jej działanie jest podobne do działania funkcji AX = 5302h.

Wejście: BX = pozycja w liście zasobów, która określa drukarkę
ES:DI = adres 64-bajtowego bufora

Wyjście: Jeśli CF = 0, CX = długość zwróconego łańcucha a ES:DI = jego adres
Jeśli CF = 1, AX = kod błędu

AX = 5F02h SIEĆ: Odczytanie pozycji z listy zasobów sieci [Wersje 3.1, 3.2, 3.3, 4]

Funkcja służy do ustalenia, które urządzenia zostały przypisane do zasobów sieciowych oraz jaka jest ich pozycja na tej liście.

Wejście: BX = pozycja w liście zasobów
DS:SI = adres 128-bajtowego bufora na nazwę lokalną
ES:DI = adres 128-bajtowego bufora na nazwę sieciową

Wyjście: Jeśli CF = 0, BH = stan urządzenia
Jeśli Bit 0 = 0, urządzenie określono poprawnie
Jeśli Bit 0 = 1, urządzenie określono niepoprawnie
BL = typ urządzenia
CX = wartość przypisana urządzeniu w chwili tworzenia pozycji w liście
DS:SI = nazwa lokalna
ES:DI = nazwa sieciowa
Jeśli CF = 1, AX = kod błędu

AX = 5F03h SIEĆ: Dodanie pozycji do listy zasobów [Wersje 3.1, 3.2, 3.3, 4]

Funkcja odwzorowuje lokalną nazwę na zasób sieciowy.

Wejście: BL = typ urządzenia:
03 = Drukarka
04 = Dysk
CX = 0000h
DS:SI = adres nazwy lokalnej
ES:DI = adres nazwy sieciowej

Wyjście: Jeśli CF = 1, AX = kod błędu

AH = 60h Nie udokumentowana

AH = 61h Nie udokumentowana

AH = 62h Pobranie adresu bloku PSP [Wersje 3 i 4]

Funkcja zwraca adres bieżącego bloku PSP. Zwykle jest on umieszczony pod adresem CS:0000 programu pierwszoplanowego. Pod pewnymi warunkami funkcja ta może być wywoływana z procedury obsługi przerwania programu sterownika urządzenia lub innego programu rezydentnego. Jeśli proces, korzystający z niej, nie jest programem pierwszoplanowym, to przed wywołaniem tej lub większości innych funkcji przerwania INT 21h, program musi podjąć odpowiednie środki ostrożności, zapewniające bezpieczeństwo systemu.

Wejście: Nie ma

Wyjście: BX:0 = adres bieżącego bloku PSP

AH = 63h Pobranie adresu tablicy DBCS

Funkcja jest dostępna tylko w wersji 2.25 systemu MS-DOS.

AH = 64h Nie udokumentowana

AH = 65h Pobranie rozszerzonej informacji o kraju [Wersje 3.3 i 4]

Funkcja dostarcza podobnej, ale bardziej szczegółowej informacji niż funkcja AH = 38h. Więcej detali można znaleźć w dokumentacji technicznej systemu MS-DOS.

Wejście: AL = identyfikator informacji
BX = tablica kodów (-1 = globalna tablica kodów)
DX = kod kraju (-1 = kraj bieżący)
CX = rozmiar
CX = adres bufora, w którym zostanie umieszczona informacja

Wyjście: Jeśli CF = 0, CX = rozmiar zwróconej informacji; ES:DI = bufor, który zawiera informację o kraju
Jeśli CF = 1, AX = kod błędu

Funkcje związane: AH = 38h, AH = 66h

AH = 66h Pobranie/Ustawienie numeru globalnej tablicy kodów [Wersje 3.3 i 4]

Funkcja pozwala na pobranie lub zmianę numeru globalnej tablicy kodów z pliku COUNTRY.SYS.

Wejście: AL = 01h, pobranie numeru globalnej tablicy kodów; AL = 02h, zmiana
BX = numer tablicy kodów (jeśli AL = 02h)

Wyjście: Jeśli CF = 0, BX = numer aktualnie używanej tablicy kodów, a DX = systemowa tablica kodów
Jeśli CF = 1, AX = kod błędu

AH = 67h Ustawienie liczby otwartych plików [Wersje 3.3 i 4]

Funkcja pozwala na określenie maksymalnej liczby plików, które program może równocześnie otworzyć.

Wejście: BX = maksymalna liczba otwartych plików

Wyjście: Jeśli CF = 1, AX = kod błędu

AH = 68h Opróżnienie buforów pliku [Wersje 3.3 i 4]

Funkcja powoduje efektywne zamknięcie pliku identyfikowanego przez uchwyt, który znajduje się w rejestrze BX. Zapisuje ona wszystkie bufory związane z plikiem na dysk i aktualizuje jego czas i datę ostatniej modyfikacji, a następnie dokonuje ponownego otwarcia pliku. Pewnym jej ograniczeniem jest to, że, gdy system musi zapisać bufory do urządzenia znakowego, zwraca CF = 0, ale operacji tej nie wykonuje.

Wejście: BX = uchwyt pliku

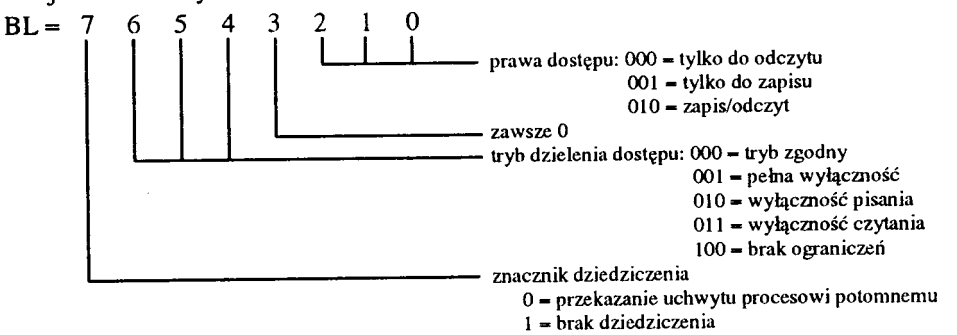
Wyjście: Jeśli CF = 1, AX = kod błędu

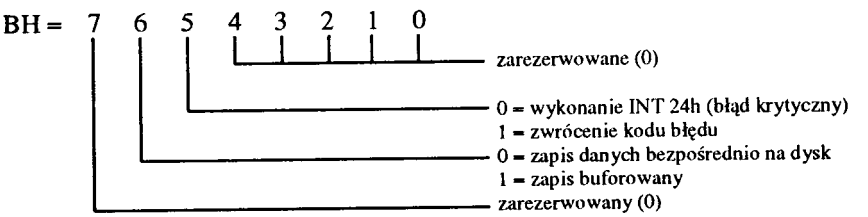
AH = 69h - AH = 6Bh Nie udokumentowane

AH = 6Ch Rozszerzone otwarcie/utworzenie pliku [Wersje 4]

Funkcja może być użyta do otwarcia, utworzenia lub zastąpienia pliku o nazwie wskazywanej przez rejestry DS:SI. Różnica pomiędzy wcześniejszymi funkcjami wynika przede wszystkim z dostępnych opcji, które sterują operacjami odczytu, zapisu, dziedziczenia, zarządzania buforami itd.

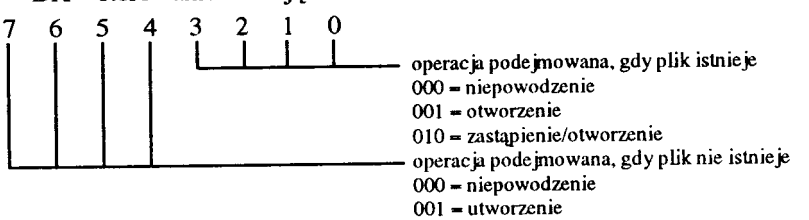
Wejście: BX = tryb otwarcia





CX = nowe atrybuty pliku (ignorowane przy otwieraniu pliku)

DX = sterowanie funkcją:



DS:SI = adres 64-bajtowej nazwy pliku

Wyjście: Jeśli CF = 0, AX = uchwyt pliku
CX = wykonana operacja
1 = plik otwarty
2 = plik otwarty/utworzony
3 = plik otwarty/zastąpiony

Jeśli CF = 1, AX = kod błędu

Przerwanie 22h Adres skoku po zakończeniu programu

Wektor tego przerwania wskazuje procedurę, która zostanie wykonana po zakończeniu bieżącego programu. Podczas procesu ładowania programu, system kopiuje ten adres do bloku PSP, umieszczając go w zmiennej o przemieszczeniu 0Ah względem jego początku. Uwaga: Procedura wskazywana tym przerwaniem jest przeznaczona tylko do wewnętrznego użytku przez DOS. Nie należy jej więc bezpośrednio wywoływać. Dwa pierwsze bajty bloku PSP to 0CDh i 20h. Jest to szesnastkowy kod instrukcji INT 20h. Przyjmując, że blok PSP umieszczony jest w pierwszych 256 bajtach segmentu wskazywanego przez rejestr CS, ustawienie wartości rejestru IP na zero powoduje wykonanie tego przerwania i zakończenie programu.

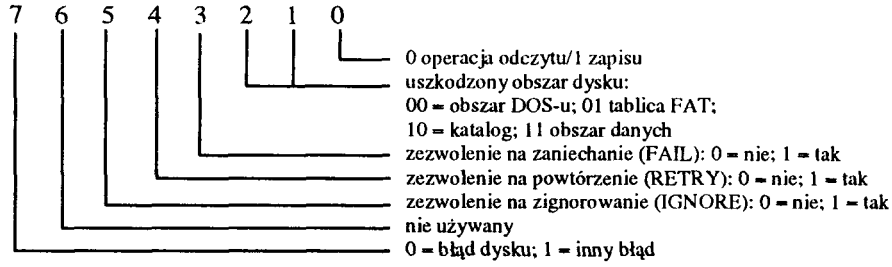
Przerwanie 23h Adres skoku po naciśnięciu klawiszy Control-C

Wektor tego przerwania wskazuje procedurę obsługi sytuacji wyjątkowej, powstałej wskutek naciśnięcia klawiszy Control-C; nie jest to procedura obsługi przerwania, dlatego też to przerwanie nie powinno być nigdy wywoływane.
Normalna praca systemu DOS polega na wykonywaniu funkcji przerwania INT 21h. W jej trakcie sprawdzany jest stan znacznika wciśnięcia klawiszy Control-C. Jeśli jest on ustawiony, wykonywana jest procedura wskazywana tym przerwaniem. Powszechną praktyką jest instalowanie własnych podprogramów, które przejmują obsługę tej sytuacji.
Naciśnięcie klawiszy Control-C różni się od naciśnięcia Control-Break. Wykonanie procedury obsługi Control-C może być tylko rezultatem wykonania jednej z większości funkcji systemowych. Pozwala to więc na ich poprawne zakończenie przed uruchomieniem procedury obsługi Control-C. Natomiast wciśnięcie klawiszy Control-Break powoduje natychmiastowe uruchomienie procedury o adresie przechowywanym w wektorze przerwania INT 1Bh (która również nie jest procedurą obsługi przerwania). Różnica ta pozwala stosować podprogramy obsługi tych sytuacji, które pozwalają na prawidłowe dokończenie działalności systemu DOS i takie, które pozwalają przerwać aktualną operację w dowolnym momencie, być może w ten sposób zapobiegając katastrofie.

Przerwanie 24h Adres procedury obsługi błędu krytycznego [Wersje 3 i 4]

System DOS wywołuje to przerwanie po wykryciu nieodwracalnego błędu we/wy. Standardowo powoduje to wypisanie komunikatu "Abort, Retry, Ignore?". Tabela C-1 zawiera kody i nazwy błędów.

Wejście: AH = bity



Wejście: BP:SI = adres nagłówka urządzenia sterującego, z którego może być pobrana dodatkowa informacja
DL = kod błędu urządzenia, jak poniżej

Wyjście: AL = 0, zignoruj błąd
1, powtórz operację
2, zakończ program przerwaniem INT 23h
3, powrót do programu użytkownika z sygnalizacją błędu

Tabela C-1. Kody błędów dla przerwania INT 24h

Kod błędu	Nazwa błędu
0	Próba zapisu na dysk zabezpieczony przed zapisem
1	Nieznana jednostka
2	Stacja dysków nie jest gotowa
3	Nieznane polecenie
4	Błąd danych (CRC)
5	Błędna długość struktury polecenia
6	Błąd pozycjonowania
7	Nieznany typ nośnika
8	Nie znaleziono sektora
9	Brak papieru w drukarce
A	Błąd zapisu
B	Błąd odczytu
C	Błąd ogólny

Przerwanie 25h Fizyczny odczyt kolejnych sektorów dysku [Wszystkie wersje]

Przerwanie to powoduje wywołanie procedury, która wykonuje operację odczytu kolejnych sektorów z bieżącego dysku począwszy od sektora o podanym numerze nie większym niż 64k. Dane są umieszczane w bieżącym obszarze DTA.
Funkcja ta istnieje we wszystkich wersjach systemu MS-DOS, ale tylko wersja 4 dostarcza funkcję, która może obsługiwać dyski o pojemnościach większych niż 32M. Ponadto do wersji 3.3 przemieszczenia sektorów mogły mieć wartość tylko do 64k, podczas gdy w wersji 4 można operować do 1G sektorami. Oczywiście tablica FAT musi mieć wtedy inną organizację.
Ostrzeżenie: funkcja ta powinna być stosowana z rozwagą. Odczytuje ona bowiem zadaną liczbę sektorów i przesyła do bufora DTA. Przed jej wywołaniem należy jednakże upewnić się, czy prawidłowo została obliczona liczba bajtów, przypadająca na jeden sektor. Błędne jej ustalenie może doprowadzić do przepełnienia bufora DTA.
=< 32M:

Wejście: AL = numer dysku (0 = A, 1 = B, itd.)
CX = liczba sektorów do odczytu

DX = logiczny numer sektora początkowego
DS:BX = adres obszaru DTA

Wyjście: CF = 0, sukces
CF = 1, błąd
AX = kod błędu
>32M [tylko wersja 4]:

Wejście: AL = numer dysku (0 = A, 1 = B, itd.)
BX = adres 10-bajtowej listy parametrów
32-bitowe podwójne słowo, które określa numer pierwszego sektora do odczytu
16-bitowe słowo określające ilość odczytywanych sektorów
32-bitowe podwójne słowo, które określa adres obszaru DTA
CX = -1 (oznacza format rozszerzony > 32M)

Wyjście: CF = 0, sukces
CF = 1, błąd
AX = kod błędu

Przerwanie 26h Fizyczny zapis do kolejnych sektorów dysku [Wszystkie wersje]

Jest to dopełnienie przerwania INT 25h. Powoduje ono zapis danych z podanego obszaru DTA do jednego lub kilku sektorów na dysku bieżącym. W systemie MS-DOS wersja 4 można dokonywać tej operacji również na dysku o pojemności większej niż 32M.
Ostrzeżenie: Podobnie jak przerwanie INT 25h, funkcja ta bezpośrednio przesyła dane z podanego obszaru pamięci na dysk. Jedyną wartością, jaką można sterować, jest liczba sektorów podawana w rejestrze CX. Przed wywołaniem funkcji należy prawidłowo wyznaczyć rozmiar sektora oraz upewnić się, czy sektory, do których ma być wykonywany zapis, są dostępne. W wypadku błędnego wyznaczenia danych może dojść do zamazania innych programów lub danych.
=< 32M:

Wejście: AL = numer dysku (0 = A, 1 = B, itd.)
CX = liczba sektorów do zapisu
DX = logiczny numer początkowego sektora
DS:BX = adres obszaru DTA

Wyjście: CF = 0, transmisja poprawna
CF = 1, błąd
AL = kod błędu
AH = 80h jeśli dysk nie odpowiada
40h = błąd pozycjonowania
08h = błędna suma kontrolna CRC podczas operacji odczytu
04h = nie znaleziono żądanego sektora
03h = próba zapisu na zabezpieczoną dyskietkę
02h = inny błąd

AX = 0207h jeśli wystąpił błąd podczas operacji zapisu/odczytu w formacie rozszerzonym przerwania INT 25h/26h

Przerwanie 27h Zakończenie programu bez usuwania go z pamięci [Wszystkie wersje]

Przerwanie to jest dość wczesnym zbiorem podprogramów, które są używane do zakończenia programu pierwszoplanowego bez usuwania go z pamięci (tym samym program ten staje się rezydentny). Przerwania tego nie należy jednak używać. Zastępuje je funkcja AH = 31h przerwania INT 21h.

Wejście: CS = adres segmentu przedrostka programu PSP
DX = adres końca żądanej pamięci

Wyjście: Nie ma

Uwaga: Funkcja nie powoduje zamknięcia plików

Przerwanie 28h Pętla oczekiwania na gotowość klawiatury

Przerwanie to jest automatycznie wywoływane przez funkcję, która pobiera znak z klawiatury (INT 21h, AH = 01h). Jeśli procedura jego obsługi jest aktywna, to MS-DOS jest jest w "bezpiecznym" stanie, co oznacza, że zawartości wszystkich stosów INT 21h są przewidywalne, a dostęp do dysku nie grozi katastrofą.
Przerwanie to jest często wykorzystywane przez wiele procesów drugoplanowych. Używane jest np. przez aplikację PRINT.COM. Przed jego przechwyceniem należy najpierw zapamiętać jego poprzedni wektor, a potem do tablicy IVT wpisać adres swej własnej procedury obsługi. Wywołanie tego przerwania powoduje więc jej uaktywnienie. Tak jak w innych wypadkach przechwycenia podprogramów ISR dobrze jest teraz wywołać oryginalną procedurę obsługi. Dopiero gdy zwróci ona sterowanie instrukcją IRET, można wykonać żądane zadanie. Należy pamiętać o przesłaniu na stos wartości rejestrów CS:IP oraz o przesłaniu znaczników przed wywołaniem oryginalnej procedury obsługi przerwania INT 28h.

Przerwanie 29h Szybka funkcja PUTCHAR (Wewnętrzna procedura DOS-u) [Wszystkie wersje]

Funkcje przerwania INT 21h, które obsługują wyjście, korzystają z niego w celu wysłania znaku do urządzenia, o ile słowo atrybutu jego sterownika ma ustawiony trzeci bit (04h). Przerwanie to nie powinno być wywoływane bezpośrednio.

Przerwanie 2Ah Sprawdzenie obecności sieci [Wersje 3 i wyższe]

Przerwanie to składa się z kilku funkcji, które służą do sprawdzenia obecności sieci, ustalenia jej nazwy, wywołania NETBIOS-u czy pobrania informacji o zasobach sieciowych. Zarezerwowane jest dla Microsoft Networks.

Przerwanie 2Bh-2Dh Nie udokumentowane

Zarezerwowane dla systemu MS-DOS.

Przerwanie 2Eh Wykonanie polecenia (Nie udokumentowana)

Rejestry ES:SI powinny zawierać adres łańcucha komendy zakończonego znakiem powrotu karetki. Komenda jest wykonywana przez najwcześniej zainstalowaną kopię programu COMMAND.COM. Operacja ta powoduje zniszczenie zawartości wszystkich rejestrów. Jednakże zachowanie tej funkcji nie jest do końca przewidywalne. Dobrze jest przed jej wywołaniem zapisać wartości wszystkich rejestrów, a przede wszystkim rejestrów SS i SP.

Przerwanie 2Fh Wywołanie wielu różnorakich funkcji [Wersje 3 i wyższe]

Udokumentowanym zastosowaniem tego przerwania jest dostarczenie funkcji do drukowania buforowanego. Ponadto zawiera ono wiele różnych procedur. Tak jak w innych tego typu przerwaniach wybór funkcji odbywa się poprzez podanie w rejestrze AH odpowiedniego jej numeru. Wartości 7Fh i mniejsze zarezerwowane są do obsługi zarządzania drukarką. Wartości 80h i większe mogą być wykorzystane przez programy użytkownika.
Przy instalowaniu nowych funkcji obowiązuje kilka zasad. Najpierw należy sprawdzić czy żądana wartość AH nie jest już zajęta przez inny podprogram. W tym celu wartość tę wpisuje się do AH, a AL ustawia się na zero i wywołuje przerwanie INT 2Fh. Zwrócona w AL wartość 0 oznacza, że numer jest wolny, 1, że jest niedostępny, a 0FFh, że procedura jest już zainstalowana. Na przykład, można wywołać przerwanie INT 2Fh z AH = 0C0h (najniższy numer dla funkcji użytkownika zalecany przez firmę Microsoft). Jeśli w AL zostanie zwrócona wartość zero, to pod numerem 0C0h można zainstalować własną funkcję. Oczywiście musi ona odpowiadać pewnym wymaganiom, na przykład, jeśli inny program wywoła INT 2Fh z AH =2Fh, to powinna ona zwrócić w AL wartość 0FFh.
W tablicy C-2 zostały zamieszczone kody błędów i nazwy.

Wejście: AH =< 7Fh zarezerwowane dla systemu DOS
AH => 80h identyfikator funkcji użytkownika
AL = kod funkcji
AX = 0100h Ustalenie stanu programu PRINT

Wyjście: AL = 0; PRINT nie jest zainstalowany; można go zainstalować
AL = 1; PRINT nie jest zainstalowany; nie można go jednak zainstalować
AL = FFh; PRINT jest zainstalowany;

Tabela C-2. Kody błędów przerwania INT 2Fh

Kod błędu	Nazwa błędu
1	Niepoprawna funkcja
2	Nie znaleziono pliku
3	Nie znaleziono ścieżki dostępu
4	Zbyt wiele otwartych plików
5	Zabroniony dostęp
8	Pełna kolejka
9	Zajętość
C	Zbyt długa nazwa
F	Błędnie określony dysk

AX = 0101h Przesłanie pliku do programu PRINT

Wejście: DS:DX = adres początku pakietu, który opisuje drukowany plik
1 BYTE poziom (0)
1 DWORD wskaźnik od łańcucha znaków ASCIIZ, który określa dysk, ścieżkę dostępu i nazwę pliku

Wyjście: Jeśli CF = 1, AX = kod błędu

AX = 0102h Usunięcie pliku z kolejki programu PRINT

Wejście: DS:DX = wskaźnik od łańcucha znaków ASCIIZ, który określa dysk, ścieżkę dostępu i nazwę pliku

Wyjście: Jeśli CF = 1, AX = kod błędu

AX = 0103h Usunięcie wszystkich plików z kolejki programu PRINT

Wejście: Nie ma

Wyjście: Jeśli CF = 1, AX = kod błędu

AX = 0104h Pobranie stanu programu PRINT

Wejście: Nie ma

Wyjście: DX = liczba błędów
DS:SI = wskaźnik do kolejki, która zawiera 64-bajtowe nazwy drukowanych plików; ostatni element zaczyna się znakiem pustym

AX = 0105h Zakończenie programu PRINT

Wejście: Nie ma

Wyjście: Jeśli CF = 1, AX = kod błędu

AX = 0200h Pobranie stanu zainstalowania programu ASSIGN

Wejście: Nie ma

Wyjście: AL = 0; ASSIGN nie jest zainstalowany; można go zainstalować
AL = 1; ASSIGN nie jest zainstalowany; nie można go jednak zainstalować
AL = FFh; ASSIGN jest zainstalowany;

AX = 1000h Pobranie stanu zainstalowania programu SHARE

Wejście: Nie ma

Wyjście: AL = 0; SHARE nie jest zainstalowany; można go zainstalować
AL = 1; SHARE nie jest zainstalowany; nie można go jednak zainstalować
AL = FFh; SHARE jest zainstalowany;

AX = B700h Pobranie stanu zainstalowania programu APPEND

Wejście: Nie ma

Wyjście: AL = 0; APPEND nie jest zainstalowany; można go zainstalować

AL = 1; APPEND nie jest zainstalowany; nie można go jednak zainstalować
AL = FFh; APPEND jest zainstalowany;

Przerwanie 30h Instrukcja FAR JUMP

Przerwanie współpracuje z przerwaniem INT 31h.

Przerwanie 31h

Przerwanie to jest drugą częścią przerwania INT 30h. Nie zawierają one adresów procedur ISR. Jest to już przestarzały (zgodność z CP/M) sposób na udostępnienie instrukcji dalekich skoków FAR JUMP.

Przerwanie 32h

Nie używane

Przerwanie 33h

Mysz firmy Microsoft

Przerwania 34h-3Eh

Wykorzystywane do emulacji operacji zmiennoprzecinkowych w językach Microsoft C i Turbo C.

Przerwanie 3Fh

Wykorzystywane przez zarządcę nakładek (LINK.EXE) firmy Microsoft.

Przerwanie 40h

Program obsługi stacji dysków (przeniesiony przez program twardego dysku, oryginalnie INT 13h). Nie jest to procedura obsługi przerwania.

Przerwanie 41h

Przerwanie to zawiera parametry dysków dla komputerów XT, AT, XT2, XT286, PS. Tak, jak w przerwaniu INT 40h, wektor tego przerwania wskazuje strukturę danych, która opisuje parametry twardego dysku.

16-bitowe słowo, które określa liczbę cylindrów
8-bitowy bajt, który określa liczbę głowic
16-bitowe słowo, które określa Write Pre-Comp
8-bitowy bajt, który zawiera wartość zero
8-bitowy bajt, który zawiera wartość zero
8-bitowy bajt, który zawiera wartość zero
8-bitowy bajt, który zawiera wartość zero
8-bitowy bajt, który zawiera wartość zero
16-bitowe słowo, które określa strefę parkowania głowicy
8-bitowy bajt, który określa liczbę sektorów na ścieżce
8-bitowy bajt, który zawiera wartość zero

Przerwanie 42h

Program obsługi kart EGA, VGA (oryginalnie INT 10h).

Przerwanie 43h

Parametry dla inicjacji kart EGA, VGA.

Przerwanie 44h

Kroje znaków dla kart EGA, VGA.

Przerwanie 45h Zarezerwowane

Przerwanie 46h

Parametry dla drugiego dysku stałego dla komputerów AT, XT286, PS.

Przerwanie 47h Zarezerwowane

Przerwanie 48h

Tablica kodów translacji klawiatury bezprzewodowej dla komputera PCjr.

Przerwanie 49h

Tablica kodów, które nie pochodzą z klawiatury dla komputera PCjr.

Przerwanie 4Ah

Alarm w komputerach AT, PS/2.

Przerwania 4Bh-4Fh Zarezerwowane

Przerwania 50h-57h

Żądanie przerwania 0-7 programu DesqView.

Przerwanie 58h Zarezerwowane

Przerwanie 59h

GSS Computer Graphics Interface.

Przerwanie 5Ah

Adres wejścia do programu BIOS, który obsługuje grupy dyskowe.

Przerwanie 5Bh Zarezerwowane

Przerwanie 5Ch Funkcje NETBIOS-u

Wejście: ES:BX = adres bloku NCB (Network Control Block)

Wyjście: AL = kod błędu

Przerwania 5Dh-5Fh Zarezerwowane

Przerwania 60h-66h

Przerwania przeznaczone dla użytkownika.

Przerwanie 67h Zarządca pamięci rozszerzonej (EMS)

Patrz rozdział 13.

Przerwania 68h-6Fh Nie używane

Przerwanie 70h

Zegar czasu rzeczywistego w komputerach AT, XT286, PS50, 60, 80.

Przerwanie 71h

Pierwsza karta LAN w komputerach AT, XT286, PS50, 60, 80.

Przerwanie 72h

W komputerach AT, XT286, PS50, 60, 80 zarezerwowane dla linii sprzętowych.

Przerwanie 73h

W komputerach AT, XT286, PS50, 60, 80 zarezerwowane dla linii sprzętowych.

Przerwanie 74h

Mysz w komputerach AT, XT286, PS50, 60, 80.

Przerwanie 75h

Błąd koprocesora w komputerach AT, XT286, PS50, 60, 80.

Przerwanie 76h

Sterownik dysku stałego w komputerach AT, XT286, PS50, 60, 80.

Przerwanie 77h

W komputerach AT, XT286, PS50, 60, 80 zarezerwowane dla linii sprzętowych.

Przerwania 78h-79h Nie używane

Przerwania 80h-85h

Używane przez interpreter języka BASIC.

Przerwanie 86h

Przekazanie sterowania do interpretera języka BASIC w pamięci ROM, przemieszczone przerwanie INT 18h.

Przerwania 87h-F0h

Używane przez interpreter języka BASIC.

Przerwania F1h-FDh

Przerwania dostępne dla programisty.

Przerwania FEh-FFh Nie używane

Dodatek D Konwersje znaków ASCII

Tabela D-1 zawiera konwersję znaków klawiatury na ich kod dziesiętny, szesnastkowy, ósemkowy i ASCII (American Standard Code for Information Interchange). Sekwencje klawiszy, które zawierają ^- uzyskuje się przez równoczesne wciśnięcie klawisza Control i klawisza podanego. Sekwencje te są oparte na definicjach, które obowiązują w większości standardowych terminali, na przykład klawiatura Diablo 1640 i seria terminali Televideo. Jednak definicje te mogą być różne na różnych klawiaturach.

Tabela D-1. Konwersja znaków ASCII

DZIESIĘTNIE	SZESNASTKOWO	ÓSEMKOWO	KOD ASCII	ZNAK TERMINALA*
0	0	0	NUL	^@
1	1	1	SOH	^A
2	2	2	STX	^B
3	3	3	ETX	^C
4	4	4	EOF	^D
5	5	5	ENQ	^E
6	6	6	ACK	^F
7	7	7	BEL	^G
8	8	10	BS	^H
9	9	11	HT	^I
10	A	12	LF	^J
11	B	13	VT	^K
12	C	14	FF	^L
13	D	15	CR	^M
14	E	16	SO	^N
15	F	17	SI	^O
16	10	20	DLE	^P
17	11	21	DC1	^Q
18	12	22	DC2	^R
19	13	23	DC3	^S
20	14	24	DC4	^T
21	15	25	NAK	^U
22	16	26	SYN	^V
23	17	27	ETB	^W
24	18	30	CAN	^X
25	19	31	EM	^Y
26	1A	32	SUB	^Z

DZIESIĘTNIE	SZESNASTKOWO	ÓSEMKOWO	KOD ASCII	ZNAK TERMINALA*
27	1B	33	ESC	^[
28	1C	34	FS	^\
29	1D	35	GS	^]
30	1E	36	RS	^
31	1F	37	US	^_
32	20	40	SP	Spacja
33	21	41	!	!
34	22	42	"	"
35	23	43	#	#
36	24	44	$	$
37	25	45	%	%
38	26	46	&	&
39	27	47	'	'
40	28	50	(	(
41	29	51	)	)
42	2A	52	*	*
43	2B	53	+	+
44	2C	54	,	,
45	2D	55	-	-
46	2E	56	.	.
47	2F	57	/	/
48	30	60	0	0
49	31	61	1	1
50	32	62	2	2
51	33	63	3	3
52	34	64	4	4
53	35	65	5	5
54	36	66	6	6
55	37	67	7	7
56	38	70	8	8
57	39	71	9	9
58	3A	72	:	:
59	3B	73	;	;
60	3C	74	<	<
61	3D	75	=	=
62	3E	76	>	>
63	3F	77	?	?
64	40	100	@	@

DZIESIĘTNIE	SZESNASTKOWO	ÓSEMKOWO	KOD ASCII	ZNAK TERMINALA*
65	41	101	A	A
66	42	102	B	B
67	43	103	C	C
68	44	104	D	D
69	45	105	E	E
70	46	106	F	F
71	47	107	G	G
72	48	110	H	H
73	49	111	I	I
74	4A	112	J	J
75	4B	113	K	K
76	4C	114	L	L
77	4D	115	M	M
78	4E	116	N	N
79	4F	117	O	O
80	50	120	P	P
81	51	121	Q	Q
82	52	122	R	R
83	53	123	S	S
84	54	124	T	T
85	55	125	U	U
86	56	126	V	V
87	57	127	W	W
88	58	130	X	X
89	59	131	Y	Y
90	5A	132	Z	Z
91	5B	133	[	[
92	5C	134	\	\
93	5D	135	]	]
94	5E	136	^	^
95	5F	137	_	_
96	60	140	`	`
97	61	141	a	a
98	62	142	b	b
99	63	143	c	c
100	64	144	d	d
101	65	145	e	e
102	66	146	f	f

DZIESIĘTNIE	SZESNASTKOWO	ÓSEMKOWO	KOD ASCII	ZNAK TERMINALA*
103	67	147	g	g
104	68	150	h	h
105	69	151	i	i
106	6A	152	j	j
107	6B	153	k	k
108	6C	154	l	l
109	6D	155	m	m
110	6E	156	n	n
111	6F	157	o	o
112	70	160	p	p
113	71	161	q	q
114	72	162	r	r
115	73	163	s	s
116	74	164	t	t
117	75	165	u	u
118	76	166	v	v
119	77	167	w	w
120	78	170	x	x
121	79	171	y	y
122	7A	172	z	z
123	7B	173	{	{
124	7C	174	|	|
125	7D	175	}	}
126	7E	176	~	~
127	7I	177	DEL	DEL

*Sekwencje klawiszy, zawierające ^, są uzyskiwane przez wciśnięcia klawisza Ctrl i jednoczesne naciśnięcie oznaczonego klawisza. Sekwencje te są oparte o definicje dla serii klawiatur IBM. Na innych klawiaturach mogą być one zdefiniowane inaczej. Znaki z rozszerzonego kodu ASCII mogą być uzyskane poprzez wciśnięcie klawisza <Alt> i wpisanie kodu znaku na klawiaturze numerycznej.

Skróty:
DEC = Dziesiętnie (podstawa 10)
HEX = Szesnastkowo (podstawa 16)
OCT = Ósemkowo (podstawa 8)
ASCII = American Standard Code for Information Interchange

Definicje niewidzialnych znaków ASCII

ACK (ACKNOWLEDGMENT) Znak sterujący, który w procesach komunikacji generalnie oznacza odpowiedź "tak" na wiele różnych zapytań. Czasami może również

oznaczać "Otrzymałem twoją ostatnią transmisję i jestem gotowy do przyjęcia następnej."

BELL (BELL) Znak sterujący ogólnego zastosowania, który powoduje uaktywnienie dzwonka, brzęczyka lub innego alarmu akustycznego na urządzeniu, na które został wysłany.

BS (BACKSPACE) Znak sterujący, którego działanie polega na przesunięciu karetki, głowicy drukarki lub kursora o jedno miejsce do tyłu.

CAN (CANCEL) Znak sterujący ogólnego zastosowania, który oznacza, że dane, otrzymane w poprzedniej transmisji, mają zostać zignorowane. O ich ilości decyduje użytkownik.

CR (CARRIAGE RETURN) Znak sterujący, który powoduje przemieszczenie karetki, głowicy drukarki lub kursora do początku linii. Na większości terminali klawisz Return powoduje wysłanie dwóch znaków CR i LF.

DC1-DC4 (DEVICE CONTROL) Znaki sterujące ogólnego zastosowania, które służą do współpracy z terminalami użytkownika lub podobnymi urządzeniami. Standardowo nie jest im przypisana żadna funkcja, z tym, że DC4 często oznacza stop. CCITT (Comite Consultatif International Telegraphe et Telephone [International Telegraph and Telephone Consultative Committee]) sugeruje pewną liczbę ich zastosowań. Ogólnie CCITT zaleca używanie dwóch pierwszych znaków do włączania (on), a pozostałych dwóch do wyłączania (off), przy czym DC2 i DC4 odnoszą się do ważniejszych urządzeń. W niektórych systemach znaki te nazywają się XON, TAPE, XOFF, NO TAPE.

DEL (DELETE) Znak sterujący, który powoduje skasowanie znaku. Na niektórych terminalach jest on zwany RUBOUT. Znak DEL nie jest ściśle znakiem sterującym, gdyż nie jest on zgrupowany razem ze znakami sterującymi ASCII. Jego kod składa się z samych jedynek (1111 1111, binarnie). Powód tego jest historyczny: Bowiem jedynym sposobem skasowania wydziurkowanego na papierze wzorca bitowego jest usunięcie wszystkich jego pozostałych dziurek. Staje się on wtedy ekwiwalentem znaku pustego. W kodzie ASCII znak DEL gra rolę właśnie takiego znaku, aczkolwiek w wielu systemach operacyjnych używany jest on do skasowania poprzedniego znaku.

DLE (DATA LINK ESCAPE) Znak sterujący używany do sterowania transmisją danych.

EM (END OF MEDIUM) Znak sterujący ogólnego zastosowania, który oznacza koniec taśmy papieru lub innego nośnika danych.

ENQ (ENQUIRY) Znak sterujący używany w komunikacji do wysłania żądania identyfikacji lub informacji o stanie. W niektórych systemach znak ten ma nazwę WRU (who are you?).

EOT (END OF TRANSMISSION) Znak sterujący, który oznacza koniec transmisji po wysłaniu jednej lub kilku wiadomości.

ESC (ESCAPE) Znak sterujący ogólnego zastosowania, który oznacza początek sekwencji rezygnacji. Sekwencja ta zawiera szereg kodów, które jako grupa mają określone znaczenie, zwykle jest to pewna funkcja sterująca. Na niektórych terminalach znak ten jest zwany ALT MODE.

ETB (END OF TRANSMISSION BLOCK) Znak sterujący przy dzieleniu długiego komunikatu na mniejsze bloki. Znak ETB określa ich koniec. Zwykle podział ten nie ma nic wspólnego z przesyłaną wiadomością.

EXT (END OF TEXT) Znak sterujący, który oznacza koniec tekstu. Patrz znak SOH. Znak ten zwany był dawniej EOM (End Of Message) i na niektórych terminalach może mieć taką właśnie etykietę.

FF (FORM FEED) Znak sterujący, który powoduje przemieszczenie się karetki, głowicy drukarki lub kursora do początku następnej strony.

FS, GS, RS, US (FILE, GROUP, RECORD AND UNIT SEPARATOR) Znaki, które sterują podziałem informacji na porcje. Nie istnieje żadne ich standardowe zastosowanie. Znak FS odnosi się do podziału największego, a US najmniejszego.

HT (HORIZONTAL TAB) Znak sterujący, który powoduje przemieszczenie karetki, głowicy drukarki lub kursora do następnej pozycji tabulacyjnej w tej samej linii. Pozycje te zwykle mogą być definiowane przez użytkownika.

LF (LINE FEED) Znak sterujący, który powoduje przemieszczenie karetki, głowicy drukarki lub kursora do następnej linii. W większości wypadków jest on łączony ze znakiem CR. Kombinacja taka jest zwana NL (New Line).

NAK (NEGATIVE ACKNOWLEDGMENT) Znak sterujący, który w procesach komunikacji oznacza odpowiedź "nie" na wiele różnych zapytań. Czasami może również oznaczać "Otrzymałem twoją ostatnią transmisję, ale pojawiły się w niej błędy i czekam na jej powtórzenie."

NUL (NULL) Znak sterujący ogólnego przeznaczenia, który głównie używany jest jako wypełniacz pustych miejsc. Patrz również znak SYN.

SI (SHIFT IN) Znak sterujący ogólnego przeznaczenia, który jest używany po znaku SO. Powoduje on przywrócenie normalnego znaczenia kodom występującym za nim.

SO (SHIFT OUT) Znak sterujący ogólnego przeznaczenia, który oznacza, że występujące za nim znaki mają inne znaczenie, niż w kodzie ASCII. Taki blok kończy pojawienie się znaku SI.

SOH (START OF HEADING) Znak sterujący, który oznacza początek nagłówka komunikatu. Zwykle zawiera on nazwę i położenie jego adresata. Pierwotnie był on nazywany SOM (Start Of Message).

STX (START OF TEXT) Znak sterujący, który używany jest jako znacznik początku tekstu i końca nagłówka (jeśli wystąpił). Pierwotnie był on nazywany EOA (End Of Address).

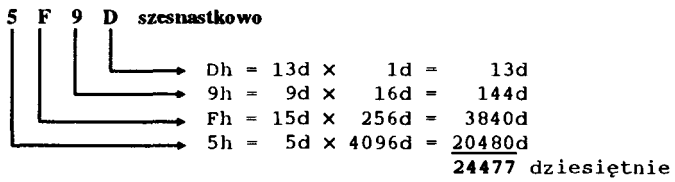
SUB (SUBSTITUTE) Znak sterujący ogólnego przeznaczenia oznaczający znak, który ma zastąpić kod, o którym wiadomo, że jest błędny.

SYN (SYNCHRONOUS IDLE) Znak sterujący używany w systemach komunikacyjnych o dużej szybkości transmisji do zsynchronizowania zegarów nadajnika i odbiornika. Gdy jest bezczynny nie są przesyłane żadne znaki, to zegar odbiornika może bowiem ulec rozsynchronizowaniu z zegarem nadajnika. Tak więc każda transmisja po okresie "ciszy" jest poprzedzana trzema, czterema znakami SYN. Bity tego znaku są tak dobrane, że pozwalają nie tylko zgrać się obu zegarom, ale także ustalić początek i koniec znaków. Znaki te mogą być również przesyłane w okresie bezczynności, zapewniając ciągłą synchronizację, stąd z resztą nazwa.

VT (VERTICAL TAB) Znak sterujący, który powoduje przemieszczenie karetki, głowicy drukarki lub kursora do następnej pionowej pozycji tabulacyjnej. Zwykle jest to jedna linia.

Zamiana liczb szesnastkowych na dziesiętne

Rysunek D-1 przedstawia sposób zamiany szesnastkowej liczby 5F9D na jej dziesiętny odpowiednik.

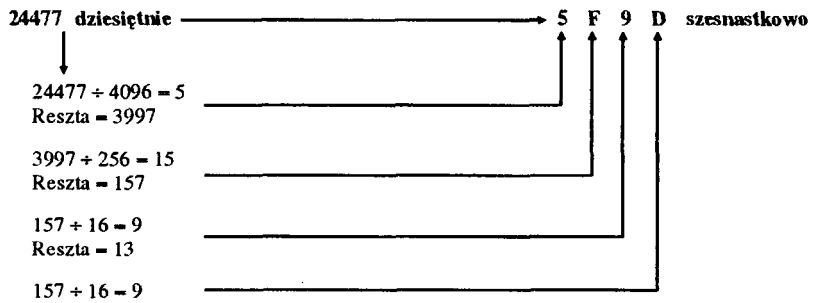


Rysunek D-1. ***Zamiana szesnastkowej liczby 5F9D na liczbę dziesiętną.***

Każda cyfra szesnastkowa jest 16 razy większa niż cyfra występująca po jej prawej stronie.

Zamiana liczb dziesiętnych na szesnastkowe

Proces ten jest odwróceniem procesu konwersji liczb szesnastkowych na dziesiętne. Najpierw wybiera się cyfrę wysuniętą najbardziej w lewo i wyznacza się jej pozycję w liczbie (pozycja setek, tysięcy, itd.). Następnie jest ona dzielona przez liczbę szesnastkową, odpowiadającą tej pozycji. Jeśli, na przykład, pierwsza cyfra znajduje się na pozycji tysięcy, to całą liczbę należy podzielić przez 4,096 (dziesiętny odpowiednik szesnastkowej liczby 1,000). Rezultat tej operacji jest pierwszą cyfrą szesnastkową. Reszta zaś dzielona jest przez heksadecymalną wartość, która odpowiada następnej pozycji (to znaczy, należy podzielić przez wartość 256, gdyż jest ona dziesiętnym odpowiednikiem szesnastkowej liczby 100). Rysunek D-2 przedstawia sposób zamiany dziesiętnej liczby otrzymanej w poprzednim przykładzie na jej szesnastkowy odpowiednik.



Rysunek D-2. ***Zamiana dziesiętnej liczby 24477 na liczbę szesnastkową.***

Instrukcje procesora i dyrektywy Turbo Assemblera
Uporządkowane alfabetycznie

UWAGI: (286) lub (386) oznaczają, że instrukcje te są osiągalne tylko w procesorach 80286 lub 80386 i wyższych. Pozostałe instrukcje osiągalne są na wszystkich procesorach 80x86

Instrukcje procesora

AAA, 135
AAD, 136
AAM, 137
AAS, 138
ADC, 139
ADD, 141
AND, 143
ARPL (286), 257
BOUND (286), 213
BSF (386), 144
BSR (386), 144
BT (386), 146
BTC (386), 146
BTR (386), 146
BTS (386), 146
CALL, 215
CBW, 149
CDQ (386), 150
CLC, 258
CLD, 259
CLI, 260
CLTS (286), 261
CMC, 262
CMP, 151
CMPS, 153
CWD, 154
CWDE (386), 155
DAA, 156
DAS, 157
DEC, 159
DIV, 160
ENTER (286), 217
ESC, 264
HLT, 265
IDIV, 162
IMUL, 163
IN, 71
INC, 166
INS (286), 72
INT, 219
INTO, 221
IRET, 223
JA lub JNBE, 226
JAE lub JNB, 226
JB lub JNAE, 226
JBE lub JNA, 226
JC, 226
JCXZ lub JECXZ, 227
JE lub JZ, 226
JG lub JNLE, 226
JGE lub JNL, 226
JL lub JNGE, 226
JLE lub JNZ, 226
JMP, 228
JNC, 226
JNE lub JNG, 226
JNO, 226
JNP lub JPO, 226
JNS, 226
JO, 226
JP lub JPE, 226
JS, 226
LAHF, 267
LAR (286), 268
LDS, 73
LEA, 75
LEAVE (286), 230
LES, 74
LFS (386), 75
LGDT (286), 269
LGS (386), 75
LIDT (286), 269
LLDT (286), 269
LMSW (286), 271
LOCK, 272
LODS, 77
LOOP, 231
LOOPE lub LOOPZ, 231
LOOPNE lub LOOPNZ, 231
LSL, 273
LSS (386), 75
LTR, 275
MOV, 78
MOVS, 81
MOVSX (386), 83
MOVZX (386), 85
MUL, 167
NEG, 168
NOP, 237
NOT, 170
OR, 171
OUT, 86
OUTS (286), 87
POP, 89
POPA, 90
POPF, 91
PUSH, 93
PUSHA, 94
PUSHF, 95
RCL, 172
RCR, 174
REP, 96
REPE lub REPZ, 97
REPNE lub REPNZ, 97
RET, 233
RETF, 233
RETN, 233
ROL, 175
ROR, 177
SAHF, 279
SAL lub SHL, 179
SAR, 181
SBB, 187
SCAS, 190
SETA lub SETNBE (386), 98
SETAE lub SETNB (386), 97
SETB lub SETNAE (386), 98
SETBE lub SETNA (386), 98
SETC (386), 98
SETE lub SETZ (386), 98
SETG lub SETNLE (386), 98
SETGE lub SETNL (386), 98
SETL lub SETNGE (386), 98
SETLE lub SETNG (386), 98
SETNC (386), 98
SETNE lub SETNZ (386), 98
SETNO (386), 98
SETNP lub SETPO (386), 98
SETNS (386), 98
SETO (386), 98
SETP lub SETPE (386), 98
SETS (386), 98
SGDT (286), 280
SHLD (386), 183
SHR, 184
SHRD (386), 186
SIDT (286), 280
SLDT (286), 280
SMSW (286), 281
STC, 258
STD, 282
STI, 283
STOS, 100
STR (286), 283
SUB, 191
TEST, 193
VERR (286), 284
VERW (286), 284
WAIT, 286
XCHG, 101
XLAT lub XLATB, 102
XOR, 195

Dyrektywy TASM

.186, 442
.286, 442
.286P, 443
.287, 444
.386, 445
.386P, 446
.387, 447
.8086, 447
.8087, 448
.ALPHA, 301
.CODE, 304
.CREF, 451
.DATA?, 306
.DATA, 306
.ERR, 457
.ERR1, 457
.ERR2, 457
.ERRB, 457
.ERRDEF, 457
.ERRDIF lub .ERRDIFI, 457
.ERRE, 457
.ERRIDN lub .ERRIDNI, 457
.ERRNB, 457
.ERRNDEF, 457
.ERRNZ, 457
.FARDATA?, 309
.FARDATA, 309
.LALL, 461
.LFCOND, 462
.LIST, 463
.MODEL, 311
.RADIX, 362
.SALL, 479
.SEQ, 301
.SFCOND, 479
.STACK, 313
.TFCOND, 482
.XALL, 486
.XCREF, 486
.XLIST, 486
%BIN, 449
%CONDS, 450
%CREF, 451
%CREFALL, 452
%CREFUREF, 452
%CTLS, 453
%DEPTH, 455
%LINUM, 462
%LIST, 463
%MACS, 464
%NEWPAGE, 467
%NOCONDS, 468
%NOCREF, 468
%NOCTLS, 468
%NOINCL, 468
%NOLIST, 469
%NOMACS, 469
%NOSYMS, 469
%NOTRUNC, 469
%OUT, 470
%PAGESIZE, 475
%PCNT, 476
%POPLCTL, 477
%PUSHLCTL, 478
%SUBTTL, 479
%SYMS, 480
%TABSIZE, 481
%TEXT, 481
%TITLE, 483
%TRUNC, 484
ALIGN, 334
ARG, 401
ASSUME, 302
CATSTR, 336
CODESEG, 304
COMM, 402
COMMENT, 435
DATASEG, 306
DB, 337
DD, 337
DF, 338
DOSSEG, 301
DQ, 338
DT, 339
DW, 340
ELSE, 404
ELSEIF, 405
ELSEIF1, 405
ELSEIF2, 405
ELSEIFB, 405
ELSEIFDEF, 405
ELSEIFDIF, 405
ELSEIFDIFI, 405
ELSEIFE, 405
ELSEIFIDN, 405
ELSEIFIDNI, 405
ELSEIFNB, 405
ELSEIFNDEF, 405
EMUL, 455
END, 307
ENDIF, 406
ENDM, 407
ENDP, 408
ENDS, 308
ENDS, 308
EQU, 373
ERR, 457
ERRIF, 457
ERRIF1, 457
ERRIF2, 457
ERRIFB, 457
ERRIFDEF, 457
ERRIFDIF, 457
ERRIFDIFI, 457
ERRIFE, 457
ERRIFIDN, 457
ERRIFIDNI, 457
ERRIFNB, 457
ERRIFNDEF, 457
EVEN, 342
EVENDATA, 342
EXITM, 409
EXTRN, 411
GLOBAL, 413
GROUP, 310
IDEAL, 458
IF, 415
IF1, 416
IF2, 416
IFB, 416
IFDEF, 416
IFDIF lub IFDIFI, 416
IFE, 416
IFIDN lub IFIDNI, 416
IFNB, 416
IFNDEF, 416
INCLUDE, 418
INCLUDELIB, 418
INSTR, 348
IRP, 419
IRPC, 420
JUMPS, 460
LABEL, 349
LOCAL, 421
MACRO, 423
MASM, 464
MASM51, 465
MULTERRS, 466
NAME, 466
NOEMUL, 468
NOJUMPS, 468
NOLOCALS, 422
NOMASM51, 469
NOWARN, 470
ORG, 358
P186, 471
P286, 471
P286N, 471
P287, 471
P386, 471
P386N, 471
P387, 472
P8086, 472
P8087, 472
PAGE, 474
PNO87, 472
PROC, 358
PUBLIC, 426
PURGE, 426
RECORD, 363
REPT, 427
SEGMENT, 312
SIZESTR, 368
STACK, 313
STRUC, 370
SUBSTR, 372
SUBTTL, 479
TITLE, 483
UDATASEG, 314
UFARDATA, 314
WARN, 485

Instrukcje procesora i dyrektywy Turbo Assemblera Uporządkowane tematycznie

UWAGI: (286) lub (386) oznaczają, że instrukcje te są osiągalne tylko w procesorach 80286 lub 80386 i wyższych. Pozostałe instrukcje osiągalne są na wszystkich procesorach 80x86

Instrukcje procesora

Rozkazy transmisji danych

IN, 71
INS (286), 72
LDS, 73
LEA, 75
LES, 74
LFS (386), 75
LGS (386), 75
LODS, 77
LSS (386), 75
MOV, 78
MOVS, 81
MOVSX (386), 83
MOVZX (386), 85
OUT, 86
OUTS (286), 87
POP, 89
POPA, 90
POPF, 91
PUSH, 93
PUSHA, 94
PUSHF, 95
REP, 96
REPE lub REPZ, 97
REPNE lub REPNZ, 97
STOS, 100
XCHG, 101
XLAT lub XLATB, 102

Instrukcje arytmetyczne i logiczne

AAA, 135
AAD, 136
AAM, 137
AAS, 138
ADC, 139
ADD, 141
AND, 143
BSF (386), 144
BSR (386), 144
BT (386), 146
BTC (386), 146
BTR (386), 146
BTS (386), 146
CBW, 149
CDQ (386), 150
CMP, 151
CMPS, 153
CWD, 154
CWDE (386), 155
DAA, 156
DAS, 157
DEC, 159
DIV, 160
IDIV, 162
IMUL, 163
INC, 166
MUL, 167
NEG, 168
NOT, 170
OR, 171
RCL, 172
RCR, 174
ROL, 175
ROR, 177
SAL lub SHL, 179
SAR, 181
SBB, 187
SCAS, 190
SETA lub SETNBE (386), 98
SETAE lub SETNB (386), 97
SETB lub SETNAE (386), 98
SETBE lub SETNA (386), 98
SETC (386), 98
SETE lub SETZ (386), 98
SETG lub SETNLE (386), 98
SETGE lub SETNL (386), 98
SETL lub SETNGE (386), 98
SETLE lub SETNG (386), 98
SETNC (386), 98
SETNE lub SETNZ (386), 98
SETNO (386), 98
SETNP lub SETPO (386), 98
SETNS (386), 98
SETO (386), 98
SETP lub SETPE (386), 98
SETS (386), 98
SHLD (386), 183
SHR, 184
SHRD (386), 186
SUB, 191
TEST, 193
XOR, 195

Procedury, pętle i skoki

BOUND (286), 213
CALL, 215
ENTER (286), 217
INT, 219
INTO, 221
IRET, 223
JA lub JNBE, 226
JAE lub JNB, 226
JB lub JNAE, 226
JBE lub JNA, 226
JC, 226
JCXZ lub JECXZ, 227
JE lub JZ, 226
JG lub JNLE, 226
JGE lub JNL, 226
JL lub JNGE, 226
JLE lub JNZ, 226
JMP, 228
JNC, 226
JNE lub JNG, 226
JNO, 226
JNP lub JPO, 226
JNS, 226
JO, 226
JP lub JPE, 226
JS, 226
LEAVE (286), 230
LOOP, 231
LOOPE lub LOOPZ, 231
LOOPNE lub LOOPNZ, 231
RET, 233
RETF, 233
RETN, 233

Sterowanie procesorem i operacje w trybie chronionym

ARPL (286), 257
CLC, 258
CLD, 259
CLI, 260
CLTS (286), 261
CMC, 262
ESC, 264
HLT, 265
LAHF, 267
LAR (286), 268
LGDT (286), 269
LIDT (286), 269
LLDT (286), 269
LMSW (286), 271
LOCK, 272
LSL, 273
LTR, 275

NOP, 237
SAHF, 279
SGDT (286), 280
SIDT (286), 280
SLDT (286), 280
SMSW (286), 281
STC, 258
STD, 282
STI, 283
STR (286), 283
VERR (286), 284
VERW (286), 284
WAIT, 286

Dyrektywy TASM

Deklarowanie segmentów

.ALPHA, 301
.CODE, 304
.DATA, 306
.DATA?, 306
.FARDATA, 309
.FARDATA?, 309
.MODEL, 311
.SEQ, 301
.STACK, 313
ASSUME, 302
CODESEG, 304
DATASEG, 306
DOSSEG, 301
END, 307
ENDS, 308
GROUP, 310
SEGMENT, 312
STACK, 313
UDATASEG, 314
UFARDATA, 314

Definicje zmiennych i ich rozmieszczenie w pamięci

.RADIX, 362
ALIGN, 334
CATSTR, 336
DB, 337
DD, 337
DF, 338
DQ, 338
DT, 339
DW, 340
ENDS, 308
EQU, 373
EVEN, 342
EVENDATA, 342
INSTR, 348
LABEL, 349
ORG, 358
RECORD, 363
SIZESTR, 368
STRUC, 370
SUBSTR, 372

Makrodefinicje, procedury i programowanie modułowe

ARG, 401
COMM, 402
ELSE, 404
ELSEIF, 405
ELSEIF1, 405
ELSEIF2, 405
ELSEIFB, 405
ELSEIFDEF, 405
ELSEIFDIF, 405
ELSEIFDIFI, 405
ELSEIFE, 405
ELSEIFIDN, 405
ELSEIFIDNI, 405
ELSEIFNB, 405
ELSEIFNDEF, 405
ENDIF, 406
ENDM, 407
ENDP, 408
EXITM, 409
EXTRN, 411
GLOBAL, 413
IF, 415
IF1, 416
IF2, 416
IFB, 416
IFDEF, 416
IFDIF lub IFDIFI, 416
IFE, 416
IFIDN lub IFIDNI, 416
IFNB, 416
IFNDEF, 416
INCLUDE, 418
INCLUDELIB, 418
IRP, 419
IRPC, 420
LOCAL, 421
MACRO, 423
PROC, 358
PUBLIC, 426
PURGE, 426
REPT, 427

Obsługa błędów i tworzenie raportów

%BIN, 449
%CONDS, 450
%CREF, 451
%CREFALL, 452
%CREFUREF, 452
%CTLS, 453
%DEPTH, 455
%LINUM, 462
%LIST, 463
%MACS, 464
%NEWPAGE, 467
%NOCONDS, 468
%NOCREF, 468
%NOCTLS, 468
%NOINCL, 468
%NOLIST, 469
%NOMACS, 469
%NOSYMS, 469
%NOTRUNC, 469
%OUT, 470
%PAGESIZE, 475
%PCNT, 476
%POPLCTL, 477
%PUSHLCTL, 478
%SUBTTL, 479
%SYMS, 480
%TABSIZE, 481
%TEXT, 481
%TITLE, 483
%TRUNC, 484
.186, 442
.286, 442
.286P, 443
.287, 444
.386, 445
.386P, 446
.387, 447
.8086, 447
.8087, 448
.CREF, 451
.ERR, 457
.ERR1, 457
.ERR2, 457
.ERRB, 457
.ERRDEF, 457
.ERRDIF lub .ERRDIFI, 457
.ERRE, 457
.ERRIDN lub .ERRIDNI, 457
.ERRNB, 457
.ERRNDEF, 457
.ERRNZ, 457
.LALL, 461
.LFCOND, 462
.LIST, 463
.SALL, 479
.SFCOND, 479
.TFCOND, 482
.XALL, 486
.XCREF, 486
.XLIST, 486
COMMENT, 435
EMUL, 455
ERR, 457
ERRIF, 457
ERRIF1, 457
ERRIF2, 457
ERRIFB, 457
ERRIFDEF, 457
ERRIFDIF, 457
ERRIFDIFI, 457
ERRIFE, 457
ERRIFIDN, 457
ERRIFIDNI, 457
ERRIFNB, 457

ERRIFNDEF, 457
IDEAL, 458
JUMPS, 460
MASM, 464
MASM51, 465
MULTERRS, 466
NAME, 466
NOEMUL, 468
NOJUMPS, 468
NOLOCALS, 422
NOMASM51, 469
NOWARN, 470
P186, 471
P286, 471
P286N, 471
P287, 471
P386, 471
P386N, 471
P387, 472
P8086, 472
P8087, 472
PAGE, 474
PNO87, 472
SUBTTL, 479
TITLE, 483
WARN, 485

Indeks

Symbole

%BIN dyrektywa, 449
%CONDS dyrektywa, 450
%CREF dyrektywa, 451
%CREFALL dyrektywa, 452
%CREFREF dyrektywa, 452
%CREFUREF dyrektywa, 452
%CTLS dyrektywa, 453
%DEPTH dyrektywa, 455
%INCL dyrektywa, 459
%LINUM dyrektywa, 462
%LIST dyrektywa, 463
%MACS dyrektywa, 464
%NEWPAGE dyrektywa, 467
%NOCONDS dyrektywa, 468
%NOCREF dyrektywa, 468
%NOCTLS dyrektywa, 468
%NOINCL dyrektywa, 468
%NOLIST dyrektywa, 469
%NOMACS dyrektywa, 469
%NOSYMS dyrektywa, 469
%NOTRUNC dyrektywa, 469
%OUT dyrektywa, 470
%PAGESIZE dyrektywa, 475
%PCNT dyrektywa, 476
%POPLCTL dyrektywa, 477
%PUSHLCTL dyrektywa, 478
%SUBTTL dyrektywa, 479
%SYMS dyrektywa, 480
%TABSIZE dyrektywa, 481
%TEXT dyrektywa, 481
%TITLE dyrektywa, 483
%TRUNC dyrektywa, 484
.186 dyrektywa, 442
.286 dyrektywa, 442
.286C dyrektywa, 442
.286P dyrektywa, 443
.287 dyrektywa, 444
.386 dyrektywa, 445
.386C dyrektywa, 445
.386P dyrektywa, 446
.387 dyrektywa, 447
.8086 dyrektywa, 447
.8087 dyrektywa, 448
.ALPHA dyrektywa, 301
.CODE dyrektywa, 304
.CREF dyrektywa, 451
.DATA dyrektywa, 306
.DATA? dyrektywa, 306
.ERR dyrektywa, 457
.ERR1 dyrektywa, 457
.ERR2 dyrektywa, 457
.ERRB dyrektywa, 457
.ERRDEF dyrektywa, 457
.ERRDIF dyrektywa, 457
.ERRDIFI dyrektywa, 457
.ERRE dyrektywa, 457
.ERRIDN dyrektywa, 457
.ERRIDNI dyrektywa, 457
.ERRNB dyrektywa, 457
.ERRNDEF dyrektywa, 457
.ERRNZ dyrektywa, 457
.FARDATA dyrektywa, 309
.FARDATA? dyrektywa, 309
.LALL dyrektywa, 461
.LFCOND dyrektywa, 462
.LIST dyrektywa, 463
.MODEL dyrektywa, 311
.RADIX dyrektywa, 362
.SALL dyrektywa, 479
.SEQ dyrektywa, 301, 313
.SFCOND dyrektywa, 479
.STACK dyrektywa, 313
.TFCOND dyrektywa, 482
.TYPE dyrektywa, 374
.XALL dyrektywa, 486
.XCREF dyrektywa, 486
.XLIST dyrektywa, 486
00h przerwanie, dzielenie przez zero, 610
01h przerwanie, praca krokowa, 610
02h przerwanie niemaskowalne , 610
03h przerwanie, punkt kontrolny, 610
04h przerwanie, nadmiar, 611
05h przerwanie, wydruk zawartości ekranu, 611
06h przerwanie, zarezerwowane, 611
07h przerwanie, zarezerwowane, 611
08h przerwanie, zegar czasu rzeczywistego, 611
09h przerwanie, klawiatura, 611
0Ah przerwanie, 611
0Bh przerwanie, 611
0Ch przerwanie, 611
0Dh przerwanie, 612
0Eh przerwanie, sterownik dysków, 612
0Fh przerwanie, układ wyjściowy LPT1, 612
10h przerwanie, obsługa ekranu, 612
    00h wybór trybu pracy monitora, 612
    01h definiowanie rozmiaru kursora, 613
    02h ustawienie pozycji kursora, 614
    03h odczytanie pozycji kursora, 614
    04h odczytanie pozycji pióra świetlnego, 614
    05h wybranie strony, 614
    06h przewijanie strony w górę, 615
    07h przewijanie strony w dół, 615
    08h odczytanie znaku i atrybutu, 615

09h zapis znaku i atrybutu, 615
0Ah zapis znaku, 616
0Bh ustawienie palety kolorów, 616
0Ch wyświetlenie punktu, 616
0Dh odczytanie koloru punktu, 616
0Eh zapis znaku w trybie dalekopisowym, 617
0Fh odczyt aktualnego trybu, 617
10h ustawienie rejestru palety kolorów, 617
11h przerwanie, test konfiguracji sprzętu, 618
12h przerwanie, odczyt rozmiaru pamięci, 618
13h przerwanie, obsługa dysków, 618
00h inicjacja systemu dyskowego, 619
01H odczyt statusu dysku, 619
02h odczyt sektorów, 619
03h zapis sektorów, 620
04h weryfikacja sektorów, 620
05h formatowanie ścieżki, 620
14h przerwanie, obsługa portu szeregowego, 620
00h inicjacja portu szeregowego, 621
01h wysłanie znaku, 622
02h odczyt znaku, 622
03h pobranie statusu portu szeregowego, 622
15h przerwanie, obsługa magnetofonu, 622
16h przerwanie, klawiatura, 622
00h odczytanie znaku, 622
01h pobranie stanu bufora klawiatury, 622
02h odczyt stanu klawiszy sterujących, 623
17h przerwanie, obsługa drukarki, 623
00h wysłanie znaku, 623
01h inicjacja drukarki, 624
02h odczyt stanu drukarki, 624
.186 dyrektywa, 442
18h przerwanie, wywołanie BASIC, 624
19h przerwanie, 'bootstrap', 624
1Ah przerwanie, obsługa zegara, 624
00h odczytanie czasu, 624
01h ustawienie czasu, 624
1Bh przerwanie, naciśnięcie Ctrl-Break, 625
1Ch przerwanie zegarowe, 625
1Dh przerwanie, tablica parametrów video, 626
1Eh przerwanie, tablica parametrów dyskowych, 626
1Fh przerwanie, tablica wzorów znaków, 626
20h przerwanie, zakończenie programu, 627
21h przerwanie, funkcje systemowe, 627
00h zakończenie programu, 627
01h odczyt znaku z echem, 628
02h wysłanie znaku na konsolę, 628
03h odczyt znaku z portu dodatkowego, 628
04h wysłanie znaku do portu dodatkowego, 629
05h wysłanie znaku do drukarki, 629
06h bezpośrednia komunikacja z konsolą, 629
07h bezp. odczytanie znaku bez echa, 630
08h odczytanie znaku bez echa, 630
09h wyślanie na konsolę ciągu znaków, 630
0Ah buforowane wejście z echem, 631
0Bh sprawdzenie stanu standardowego wejścia, 631
0Ch wyczyszczenie bufora klawiatury, 631
0Dh opróżnienie buforów dyskowych, 632
0Eh wybór dysku, 632
0Fh otwarcie pliku, 632
10h zamknięcie pliku, 633
11h poszukiwanie pierwszego pliku, 633
12h poszukiwanie następnego pliku, 634
13h usunięcie pliku, 634
14h odczyt sekwencyjny, 634
15h zapis sekwencyjny, 635
16h utworzenie pliku, 635
17h zmiana nazwy pliku, 636
18h nie udokumentowana, 636
19h pobranie bieżącego dysku, 636
1Ah ustawienie bufora operacji dyskowych, 636
1Bh informacja o dysku bieżącym, 637
1Ch informacja o wskazanym dysku, 637
1Dh nie udokumentowana, 637
1Eh nie udokumentowana, 637
1Fh pobranie bloku parametrów dysku, 637
21h odczyt swobodny, 638
22h zapis swobodny, 638
23h pobranie rozmiaru pliku, 638
24h określenie rekordu, 639
25h ustawienie wektora przerwań, 639
26h utworzenie nowego przedrostka programu, 639
27h swobodny odczyt bloku rekordów, 639
28h swobodny zapis bloku rekordów, 640
29h analiza nazwy zbioru, 640
2Ah pobranie daty, 641
2Bh ustawienie daty, 641
2Ch pobranie czasu, 641
2Dh ustawienie czasu, 642
2Eh ustawienie/zerowanie znacz. weryf., 642
2Fh pobranie adresu obszaru op. dyskowych, 642
30h pobranie numeru wersji systemu, 642
31h zakończenie programu bez usuwania go, 643
32h odnalezienie bloku parametrów dysku, 643
33h reakcja na Control-C, 644
34h zwrócenie znacznika sekcji krytycznej, 644
35h pobranie wektora przerwania, 644
36h ilość wolengo miejsca na dysku, 645
37h miana znaku opcji, 645
38h pobranie informacji o kraju, 645
39h stworzenie podkatalogu, 646
3Ah usunięcie podkatalogu, 646
3Bh zmiana bieżącego katalogu, 646
3Ch utworzenie pliku, 646
3Dh otwarcie pliku, 647
3Eh zamknięcie pliku, 648
3Fh odczyt z pliku, 648
40h zapis do pliku, 648
41h usunięcie pliku, 649

42h przesunięcie wskaźnika pliku, 649
43h zmiana atrybutów pliku, 649
44h sterowanie urządzeniami we/wy, 650
45h duplikacja uchwytu, 651
46h wymuszona zmiana uchwytu pliku, 651
47h pobranie nazwy bieżącego katalogu, 651
48h przydzielenie bloku pamięci, 652
49h zwolnienie bloku pamięci, 652
4Ah zmiamna rozmiaru bloku pamięci, 652
4Bh załadowanie i uruchomienie programu, 653
4Ch zakończenie procesu, 654
4Dh pobranie kodu procesu potomnego, 654
4Eh poszukiwanie pierwszego pliku, 654
4Fh poszukiwanie następnego pliku, 655
50h ustawienie bloku PSP, 655
51h pobranie adresu bieżącego bloku PSP, 655
52h pobranie adresu listy, 655
53h nie udokumentowana, 656
54h pobranie znacznika weryfikacji, 656
55h utworzenie bloku PSP, 656
56h zmiana nazwy pliku, 656
57h pobranie daty i czasu mod. pliku, 656
58h strategia przydziału pamięci, 657
59h pobranie rozszerzonego kodu błędu, 657
5Ah utworzenie pliku tymczasowego, 658
5Bh utworzenie nowego pliku, 658
5Ch blokada dostępu do pliku, 659
5Dh nie udokumentowana, 659
5E00h sieć, 659
5E02h sieć, 659
5E03h sieć, 659
5F02h sieć, 660
5F03h sieć, 660
60h nie udokumentowana, 660
62h pobranie adresu bloku PSP, 660
63h pobranie adresu tablicy DBCS, 661
64h nie udokumentowana, 661
65h rozszerzona informacja o kraju, 661
66h globalna tablica kodów, 661
67h ustawienie liczby otwartych plików, 661
68h opróżnienie buforów pliku, 661
69h - 6Bh nie udokumentowane, 662
6Ch rozszerzone otwarcie pliku, 662
22h przerwanie, adres skoku, 663
23h przerwanie, adres skoku po Ctrl-C, 663
24h przerwanie, adres obsługi błedu kryt., 663
25h przerwanie, fizyczny odczyt sektorów, 664
26h przerwanie, fizyczny zapis sektorów, 665
27h przerwanie, zakończenie programu, 666
.286 dyrektywa, 442
.286C dyrektywa, 442
.286P dyrektywa, 443
.287 dyrektywa, 444
28h przerwanie, pętla oczekiwania na klaw., 666
29h przerwanie, szybka funkcja PUTCHAR, 666
2Ah przerwanie, sprawdzenie obecności sieci, 666
2Bh-2Dh przerwanie, nie udokumentowane, 666
2Eh przerwanie, wykonanie polecenia, 667
2Fh przerwanie, różne funkcje, 667-668
0101h przesłanie pliku do PRINT, 668
0102h usunięcie pliku z kolejki PRINT, 668
0103h usunięcie wszystkich plików z k. PRINT, 668
0104h pobranie stanu programu PRINT, 668
0105h zakończenie programu PRINT, 668
0200h pobranie stanu programu ASSIGN, 668
1000h pobranie stanu programu SHARE, 668
B700h pobranie stanu programu APPEND, 668
30h przerwanie, 669
31h przerwanie, 669
32h przerwanie, nie używana, 669
33h przerwanie, mysz, 669
34h-3Eh przerwania, 669
.386 dyrektywa, 445
.386C dyrektywa, 445
.386P dyrektywa, 446
.387 dyrektywa, 447
3Fh przerwanie, 669
40h przerwanie, 669
41h przerwanie, 669
42h przerwanie, 669
43h przerwanie, 670
44h przerwanie, 670
45h przerwanie, zarezerwowane, 670
46h przerwanie, 670
47h przerwanie, zarezerwowane, 670
48h przerwanie, 670
49h przerwanie, 670
4Ah przerwanie, 670
4Bh-4Fh przerwania, zarezerwowane, 670
50h-57h przerwania, 670
58h przerwanie, zarezerwowane, 670
59h przerwanie, 670
5Ah przerwanie, 670
5Bh przerwanie, zarezerwowane, 670
5Ch przerwanie, funkcje NETBIOS-u, 670
5Dh-5Fh przerwania, zarezerwowane, 670
60h-66h przerwania, 670
67h przerwanie, zarządca pamięci EMS, 670
68h-6Fh przerwania, 671
70h przerwanie, 671
71h przerwanie, 671
72h przerwanie, 671
73h przerwanie, 671
74h przerwanie, 671
75h przerwanie, 671
76h przerwanie, 671
77h przerwanie, 671
78h-79h przerwania, nie używane, 671
.8086 dyrektywa, 447
.8087 dyrektywa, 448

80h-85h przerwania, 671
8250 układ szeregowy, 547
8259 układ, 539-541
86h przerwanie, 671
87h-F0h przerwania, 671
??FileName dyrektywa, 413
@Code dyrektywa, 302
@CodeSize dyrektywa, 303
@CurSeg dyrektywa, 305
@Data dyrektywa, 306
@DataSize dyrektywa, 306
@FarData dyrektywa, 308
@FarData? dyrektywa, 308
@FileName dyrektywa, 413

A

AAA instrukcja, 135
AAD instrukcja, 136
AAM instrukcja, 137
AAS instrukcja, 138
ADC instrukcja, 139-140
ADD instrukcja, 141-142
adres efektywny, 57
ALIGN dyrektywa, 334
.ALPHA dyrektywa, 301
AND dyrektywa, 333
AND instrukcja, 143
ARG dyrektywa, 401
ARPL instrukcja, 257
ASSUME dyrektywa, 302

B

%BIN dyrektywa, 449
BOUND instrukcja, 213-214
BSF instrukcja, 144-145
BSR instrukcja, 144-145
BT instrukcja, 146-148
BTC instrukcja, 146-148
BTR instrukcja, 146-148
BTS instrukcja, 146-148
BYTE dyrektywa, 334-335

C

CALL instrukcja, 215-216
CATSTR dyrektywa, 336
CBW instrukcja, 149
CDQ instrukcja, 150
CLC instrukcja, 258
CLD instrukcja, 259
CLI instrukcja, 260
CLTS instrukcja, 261
CMC instrukcja, 262-263
CMP instrukcja, 151-152
CMPS instrukcja, 153
CMPSB instrukcja, 153
CMPSW instrukcja, 153
@Code dyrektywa, 302, 304
CODESEG dyrektywa, 304
@CodeSize dyrektywa, 303
COMM dyrektywa, 402-403
%CONDS dyrektywa, 450
.CREF dyrektywa, 451
%CREFALL dyrektywa, 452
%CREFREF dyrektywa, 452
%CREFUREF dyrektywa, 452
%CTLS dyrektywa, 453
@CurSeg dyrektywa, 305
CWD instrukcja, 154
CWDE instrukcja, 155

D

DAA instrukcja, 156
DAS instrukcja, 157-158
@Data dyrektywa, 306
.DATA? dyrektywa, 306
DATASEG dyrektywa, 306
@DataSize dyrektywa, 306
DB dyrektywa, 337
DD dyrektywa, 337
DEC instrukcja, 159
%DEPTH dyrektywa, 455
DF dyrektywa, 338
DISPLAY dyrektywa, 454
DIV instrukcja, 160-161
DOSSEG dyrektywa, 301, 307
DQ dyrektywa, 338
DT dyrektywa, 339
DUP dyrektywa, 340
DW dyrektywa, 340
DWORD dyrektywa, 341

E

ELSE dyrektywa, 404
ELSEIF dyrektywa, 405
ELSEIF1 dyrektywa, 405
ELSEIF2 dyrektywa, 405
ELSEIFB dyrektywa, 405
ELSEIFDEF dyrektywa, 405
ELSEIFDIF dyrektywa, 405
ELSEIFDIFI dyrektywa, 405
ELSEIFE dyrektywa, 405

ELSEIFIDN dyrektywa, 405
ELSEIFIDNI dyrektywa, 405
ELSEIFNB dyrektywa, 405
ELSEIFNDEF dyrektywa, 405
EMUL dyrektywa, 455
END dyrektywa, 307
ENDIF dyrektywa, 406
ENDM dyrektywa, 407
ENDP dyrektywa, 408
ENDS dyrektywa, 308, 341
ENTER instrukcja, 217-218
EQ dyrektywa, 343
.ERR dyrektywa, 457
.ERR1 dyrektywa, 457
.ERR2 dyrektywa, 457
.ERRB dyrektywa, 457
.ERRDEF dyrektywa, 457
.ERRDIF dyrektywa, 457
.ERRDIFI dyrektywa, 457
.ERRE dyrektywa, 457
.ERRIDN dyrektywa, 457
.ERRIDNI dyrektywa, 457
ERRIF dyrektywa, 457
ERRIF1 dyrektywa, 457
ERRIF2 dyrektywa, 457
ERRIFB dyrektywa, 457
ERRIFDEF dyrektywa, 457
ERRIFDIF dyrektywa, 457
ERRIFDIFI dyrektywa, 457
ERRIFE dyrektywa, 457
ERRIFIDN dyrektywa, 457
ERRIFIDNI dyrektywa, 457
ERRIFNB dyrektywa, 457
ERRIFNDEF dyrektywa, 457
.ERRNB dyrektywa, 457
.ERRNDEF dyrektywa, 457
.ERRNZ dyrektywa, 457
ESC instrukcja, 264
EVEN dyrektywa, 342
EVENDATA dyrektywa, 342
EXITM dyrektywa, 409-410
EXTRN dyrektywa, 411-412

F

F1h-FDh przerwania, 671
FAR dyrektywa, 344
@FarData dyrektywa, 308-309
@FarData dyrektywa, 308-309
FEh-FFh przerwania, 671
@FileName dyrektywa, 413
FWORD dyrektywa, 345

G

GE dyrektywa, 346
GLOBAL dyrektywa, 413-414
GROUP dyrektywa, 310
GT dyrektywa, 347

H

HIGH dyrektywa, 347
HLT instrukcja, 265-266

I

IDEAL dyrektywa, 458
IDIV instrukcja, 162
IF dyrektywa, 415
IF1 dyrektywa, 416
IF2 dyrektywa, 416
IFB dyrektywa, 416
IFDEF dyrektywa, 416
IFDIF dyrektywa, 416
IFDIFI dyrektywa, 416
IFE dyrektywa, 416
IFIDN dyrektywa, 416
IFIDNI dyrektywa, 416
IFNB dyrektywa, 416
IFNDEF dyrektywa, 416
IMUL instrukcja, 163-165
IN instrukcja, 71
INC instrukcja, 166
%INCL dyrektywa, 459
INCLUDE dyrektywa, 418
INCLUDELIB dyrektywa, 418
INS instrukcja, 72
INSB instrukcja, 72
INSD instrukcja, 72
INSTR dyrektywa, 348
INSW instrukcja, 72
INT instrukcja, 219-220
INTO instrukcja, 221-222
IRET instrukcja, 223-224
IRETD instrukcja, 223-224
IRP dyrektywa, 419
IRPC dyrektywa, 420

J

JA instrukcja, 226
JAE instrukcja, 226
JB instrukcja, 226
JBE instrukcja, 226

JC instrukcja, 226
JCXZ instrukcja, 227
JE instrukcja, 226
JECXZ instrukcja, 227
JG instrukcja, 226
JGE instrukcja, 226
JL instrukcja, 226
JLE instrukcja, 226
JMP instrukcja, 228-229
JNA instrukcja, 226
JNAE instrukcja, 226
JNB instrukcja, 226
JNBE instrukcja, 226
JNC instrukcja, 226
JNE instrukcja, 226
JNG instrukcja, 226
JNGE instrukcja, 226
JNL instrukcja, 226
JNLE instrukcja, 226
JNO instrukcja, 226
JNP instrukcja, 226
JNS instrukcja, 226
JNZ instrukcja, 226
JO instrukcja, 226
JP instrukcja, 226
JPE instrukcja, 226
JPO instrukcja, 226
JS instrukcja, 226
JUMPS dyrektywa, 460
JZ instrukcja, 226

K

kody znaków, 674-676

L

LABEL dyrektywa, 349
LAHF instrukcja, 267
.LALL dyrektywa, 461
LAR instrukcja, 268
LARGE dyrektywa, 349-350
LDS instrukcja, 73
LE dyrektywa, 351
LEA instrukcja, 75-76
LEAVE instrukcja, 230
LENGTH dyrektywa, 351
LES instrukcja, 74
.LFCOND dyrektywa, 462
LFS instrukcja, 75
LGDT instrukcja, 269-270
LGS instrukcja, 75
LIDT instrukcja, 269-270
%LINUM dyrektywa, 462
.LIST dyrektywa, 463
LLDT instrukcja, 269-270
LMSW instrukcja, 271
LOCAL dyrektywa, 421
LOCALS dyrektywa, 422
LOCK instrukcja, 272
LODS instrukcja, 77
LOOP instrukcja, 231-232
LOOPE instrukcja, 231-232
LOOPNE instrukcja, 231-232
LOW dyrektywa, 352
LSL instrukcja, 273-274
LSS instrukcja, 75
LT dyrektywa, 352
LTR instrukcja, 275-278

M

MACRO dyrektywa, 423
%MACS dyrektywa, 464
MASK dyrektywa, 353
MASM dyrektywa, 464
MASM51 dyrektywa, 465
MOD dyrektywa, 353
MODEL dyrektywa, 311
MOV instrukcja, 78-80
MOVS instrukcja, 81-82
MOVSX instrukcja, 83-84
MOVZX instrukcja, 85
MUL instrukcja, 167
MULTERRS dyrektywa, 466

N

NAME dyrektywa, 466
NE dyrektywa, 354
NEAR dyrektywa, 354-355
NEG instrukcja, 168-169
%NEWPAGE dyrektywa, 467
%NOCONDS dyrektywa, 468
%NOCREF dyrektywa, 468
%NOCTLS dyrektywa, 468
NOEMUL dyrektywa, 468
%NOINCL dyrektywa, 468
NOJUMPS dyrektywa, 468
%NOLIST dyrektywa, 469
NOLOCALS dyrektywa, 422
%NOMACS dyrektywa, 469
NOMASM51 dyrektywa, 469
NOMULTERRS dyrektywa, 469
%NOSYMS dyrektywa, 469
NOT dyrektywa, 356

NOT instrukcja, 170
%NOTRUNC dyrektywa, 469
NOWARN dyrektywa, 470

O

OFFSET dyrektywa, 356
OR dyrektywa, 357
OR instrukcja, 171
ORG dyrektywa, 358
%OUT dyrektywa, 470
OUT instrukcja, 86
OUTS instrukcja, 87-88

P

P186 dyrektywa, 471
P286 dyrektywa, 471
P286N dyrektywa, 471
P287 dyrektywa, 471
P386 dyrektywa, 471
P386N dyrektywa, 471
P387 dyrektywa, 472
P8086 dyrektywa, 472
P8087 dyrektywa, 472
PAGE dyrektywa, 474
%PAGESIZE dyrektywa, 475
%PCNT dyrektywa, 476
PNO87 dyrektywa, 472-473
POP instrukcja, 89
POPA instrukcja, 90
POPAD instrukcja, 90
POPF instrukcja, 91-92
POPFD instrukcja, 91-92
%POPLCTL dyrektywa, 477
port szeregowy, 547
PROC dyrektywa, 358-359, 424-425
procesory 80x86, 3
    pamięć, 4
    przerwania, 7
    rejestry, 5
    stos, 6
przerwania sprzętowe, 539
    8259 kontroler przerwań, 540
    Interrupt Service Routine - ISR, 539
    sterownik klawiatury, 542-544
    tablica wektorów przerwań (IVT), 540
PTR dyrektywa, 360
PUBLIC dyrektywa, 426
PURGE dyrektywa, 426
PUSH instrukcja, 93
PUSHA instrukcja, 94
PUSHAD instrukcja, 94
PUSHF instrukcja, 95
PUSHFD instrukcja, 95
%PUSHLCTL dyrektywa, 478
PWORD dyrektywa, 361

Q

QUIRKS dyrektywa, 478
QWORD dyrektywa, 361

R

RADIX dyrektywa, 362
RCL instrukcja, 172-173
RCR instrukcja, 174
RECORD dyrektywa, 363
REP instrukcja, 96
REPT dyrektywa, 427-428
RET instrukcja, 233-234
RETF instrukcja, 233-234
RETN instrukcja, 233-234
ROL instrukcja, 175-176
ROR instrukcja, 177-178

S

SAHF instrukcja, 279
SAL instrukcja, 179-180
.SALL dyrektywa, 479
SAR instrukcja, 181
SBB instrukcja, 187-189
SCAS instrukcja, 190
SCASB instrukcja, 190
SCASW instrukcja, 190
SEG dyrektywa, 364
SEGMENT dyrektywa, 312
.SEQ dyrektywa, 301, 313
SETA instrukcja, 98
SETAE instrukcja, 97
SETBE instrukcja, 98
SETC instrukcja, 98
SETE instrukcja, 98
SETG instrukcja, 98
SETGE instrukcja, 98
SETL instrukcja, 98
SETLE instrukcja, 98
SETNA instrukcja, 98
SETNB instrukcja, 97
SETNBE instrukcja, 98
SETNC instrukcja, 98
SETNE instrukcja, 98
SETNG instrukcja, 98
SETNGE instrukcja, 98

SETNL instrukcja, 98
SETNLE instrukcja, 98
SETNO instrukcja, 98
SETNP instrukcja, 98
SETNS instrukcja, 98
SETNZ instrukcja, 98
SETO instrukcja, 98
SETP instrukcja, 98
SETPE instrukcja, 98
SETPO instrukcja, 98
SETS instrukcja, 98
SETZ instrukcja, 98
.SFCOND dyrektywa, 479
SGDT instrukcja, 280
SHL dyrektywa, 365
SHL instrukcja, 182
SHLD instrukcja, 183
SHORT dyrektywa, 366
SHR dyrektywa, 367
SHR instrukcja, 184-185
SHRD instrukcja, 186
SIDT instrukcja, 280
SIZE dyrektywa, 367
SIZESTR dyrektywa, 368
SLDT instrukcja, 280
SMALL dyrektywa, 369
SMSW instrukcja, 281
STACK dyrektywa, 313
STD instrukcja, 282
STI instrukcja, 283
STOS instrukcja, 100
STR instrukcja, 283
STRUC dyrektywa, 370-371
SUB instrukcja, 191-192
SUBSTR dyrektywa, 372
SUBTTL dyrektywa, 479
%SYMS dyrektywa, 480
SYMTYPE dyrektywa, 374

T

%TABSIZE dyrektywa, 481
TBYTE dyrektywa, 372
TEST instrukcja, 193-194
%TEXT dyrektywa, 481
.TFCOND dyrektywa, 482
THIS dyrektywa, 373
TITLE dyrektywa, 483
%TRUNC dyrektywa, 484
tryby adresowania, 56
    bazowy, 56
    bazowy indeksowany, 56
    bazowy indeksowany z przemieszczeniem, 57
    bazowy z przemieszczeniem, 56
    bezpośredni, 56
    domyślny, 56
    indeksowy, 56
    indeksowy z przemieszczeniem, 56
    natychmiastowy, 56
    skalowany bazowy indeks. z przemieszcz., 57
    skalowany bazowy z przemieszczeniem, 57
    skalowany indeksowy z przemieszczeniem, 57
TYPE dyrektywa, 375

U

UDATASEG dyrektywa, 314
UFARDATA dyrektywa, 314
układ 8250, 547
UNION dyrektywa, 376
UNKNOWN dyrektywa, 377

V

VERR instrukcja, 284-285
VERW instrukcja, 284-285

W

WAIT instrukcja, 286
WARN dyrektywa, 485
WIDTH dyrektywa, 378
WORD dyrektywa, 379

X

.XALL dyrektywa, 486
XCHG instrukcja, 101
.XCREF dyrektywa, 486
XLAT instrukcja, 102-103
XLATB instrukcja, 102-103
.XLIST dyrektywa, 486
XOR instrukcja, 195-196

Biblioteka Gł. AGH
1000007287