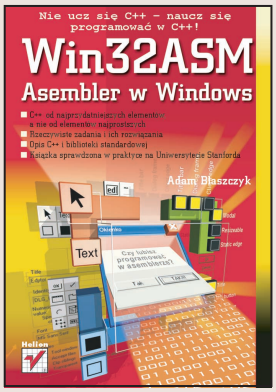
Win32ASM. Asembler w Windows

Autor: Adam Błaszczyk
ISBN: 83-7361-022-7
Format: B5, stron: 360



Przekonanie, że programowanie w asemblerze odeszło w przeszłość wraz z opartymi na interfejsie tekstowym systemami w rodzaju DOS i upowszechnieniem się wysokopoziomowych języków programowania takich jak Visual Basic, C++ czy Java, jest błędne. Całkiem spora grupa osób nadal używa asemblera w środowisku Windows. Jeśli chcesz do nich dołączyć, znalazłeś właściwą książkę.

Asembler daje programiście poczucie ogromnej wolności i pełnego panowania nad sprzętem. Dzięki tej książce nauczysz się programować w tym języku, poznasz również cały zestaw aplikacji, które programowanie to ułatwią. W tekście książki znajdziesz wiele przykładów kodu ilustrujących najważniejsze omawiane zagadnienia.
Książka przedstawia:

- Podstawowe informacje związane z asemblerem: systemy kodowania, zapis liczb i znaków
- Rejestry i pamięć maszyny z punktu widzenia programisty asemblera, adresowanie
- Instrukcje asemblera i listę rozkazów procesora
- Programowanie z użyciem MASM – Makro Asemblera
- Użycie programu NMAKE i tworzenie plików Makefile
- Programowanie w Windows: WinAPI
- Biblioteki DLL
- Technologie COM i OLE
- Zasoby programów Windows

Jeszcze jedna metoda tworzenia programów w Windows, która może zafascynować osoby lubiące „dłubanie w kodzie”? Dlaczego nie! Przeczytaj tę książkę, a sam będziesz mógł ocenić, czy asembler się zdewaluował.

Programowanie w Windows dla wtajemniczonych.

- Poznaj język asembler dla procesorów Intel i kompatybilnych
- Naucz się używać Windows API z poziomu asemblera
- Poznaj narzędzia ułatwiające programowanie

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 5.

Macro Assembler

Niniejsza część książki poświęcona jest całkowicie przybliżeniu podstawowych i najważniejszych z punktu widzenia programisty informacji na temat języka Macro Assembler.

MASM — Macro Assembler

Macro Assembler to język całkowicie typowy i nic dziwnego, że sterują nim pewne, odgórnie ustalone zasady. Przede wszystkim jest to zwykły asembler, w którym można tworzyć programy w klasyczny sposób — pisząc kolejne instrukcje w osobnych linijkach. Prawdziwa siła Macro Assemblera tak naprawdę tkwi jednak w makrach. Czym są i jak używać ich skutecznie w swoich programach opowiemy za chwilę, zastanówmy się jednak najpierw, co jest największą bolączką programistów piszących w języku asembler. Odpowiedź na to pytanie znalazła się już w poprzednim podrozdziale — chodzi o chaos widoczny w większości programów pisanych w tym języku. Jeśli masz już jakieś doświadczenia jako programista pracujący w asemblerze, zapewne znasz doskonale uczucie towarzyszące przeglądaniu programów źródłowych, szczególnie tych pochodzących z przeszłości lub pisanych przez innych programistów. Sporo czasu zajmuje zwykle zrozumienie sedna sprawy i to tutaj właśnie Macro Assembler ma okazję pokazać swoje największe zalety, a już szczególnie swój zbawienny wpływ na przejrzystość kodu.

Weźmy na przykład bardzo często stosowaną w asemblerze konstrukcję:

```
cmp  eax,1
je   eax_1
cmp  eax,2
je   eax_2
...
cmp  eax,3
je   eax_3

...
```

```
eax_1:
 ...
 jmp dalej
eax_2:
 ...
 jmp dalej
eax_3:
 ...
 jmp dalej

 ...

dalej:
```

która pozwala na wykonywanie różnego kodu w zależności od określonych warunków. W przypadku jednego rozgałęzienia nie ma problemu, natomiast komplikacje są od razu widoczne przy większej liczbie warunków. Macro Assembler świetnie rozwiązuje tę kwestię, pozwalając na uproszczenie tego typu konstrukcji za pomocą bloku `.IF`, który dla powyższego przykładu moglibyśmy zastosować w następujący sposób:

```
.IF  eax==1
 ...
.ELSEIF  eax==2
 ...
.ELSEIF  eax==3
 ...
.END
```

Od razu rzucają się w oczy trzy rzeczy. Po pierwsze, zapis jest o wiele krótszy; po drugie, nie musimy korzystać z uciążliwych etykiet; po trzecie — składnia bardzo przypomina zapis języka wyższego poziomu. I tak jest w istocie. Istnieje nawet opinia, iż programowanie w Macro Asemblerze w niedużym stopniu różni się od programowania w języku C lub podobnym. Chyba już wiesz, jaką potęgę skupia w sobie ten niepozorny język?

Tak, tak…

Składnia języka wysokiego poziomu, instrukcje asemblera, bezpośredni dostęp do pamięci, bezpośredni dostęp do funkcji Win API, możliwość ingerencji w kod programu na najniższym poziomie i wiele innych czynników, włączając w to jeden drobny, acz chyba najbardziej lubiany przez wszystkich koderów szczegół — programy wynikowe są o wiele krótsze niż pisane w językach wysokiego poziomu. Odpowiednio zdefiniowany zestaw makr może także służyć do wykrywania błędów w programie (ułatwione śledzenie programu, dzięki informacjom na temat jego wykonywania).

Poznaliśmy już konstrukcję makra `.IF`. Jeszcze raz popatrzmy, jak wygląda ono w praktyce, biorąc za przykład wykonanie pewnego typowego kodu w zależności od jakiegoś warunku. Na szybko wymyślmy sytuację, w której chcemy wyświetlić na ekranie komunikat o błędzie, jeśli wynik zwrócony przez funkcję otwierającą plik jest niepoprawny.

I tak, zapisany w typowy sposób kod miałby postać:

```
 invoke OpenFile,...    ; w tym przykładzie parametry nas nie interesują
 cmp    eax,-1          ; -1 oznacza tutaj kod błędu
 je     Blad            ; jeśli błąd — skok do etykiety Blad
 ...                    ; \ dalsza część programu, wykonująca się
                        ; / jeśli nie ma błędu
Blad:
 ...                    ; obsługa błędu
```

Z kolei w języku MASM taka operacja może zostać zastąpiona kodem:

```
invoke OpenFile,...     ; w tym przykładzie parametry nas nie interesują
.IF eax!=-1             ; -1 oznacza tutaj kod błędu
 ...                    ; \ dalsza część programu, wykonująca się
                        ; / jeśli nie ma błędu
.ENDIF
 ...                    ; obsługa błędu
```

Jak powstają makra?

Makra w MASM to bloki tekstu, które mogą być dołączane w dowolnym momencie kodu. Definicja makra nadaje mu nazwę i opisuje, co dane makro ma robić — z praktycznego punktu widzenia makro ma ogólną postać typu:

```
Nazwa MACRO argumenty
   ...      ; tutaj umieść treść makra
ENDM
```

Dla przykładu makro, którego celem jest wstawienie do kodu funkcji tzw. epilogu zwracającego daną wartość w rejestrze `eax`, może mieć postać:

```
RETURN MACRO co_zwracamy
  mov eax, co_zwracamy
  ret
ENDM
```

Wywołanie zdefiniowanego makra wygląda następująco (w przykładzie zwracamy `-1`):

```
RETURN -1
```

Umieszczenie powyższego kodu w programie powoduje, że analizujący kod asembler po napotkaniu tego wyrażenia „rozwinie" je w następujący kod:

```
mov  eax, co_zwracamy
ret
```

a za wartość `co_zwracamy` zostanie podstawiona wartość `-1`, dając finalną wersję kodu:

```
mov eax, -1
ret
```

Powyższy przykład przewija się w wielu programach źródłowych Win32ASM, jakie można znaleźć w internecie.

Najważniejsze informacje o makrach

Pisanie makr oraz wykorzystanie możliwości języka Macro Assembler są związane z koniecznością zapoznania się z najważniejszymi makrami, symbolami oraz operatorami używanymi do zapisu wyrażeń różnego typu. Zaczniemy od znaków o specjalnym znaczeniu.

Znaki specjalne

`$`

— oznacza aktualną (tj. w danym momencie procesu asemblacji) wartość licznika określającego adres bieżącej instrukcji; w praktyce można traktować to jako symboliczny zapis adresu miejsca, w którym symbol ten wystąpił.

`?`

— symbol używany podczas deklaracji danych; jego zastosowanie informuje asembler, że zmienna zadeklarowana przy użyciu symbolu `?` nie ma wartości początkowej.

`@`

— symbol używany przy deklaracji etykiet oraz w nazwach przedefiniowanych symboli (opisanych w następnym podrozdziale).

`<>`

— symbole używane przy deklaracji zmiennych lub stałych tekstowych oraz podczas łączenia wielu parametrów w jeden ciąg (traktowany jako jeden parametr).

`[]`

— symbole używane przy odwoływaniu się do komórek pamięci.

Zapis liczb

Liczba`d`

— reprezentuje *Liczbę* dziesiętną; z reguły nie trzeba dodawać przyrostka `d`, chyba że zostanie zmieniony domyślny system liczbowy (używany przez asembler w trakcie asemblacji).

`0`*Liczba*`h`

— reprezentuje *Liczbę* szesnastkową (heksadecymalną); przedrostek `0` dodaje się w przypadku liczb rozpoczynających się od cyfr szesnastkowych `A`, `B`, `C`, `D`, `E` lub `F`.

Liczba`b`

— reprezentuje *Liczbę* dwójkową (binarną).

Przedefiniowane symbole i funkcje

`@Date`

— reprezentuje aktualną datę.

`@Time`
— reprezentuje aktualny czas.

`@Version`
— reprezentuje wersję programu Macro Assembler.

`@FileCur`
— reprezentuje nazwę pliku poddawanego asemblacji.

`@FileName`
— reprezentuje pełną nazwę pliku poddawanego asemblacji.

`@code`
— dostarcza informacji o segmencie kodu.

`@data`
— dostarcza informacji o segmencie danych.

`@stack`
— dostarcza informacji o segmencie stosu.

`@CurSeg`
— dostarcza informacji o bieżącym (tzn. tym, w którym aktualnie użyty jest symbol `@CurSeg`) segmencie.

Poniższy przykład pokazuje, w jaki sposób można wykorzystać opisane symbole do wyświetlania informacji diagnostycznych w trakcie kompilowania programu.

archiwum: *\Win32ASM\Src\!Symbole\!Symbole.asm*

```
; ============================================================
; Opis      : Demonstracja wykorzystania różnych
;             funkcji wbudowanych w Macro Assembler
; ============================================================

.586                              ; instrukcje 586
.MODEL FLAT,STDCALL               ; płaski model pamięci
                                        ; konwencja STDCALL

.code                             ; sekcja kodu
  Start:                          ; początek programu
ret                               ; powrót z procedury

%ECHO Data (Date)                 = @Date
%ECHO Czas (Time)                 = @Time
%ECHO Wersja (Version)            = @Version
%ECHO Nazwa (FileCur)             = @FileCur
%ECHO Nazwa pełna (FileName)      = @FileName
%ECHO Segment kodu (code)         = @code
%ECHO Segment danych (data)       = @data
%ECHO Segment stosu (stack)       = @stack
%ECHO Segment aktualny (CurSeg)   = @CurSeg
END Start                         ; tu zaczyna się program
```

A oto rezultat przykładowej asemblacji programu.

archiwum: *\Win32ASM\Src\!Symbole\!Symbole.txt*

```
Info:
Ścieżka do MASM: c:\Masm32
Tworzę projekt "!Symbole".
Wersja RELEASE, plik EXE bez zasobów.
Assembling: !Symbole.asm
Data (Date)               = 08/24/03
Czas (Time)               = 17:32:14
Wersja (Version)          = 614
Nazwa (FileCur)           = !Symbole.asm
Nazwa pełna (FileName)    = !SYMBOLE
Segment kodu (code)       = _TEXT
Segment danych (data)     = FLAT
Segment stosu (stack)     = FLAT
Segment aktualny (CurSeg) = _TEXT
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Operatory

Operatory matematyczne

Wyrażenie1 + *Wyrażenie2*
— zwraca sumę obu *Wyrażeń*.

Wyrażenie1 – *Wyrażenie2*
— zwraca różnicę obu *Wyrażeń*.

Wyrażenie1 * *Wyrażenie2*
— zwraca iloczyn obu *Wyrażeń*.

Wyrażenie1 / *Wyrażenie2*
— zwraca iloraz obu *Wyrażeń*.

- *Wyrażenie*
— zwraca *Wyrażenie* ze znakiem zmienionym na przeciwny.

Wyrażenie1 MOD *Wyrażenie2*
— zwraca resztę z dzielenia (funkcja `MOD`, funkcja `MODULO`) *Wyrażenia1* przez *Wyrażenie2*.

Operatory logiczne

Wyrażenie1 AND *Wyrażenie2*
— zwraca iloczyn logiczny (funkcja `AND`) obu *Wyrażeń*.

Wyrażenie1 OR *Wyrażenie2*
— zwraca sumę logiczną (funkcja `OR`) obu *Wyrażeń*.

`NOT` *Wyrażenie1*

— zwraca negację logiczną (funkcja `NOT`) *Wyrażenia*.

Wyrażenie1 `XOR` *Wyrażenie2*

— zwraca sumę modulo (funkcja `XOR`, inaczej `EXCLUSIVE OR`) obu *Wyrażeń*.

Operatory przesunięć

Wyrażenie `SHL` *Liczba*

— zwraca wartość powstałą z przesunięcia w lewo wartości *Wyrażenia* zadaną *Liczbę* razy.

Wyrażenie `SHR` *Liczba*

— zwraca wartość powstałą z przesunięcia w prawo wartości *Wyrażenia* zadaną *Liczbę* razy.

Operatory porównań

Wyrażenie1 `EQ` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* jest równe *Wyrażeniu2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Wyrażenie1 `NE` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* nie jest równe *Wyrażeniu2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Wyrażenie1 `GT` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* jest większe od *Wyrażenia2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Wyrażenie1 `LT` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* jest mniejsze od *Wyrażenia2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Wyrażenie1 `GE` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* jest większe lub równe *Wyrażeniu2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Wyrażenie1 `LE` *Wyrażenie2*

— zwraca wartość `true` (o wartości `-1`), jeśli *Wyrażenie1* jest mniejsze lub równe *Wyrażeniu2*; w przeciwnym przypadku zwraca wartość `false` (o wartości `0`).

Operatory inne

Segment`:`*Wyrażenie*

— powoduje, że *Wyrażenie* odnosi się do innego *Segmentu* niż domyślny; *Segment* może być nazwą rejestru, segmentu, grupy segmentów lub wyrażeniem stałym.

Wyrażenie`.`*Pole* `[[.`*Pole2*`]], …`

— zwraca adres będący sumą adresu *Wyrażenia* i adresu *Pola* w strukturze lub unii; w ten sposób można odwoływać się także do struktur zagnieżdżonych np. *Wyrażenie*`.` *Pole*`.`*PoleInne*; *Wyrażenie* może być nazwą rejestru lub zmiennej.

"Tekst"
— powoduje, że *Tekst* traktowany jest jako łańcuch tekstowy.

'Tekst'
— powoduje, że *Tekst* traktowany jest jako łańcuch tekstowy.

%Wyrażenie
— powoduje, że *Wyrażenie* będące parametrem przekazywanym do makra będzie traktowane jako łańcuch tekstowy.

&Wyrażenie&
— powoduje zastąpienie łańcucha *&Wyrażenie&* wartością *Wyrażenia*.

ADDR *Wyrażenie*
— zwraca adres *Wyrażenia*; najczęściej dotyczy zmiennych lokalnych.

OFFSET *Wyrażenie*
— zwraca adres (offset, przesunięcie) *Wyrażenia*; najczęściej dotyczy zmiennych globalnych.

SIZE *Wyrażenie*
— zwraca rozmiar (w bajtach) *Wyrażenia*.

Makra związane z procesem asemblacji

Przyjrzymy się teraz bliżej wszystkim ważniejszym poleceniom, jakie można wykorzystać w języku Macro Asembler. Na końcu tego rozdziału przedstawię plik *makro.inc*, w którym zawarłem najczęściej używane makra, z powodzeniem możesz ich użyć w swoich programach, bez konieczności pisania od „zera".

Makra o charakterze ogólnym

Definicja rodzaju procesora

.8086
— pozwala wykorzystywać instrukcje procesora 8086 i koprocesora 8087; ustawienie domyślne.

.8087
— pozwala wykorzystywać instrukcje koprocesora 8087; ustawienie domyślne.

.186
— pozwala wykorzystywać instrukcje procesora 80186 i koprocesora 8087.

.286
— pozwala wykorzystywać instrukcje procesora 80286 i koprocesora 80287; nie pozwala na skorzystanie z instrukcji uprzywilejowanych (dotyczy trybu chronionego).

.286P
— pozwala wykorzystywać instrukcje procesora 80286 i koprocesora 80287; pozwala na użycie instrukcji uprzywilejowanych (dotyczy trybu chronionego).

`.287`

— pozwala wykorzystywać instrukcje koprocesora 80287.

`.386`

— pozwala wykorzystywać instrukcje procesora 80386 i koprocesora 80387; nie pozwala na skorzystanie z instrukcji uprzywilejowanych (dotyczy trybu chronionego).

`.386P`

— pozwala wykorzystywać instrukcje procesora 80386 i koprocesora 80387; pozwala na użycie instrukcji uprzywilejowanych (dotyczy trybu chronionego).

`.387`

— pozwala wykorzystywać instrukcje koprocesora 80387.

`.486`

— pozwala wykorzystywać instrukcje procesora 80486 (koprocesor jest wbudowany w procesor); nie pozwala skorzystanie z instrukcji uprzywilejowanych (dotyczy trybu chronionego).

`.486P`

— pozwala wykorzystywać instrukcje procesora 80486 (koprocesor jest wbudowany w procesor); pozwala na użycie instrukcji uprzywilejowanych (dotyczy trybu chronionego).

`.NO87`

— nie pozwala wykorzystywać instrukcji koprocesora.

Komentarze

`COMMENT` *`Separator`* `[[`*`Tekst`*`]]`
`[[`*`Tekst`*`]]`
`[[`*`Tekst`*`]]` Separator

— powoduje, że pierwsza linia (z wyrażeniem `COMMENT` *`Separator`* `[[`*`Tekst`*`]]`) oraz wszystkie inne, w których występuje *`Separator`* będą traktowane przez asembler jako komentarz.

`;`*`Tekst`*

— powoduje, że *`Tekst`* traktowany będzie jako komentarz.

`;;`*`Tekst`*

— powoduje, że *`Tekst`* traktowany będzie jako komentarz tylko w definicji makra i nie będzie widoczny na listingu (po rozwinięciu makra).

Listingi

`TITLE` *`Tytuł`*

— definiuje *`Tytuł`* listingu programu.

`SUBTITLE` *`Podtytuł`*

lub

`SUBTTL` *`Podtytuł`*

— definiuje *`Podtytuł`* listingu.

Wyświetlanie informacji (w trakcie asemblacji)

`ECHO` *Komunikat*

— powoduje wyświetlenie *Komunikatu* w trakcie asemblacji.

`%ECHO` *Komunikat*

— powoduje wyświetlenie *Komunikatu* w trakcie asemblacji; znak `%` powoduje, że wszystkie makra tekstowe użyte w *Komunikacie* zostaną „rozwinięte".

`%OUT` *Komunikat*

— powoduje wyświetlenie *Komunikatu* w trakcie asemblacji.

Dołączanie kodu źródłowego umieszczonego w innym pliku

`INCLUDE` *NazwaPliku*

— powoduje, że w miejscu wystąpienia zapisu `INCLUDE` *NazwaPliku* zostanie dołączony tekst zapisany w pliku *NazwaPliku*; *NazwaPliku* powinna być objęta parą nawiasów ostrych <…>, jeśli w nazwie występuje którykolwiek ze znaków: \ (lewy ukośnik), ; (średnik), > (większy niż zero, nawias ostry zamykający), < (mniejszy niż zero, nawias ostry otwierający), ' (apostrof), " (znak cudzysłowia w standardzie ASCII; dokładniej: znak cala).

Dołączanie zewnętrznej biblioteki w trakcie budowania programu

`INCLUDELIB` *NazwaBiblioteki*

— powoduje, że w trakcie łączenia (budowania) pliku wynikowego program łączący dołączy bibliotekę zawartą w pliku *NazwaBiblioteki*; *NazwaBiblioteki* powinna być objęta parą nawiasów ostrych <…>, jeśli w nazwie występuje którykolwiek ze znaków: \ (lewy ukośnik), ; (średnik), > (większy niż zero, nawias ostry zamykający), < (mniejszy niż zero, nawias ostry otwierający), ' (apostrof), " (znak cudzysłowia w standardzie ASCII; dokładniej: znak cala).

Przerywanie procesu asemblacji

Konstrukcja .ERR

`.ERR [[`*Komunikat*`]]`

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*; *Komunikat* powinien być objęty parą nawiasów ostrych <…>.

Konstrukcja .ERR2

`.ERR2 [[`*Komunikat*`]]`

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*; *Komunikat* powinien być objęty parą nawiasów ostrych <…>; w przeciwieństwie do `.ERR`, błąd generowany będzie przy każdym przebiegu asemblera (opcja `OPTION:SETIF2` musi być równa wartości `TRUE`).

Konstrukcja .ERRB

.ERRB *Element* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element* jest pusty; *Element* i *Komunikat* muszą być objęte parą nawiasów ostrych <…>.

Konstrukcja .ERRNB

.ERRNB *Element* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element* nie jest pusty; *Element* i *Komunikat* muszą być objęte parą nawiasów ostrych <…>.

Konstrukcja .ERRDEF

.ERRDEF *Nazwa* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Nazwa* była już wcześniej zdefiniowana jako zmienna, etykieta lub symbol; *Komunikat* powinien być objęty parą nawiasów ostrych <…>.

Konstrukcja .ERRNDEF

.ERRNDEF *Nazwa* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Nazwa* nie była wcześniej zdefiniowana jako zmienna, etykieta lub symbol; *Komunikat* powinien być objęty parą nawiasów ostrych <…>.

Konstrukcja .ERRDIF

.ERRDIF *Element1*, *Element2* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element1* nie jest identyczny z *Elementem2*; *Element1* i *Element2* muszą być objęte parą nawiasów ostrych <…>; małe i wielkie litery są rozróżniane podczas operacji porównania.

Konstrukcja .ERRDIFI

.ERRDIFI *Element1*, *Element2* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element1* nie jest identyczny z *Elementem2*; *Element1* i *Element2* muszą być objęte parą nawiasów ostrych <…>; małe i wielkie litery nie są rozróżniane podczas operacji porównania.

Konstrukcja .ERRIDN

.ERRIDN *Element1*, *Element2* [[, *Komunikat*]]

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element1* jest identyczny z *Elementem2*; *Element1*

i *Element2* muszą być objęte parą nawiasów ostrych <…>; małe i wielkie litery są rozróżniane podczas operacji porównania.

Konstrukcja .ERRIDNI

```
.ERRIDNI Element1, Element2 [[, Komunikat]]
```

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Element1* jest identyczny z *Elementem2*; *Element1* i *Element2* muszą być objęte parą nawiasów ostrych <…>; małe i wielkie litery nie są rozróżniane podczas operacji porównania.

Konstrukcja .ERRE

```
.ERRE Wyrażenie [[, Komunikat]]
```

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Wyrażenie* zwraca wartość FALSE; *Komunikat* musi być objęty parą nawiasów ostrych <…>.

Konstrukcja .ERRNZ

```
.ERRNZ Wyrażenie [[, Komunikat]]
```

— powoduje wygenerowanie błędu i przerwanie asemblacji oraz ewentualnie wyświetlenie *Komunikatu*, jeśli *Wyrażenie* zwraca wartość TRUE; *Komunikat* musi być objęty parą nawiasów ostrych <…>.

Sterowanie przebiegiem asemblacji

Etykiety i skoki

```
GOTO Etykieta
```

— powoduje, że kolejną przetwarzaną przez asembler linią będzie linia, w której umieszczono zapis *:Etykieta* (tj. przed etykietą zawsze występuje dwukropek); makroinstrukcja GOTO może być wykorzystywana tylko i wyłącznie wewnątrz bloków: MACRO, FOR, FORC, REPEAT, WHILE.

```
:Etykieta
```

— definiuje *Etykietę*; do *Etykiety* można odwoływać się w innych miejscach programu za pomocą konstrukcji GOTO *Etykieta*.

Pętle FOR i IRP

```
FOR Parametr [[:REQ | := Domyślna]] <Łańcuch [[, Łańcuch2]] [[, …]] [[, ŁańcuchN]]>
   Operacja
ENDM
```

lub równoważne:

```
IRP Parametr [[:REQ | := Domyślna]] <Łańcuch [[, Łańcuch2]] [[, …]] [[, ŁańcuchN]]>
   Operacja
ENDM
```

gdzie:

`FOR lub IRP`

— określają początek pętli.

`Parametr` `[[:REQ | := Domyślna]]`

— oznacza *`Parametr`*, któremu można przypisać wartość *`Domyślną`*,

`<Łańcuch [[, Łańcuch2]] [[, …]] [[, ŁańcuchN]]>`

— oznaczają kolejne *`Łańcuchy`*, na których będzie wykonana zadana *`Operacja`* w kolejnych obiegach pętli.

`ENDM`

— oznacza koniec bloku.

Pętle FORC i IRPC

```
FORC
   Parametr, <Łańcuch>, Operacja
ENDM
```

lub równoważne:

```
IRPC
   Parametr, <Łańcuch>, Operacja
ENDM
```

gdzie:

`FORC lub IRPC`

— określają początek pętli.

`Parametr, <Łańcuch>, Operacja`

— oznaczają *`Parametr`*, któremu będzie przypisana kolejna litera z *`Łańcucha`*; na każdej z liter będzie wykonana zadana *`Operacja`*.

`ENDM`

— oznacza koniec bloku.

Pętla REPEAT i REPT

```
REPEAT Liczba
   Kod
ENDM
```

lub równoważne:

```
REPT Liczba
   Kod
ENDM
```

gdzie:

`REPEAT` *`Liczba`* `lub REPT` *`Liczba`*

— określają początek pętli oraz *`Liczbę`* powtórzeń generowanego *`Kodu`*.

`Kod`

— określa *`Kod`* powtarzany zadaną *`Liczbę`* razy.

`ENDM`

— oznacza koniec bloku.

Pętla WHILE

```
WHILE Wyrażenie
   Operacja
ENDM
```

gdzie:

`WHILE Wyrażenie`

— określają początek pętli, w której *Kod* będzie przetwarzany tak długo, jak długo *Wyrażenie* będzie zwracało wartość prawdziwą.

Kod

— określa *Kod* asemblowany (w kółko) tak długo, jak długo *Wyrażenie* będzie zwracało wartość prawdziwą.

`ENDM`

— oznacza koniec bloku.

Asemblacja warunkowa

Asemblacja warunkowa pozwala na asemblację wybranych fragmentów kodu w zależności od pewnych parametrów.

Konstrukcja IF

```
IF Warunek
   Wyrażenie
[[ELSExxxx Warunek2
   Wyrażenie2]]
…
[[ELSExxxx WarunekN
   WyrażenieN]]
[[ELSE
   WyrażenieInne]]
ENDIF
```

gdzie:

`IF Warunek`
Wyrażenie

— oznacza początek konstrukcji — *Wyrażenie* zostanie poddane asemblacji po wystąpieniu określonego *Warunku*.

`[[ELSExxxx Warunek2`
Wyrażenie2`]]`
…
`[[ELSExxxx WarunekN`
WyrażenieN`]]`

— oznacza kolejne bloki (*Wyrażenie2*, *Wyrażenie3*, …, *WyrażenieN*) wykonywane, jeśli spełnione są odpowiednio: `ELSExxxx` *Warunek2*, `ELSExxxx` *Warunek3*, …, `ELSExxxx` *WarunekN*, gdzie `ELSExxxx` może przyjmować wartości:

- `ELSEIF` — jeśli… (uproszczona wersja zapisu `ELSE IF`),
- `ELSEIFB` — jeśli jest puste…,

- ELSEIFDEF — jeśli jest zdefiniowane…,
- ELSEIFDIF — jeśli następne wyrażenie jest inne (małe i wielkie litery są rozróżniane)…,
- ELSEIFDIFI — jeśli następne wyrażenie jest inne (małe i wielkie litery nie są rozróżniane)…,
- ELSEIFE — jeśli jest równe…,
- ELSEIFIDN — jeśli następne wyrażenie jest identyczne (małe i wielkie litery są rozróżniane)…,
- ELSEIFIDNI — jeśli następne wyrażenie jest identyczne (małe i wielkie litery nie są rozróżniane)…,
- ELSEIFNB — jeśli jest niepuste…,
- ELSEIFNDEF — jeśli nie jest zdefiniowane….

```
[[ELSE
   WyrażenieInne]]
```

— określa *WyrażenieInne* wykonywane tylko wtedy, jeśli żaden z powyższych warunków, tj. ELSExxxx *Warunek2*, ELSExxxx *Warunek3*, …, ELSExxxx *WarunekN*, nie został spełniony.

```
ENDIF
```

— kończy konstrukcję.

Konstrukcja IFB

```
IFB Element
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element* jest pusty; składnia jest identyczna jak w konstrukcji IF.

Konstrukcja IFDEF

```
IFDEF Element
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element* był wcześniej zdefiniowany jako symbol, zmienna lub etykieta; składnia jest identyczna jak w konstrukcji IF.

Konstrukcja IFE

```
IFE Element
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element* jest równy 0 (wartość false); składnia jest identyczna jak w konstrukcji IF.

Konstrukcja IFDIF

```
IFDIF Element1, Element2
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element1* i *Element2* są różne (małe i wielkie litery są rozróżniane); składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IFDIFI

```
IFDIFI Element1, Element2
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element1* i *Element2* są różne (małe i wielkie litery nie są rozróżniane); składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IFIDN

```
IFIDN Element1, Element2
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element1* i *Element2* są identyczne (małe i wielkie litery są rozróżniane); składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IFIDNI

```
IFIDNI Element1, Element2
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element1* i *Element2* są identyczne (małe i wielkie litery nie są rozróżniane); składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IFNB

```
IFNB Element
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element* jest niepusty; składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IFNDEF

```
IFNDEF Element
   Wyrażenie
…
```

— powoduje asemblację *Wyrażenia*, jeśli *Element* nie był wcześniej zdefiniowany; składnia jest identyczna jak w konstrukcji `IF`.

Konstrukcja IF2

```
IF2 Warunek
   Wyrażenie
…
```

— konstrukcja warunkowa `IF2` ma identyczną składnię jak `IF`; różnica polega na tym, że w przypadku konstrukcji `IF` asembler interpretuje *Warunek* tylko raz, natomiast w przypadku `IF2` — przy każdym przebiegu (opcja `OPTION:SETIF2` musi być równa wartości `TRUE`).

Konstrukcja ELSEIF2

```
ELSEIF2 Warunek
   Wyrażenie
…
```

— konstrukcja warunkowa `ELSEIF2` ma identyczną składnię jak `ELSEIF2`; w przypadku konstrukcji `ELSEIF` asembler interpretuje *Warunek* tylko raz, natomiast w przypadku `IF2` — przy każdym przebiegu (opcja `OPTION:SETIF2` musi być równa wartości `TRUE`).

Operatory związane z programowaniem

Operatory porównań matematycznych

Wyrażenie1 `==` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* jest równe *Wyrażeniu2*?

Wyrażenie1 `!=` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* nie jest równe *Wyrażeniu2*?

Wyrażenie1 `>` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* jest większe od *Wyrażenia2*?

Wyrażenie1 `>=` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* jest większe lub równe *Wyrażeniu2*?

Wyrażenie1 `<` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* jest mniejsze od *Wyrażenia2*?

Wyrażenie1 `<=` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* jest mniejsze lub równe *Wyrażeniu2*?

Operatory porównań logicznych

Wyrażenie1 `&&` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* i *Wyrażenie2* są prawdziwe?

Wyrażenie1 `||` *Wyrażenie2*

— znaczenie: czy *Wyrażenie1* lub *Wyrażenie2* są prawdziwe?

Wyrażenie1 AND *Wyrażenie2*

— znaczenie: czy iloczyn logiczny *Wyrażenia1* i *Wyrażenia2* jest prawdziwy?

!*Wyrażenie*

— znaczenie: czy przeciwieństwo *Wyrażenia* jest poprawne?; negacja logiczna.

CARRY?

— znaczenie: czy jest ustawiony znacznik przeniesienia CF w rejestrze znaczników EFLAGS?

OVERFLOW?

— znaczenie: czy jest ustawiony znacznik nadmiaru OF w rejestrze znaczników EFLAGS?

PARITY?

— znaczenie: czy jest ustawiony znacznik przepełnienia PF w rejestrze znaczników EFLAGS?

PARITY?

— znaczenie: czy jest ustawiony znacznik znaku SF w rejestrze znaczników EFLAGS?

ZERO?

— znaczenie: czy jest ustawiony znacznik zera ZF w rejestrze znaczników EFLAGS?

Podział programu na sekcje (segmenty)

Deklaracja sekcji (segmentów)

.CODE [[*Nazwa*]]

— określa początek sekcji (segmentu) kodu; może występować wielokrotnie w treści programu; istnieje możliwość nadania tej sekcji programu wybranej *Nazwy*; domyślną nazwą dla modelu FLAT jest _TEXT.

.DATA

— określa początek sekcji (segmentu) danych zainicjowanych (z wartościami początkowymi, z wartościami znanymi w momencie asemblacji); może występować wielokrotnie w treści programu; nazwą tej sekcji dla modelu FLAT jest _DATA.

.DATA?

— określa początek sekcji (segmentu) danych niezainicjowanych (bez wartości początkowych, wartości nie są znane w momencie asemblacji); może występować wielokrotnie w treści programu; nazwą tej sekcji dla modelu FLAT jest _BSS.

.CONST

— określa początek sekcji (segmentu) danych zainicjowanych (z wartościami początkowymi, z wartościami znanymi w momencie asemblacji); może występować wielokrotnie w treści programu; sekcja ta posiadać będzie atrybut „tylko do odczytu”; nazwą tej sekcji dla modelu FLAT jest CONST.

Sygnalizacja końca modułu, segmentu oraz początku programu

`END`
— określa koniec bieżącego modułu.

`END` *Etykieta*
— określa początek programu jako tego miejsca (tej linii kodu) w programie, w którym występuje deklaracja *Etykiety*; typowo używa się zapisu `END` *Start*.

Nazwa `ENDS`
— określa koniec segmentu o podanej *Nazwie*.

Inne konstrukcje dotyczące sekcji (segmentów)

`.ALPHA`
— ustawia segmenty w kolejności alfabetycznej.

`.SEQ`
— ustawia segmenty w kolejności występowania w kodzie; ustawienie domyślne.

Grupa `GROUP` *Segment* `[[,` *Segment2*`]] [[, …]] [[,` *SegmentN*`]]`
— dodaje zadane segmenty *Segment*, *Segment2*, …, *SegmentN* do *Grupy*.

`ASSUME` *Rejestr*`:`*Nazwa* `[[,` *Rejestr2*`:`*Nazwa*`]] [[, …]] [[,` *RejestrN*`:`*Nazwa*`]]`
— pozwala kontrolować błędy wynikające z przypisania *Rejestrom* nieodpowiednich wartości; użycie `ASSUME` powoduje, że asembler będzie kontrolował wartości przypisane danym *Rejestrom* po czym, w przypadku wykrycia próby przypisania im niepoprawnej wartości, zgłosi błąd; istnieje możliwość całkowitego zabronienia użycia danych *Rejestrów* przy użyciu konstrukcji:

`ASSUME` *Rejestr*`:ERROR [[,` *Rejestr2*`:ERROR]] [[, …]] [[,` *RejestrN*`:ERROR]]`

a w celu wyłączenia kontroli wartości przypisywanej *Rejestrom* należy użyć następującego zapisu:

`ASSUME` *Rejestr*`:NOTHING [[,` *Rejestr2*`:NOTHING]] [[, …]] [[,` *RejestrN*`:NOTHING]].`

Typowe konstrukcje

Definicja symboli (stałych) w programie

Nazwa `=` *Wyrażenie*
— definicja symbolu *Nazwa*; definicja może być wielokrotnie zmieniana.

Nazwa `EQU` *Wyrażenie*
— definicja symbolu *Nazwa* i przypisanie mu wartości liczbowej *Wyrażenie*; definicja nie może być zmieniana.

Nazwa `EQU <`*Tekst*`>`
— definicja symbolu *Nazwa* i przypisanie mu wartości tekstowej *Tekst*; definicja może być zmieniana.

Nazwa TEXTEQU [[*Tekst*]]

— definicja symbolu *Nazwa* i przypisanie mu wartości *Tekst*; *Tekst* może być łańcuchem tekstowym, stałą poprzedzaną znakiem % lub łańcuchem tekstowym zwracanym przez makro.

Nazwa SIZESTR *Tekst*

— powoduje przypisanie symbolowi *Nazwa* wartości będącej długością zadanego *Tekstu*.

Definicja etykiety na poziomie programu

Etykieta:

— definiuje *Etykietę*; do *Etykiety* można odwoływać się w innych miejscach programu za pomocą instrukcji zmieniających bieg programu (rozkazy skoków, pętli, wywołania procedur).

Etykieta LABEL *typ*

— definiuje *Etykietę* o zadanym *Typie*; do *Etykiety* można odwoływać się w innych miejscach programu za pomocą instrukcji zmieniających bieg programu (rozkazy skoków, pętli, wywołania procedur).

Pętle

Pętla .REPEAT .UNTIL

```
.REPEAT
    Kod
.UNTIL Warunek
```

— powoduje wykonywanie zadanego *Kodu* po uruchomieniu programu tak długo, aż zostanie spełniony zadany *Warunek*.

Pętla .REPEAT .UNTILCXZ

```
.REPEAT
    Kod
.UNTIL [[Warunek]]
```

— powoduje wykonywanie zadanego *Kodu* po uruchomieniu programu tak długo, aż wartość rejestru CX przyjmie wartość 0 i ewentualnie zostanie spełniony dodatkowy *Warunek*.

Pętla .WHILE

```
.WHILE Warunek
    Kod
.ENDW
```

— powoduje wykonywanie zadanego *Kodu* po uruchomieniu programu tak długo, jak długo spełniony jest zadany *Warunek*.

Przerywanie pętli za pomocą konstrukcji .BREAK

```
.BREAK [[.IF Warunek]]
```

— przerywa wykonywanie pętli `.WHILE` lub `.REPEAT`; możliwe jest przerwanie po wystąpieniu określonego *Warunku*.

Kontynuacja pętli za pomocą konstrukcji .CONTINUE

```
.CONTINUE [[.IF Warunek]]
```

— kontynuuje wykonywanie pętli `.WHILE` lub `.REPEAT` od jej początku; możliwe jest wykonanie skoku po wystąpieniu określonego *Warunku*.

Wyrażenie warunkowe

```
.IF Warunek
   Wyrażenie
[[.ELSEIF Warunek2
   Wyrażenie2]]
…
[[.ELSEIF WarunekN
   WyrażenieN]]
[[.ELSE
   Wyrażenie]]
.ENDIF
gdzie:
.IF Warunek
   Wyrażenie
```

— oznacza początek konstrukcji — *Wyrażenie* zostanie wykonane po spełnieniu określonego *Warunku*.

```
[[.ELSEIF Warunek2
   Wyrażenie2]]
…
[[.ELSEIF WarunekN
   WyrażenieN]]
```

— oznacza kolejne bloki (*Wyrażenie2*, *Wyrażenie3*, …, *WyrażenieN*) wykonywane, jeśli spełnione są odpowiednie *Warunek2*, *Warunek3*, …, *WarunekN*.

```
[[.ELSE
   Wyrażenie]]
```

— określa *Wyrażenie* wykonywane tylko wtedy, jeśli żaden z powyższych warunków, tj. *Warunek*, *Warunek2*, …, *WarunekN* nie został spełniony.

```
.ENDIF
```

— kończy konstrukcję.

Operacje na łańcuchach

```
Nazwa CATSTR [[Tekst1 [[, Tekst2]] [[, …]]
```

— łączy kilka łańcuchów (operacja konkatenacji) w jedną całość; łączone łańcuchy mogą być stałymi poprzedzanymi znakiem `%` lub łańcuchami tekstowymi zwracanymi przez makra.

Nazwa INSTR [[*Pozycja*,]], *Tekst1*, *Tekst2*

— wyszukuje pierwsze wystąpienie łańcucha tekstowego *Tekst1* w łańcuchu *Tekst2*; istnieje możliwość podania *Pozycji*, od której będzie rozpoczynać się przeszukiwanie; łańcuchy mogą być stałymi poprzedzanymi znakiem % lub łańcuchami tekstowymi zwracanymi przez makra.

Deklaracja danych

Deklaracja danych w różnych sekcjach programu (głównie .DATA i .DATA?)

DB

— deklaracja danej o rozmiarze 8 bitów — dana typu BYTE.

DW

— deklaracja danej o rozmiarze 16 bitów — dana typu WORD.

DD

— deklaracja danej o rozmiarze 32 bitów — dana typu DWORD (inaczej DOUBLE WORD).

DF

— deklaracja danej o rozmiarze 48 bitów — dana typu FWORD (inaczej FAR WORD).

DQ

— deklaracja danej o rozmiarze 64 bitów — dana typu QWORD (inaczej QUAD WORD).

DT

— deklaracja danej o rozmiarze 80 bitów — dana typu TBYTE (inaczej TEN BYTE).

Definiowanie struktur

```
Nazwa STRUC
   DeklaracjaPól
Nazwa ENDS
```

lub

```
Nazwa STRUCT
   DeklaracjaPól
Nazwa ENDS
```

gdzie:

Nazwa STRUC lub *Nazwa* STRUCT

— określa początek definicji struktury o podanej *Nazwie*.

DeklaracjaPól

— definiuje części składowe struktury — pola o zadanym typie (DB, DW, inne struktury, itd.).

Nazwa ENDS

— określa koniec struktury o podanej *Nazwie*.

Definiowanie unii

```
Nazwa UNION
   DeklaracjaPól
Nazwa ENDS
```

gdzie:

Nazwa UNION
— określa początek definicji unii o podanej *Nazwie*.

DeklaracjaPól
— definiuje części składowe unii.

Nazwa ENDS
— określa koniec unii o podanej *Nazwie*; w przypadku zagnieżdżonych bloków UNION wystarczy pojedynczy zapis *Nazwa* ENDS.

Deklaracja nowych typów

Nazwa TYPEDEF *Typ*
— definiuje nowy typ o podanej *Nazwie*, który jest jednoznaczny z *Typem*.

Deklaracja zmiennych lokalnych w procedurach (funkcjach)

LOCAL *Zmienna*:BYTE
— deklaracja *Zmiennej* 8-bitowej.

LOCAL *Zmienna*:SBYTE
— deklaracja *Zmiennej* 8-bitowej ze znakiem.

LOCAL *Zmienna*:WORD
— deklaracja *Zmiennej* 16-bitowej.

LOCAL *Zmienna*:WORD
— deklaracja *Zmiennej* 16-bitowej ze znakiem.

LOCAL *Zmienna*:DWORD
— deklaracja *Zmiennej* 32-bitowej.

LOCAL *Zmienna*:SDWORD
— deklaracja *Zmiennej* 32-bitowej ze znakiem.

LOCAL *Zmienna*:FWORD
— deklaracja *Zmiennej* 48-bitowej.

LOCAL *Zmienna*:QWORD
— deklaracja *Zmiennej* 64-bitowej.

LOCAL *Zmienna*:TBYTE
— deklaracja *Zmiennej* 80-bitowej.

Inne operacje

`ALIGN [[Wartość]]`

— powoduje, że adres pierwszej (następnej po wystąpieniu konstrukcji `ALIGN`) zadeklarowanej zmiennej będzie wyrównany do wielokrotności *Wartości*.

`EVEN`

— powoduje, że adres pierwszej (następnej po wystąpieniu konstrukcji `EVEN`) zadeklarowanej zmiennej lub użytej instrukcji będzie wyrównany do adresu parzystego.

Procedury i funkcje

Deklaracja procedury (funkcji) — PROC i ENDP

```
Nazwa PROC [[Dist]] [[Język]] [[Zakres]] [[Par1]] [[, Par2]] [[, …]] [[, ParN]]
[[USES ListaRejestrów]] [[, Parametr [[:Znacznik]] ]] …
  Ciało
Nazwa ENDP
```

gdzie:

`Nazwa PROC [[Par1]] [[, Par2]] [[, …]] [[, ParN]] [[USES ListaRejestrów]]`

— rozpoczyna definicję procedury o podanej *Nazwie*; procedura może być wywoływana z parametrami (*Par1*, *Par2*, …, *ParN*); procedura może być wywoływana w programie poleceniem `INVOKE` *nazwa* lub `CALL` *nazwa*; można określić, które rejestry są niszczone przez procedurę, tak że assembler automatycznie zadba o zachowanie ich w prologu i przywrócenie w epilogu procedury (`USES` *ListaRejestrów*).

Ciało

— kod procedury (z reguły kończący się rozkazem `RET`).

Nazwa `ENDP`

— kończy definicję procedury o zadanej *Nazwie*.

Wywołanie procedury (funkcji) — INVOKE

`INVOKE Nazwa [[Par1]] [[, Par2]] [[, …]] [[, ParN]]`

— powoduje wywołanie procedury o podanej *Nazwie*; istnieje możliwość przekazywania parametrów *Par1*, *Par2*, …, *ParN* do procedury (funkcji); parametry te zostaną umieszczone na stosie lub w rejestrach w kolejności wynikającej z przyjętego modelu wywoływania procedur; parametrami mogą być rejestry, adresy pamięci i wyrażenia; asembler jest w stanie sprawdzić, czy liczba parametrów i ich typ zgadza się z parametrami zadeklarowanymi w ciele procedury.

Deklaracja prototypu procedury (funkcji) — PROTO

Nazwa PROTO [[, *Parametr*]] …

— prototyp procedury (funkcji); informuje asembler o istnieniu procedury (funkcji) o zadanej *Nazwie* i atrybutach.

Tworzenie własnych makr

Konstrukcja MACRO

```
MACRO Nazwa [[Parametr1]] [[, Parametr2]] [[, …]] [[, ParametrN]]
   KodMakra
ENDM
```

gdzie:

MACRO *Nazwa* [[*Parametr1*]] [[, *Parametr2*]] [[, …]] [[, *ParametrN*]]

— rozpoczyna definicję makra o podanej *Nazwie*; makro może być wywoływane z parametrami (*Parametr1*, *Parametr2*, …, *ParametrN*); w miejscu wywołania makra zostanie ono rozwinięte zgodnie z *KodemMakra*.

ENDM

— kończy definicję makra lub blok REPEAT.

Konstrukcja EXITM

EXITM [[*Wartość*]]

— powoduje zakończenie przetwarzania bieżącego bloku (rozpoczętego konstrukcją MACRO lub REPEAT); przetwarzanie rozpocznie się od następnej instrukcji po zakończonym właśnie bloku; istnieje możliwość zwrócenia zadanej *Wartości* do wyrażenia, które spowodowało wywołanie makra.