

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Asembler. Podręcznik programisty

Autor: Vlad Pirogow

Tłumaczenie: Wojciech Moch

ISBN: 83-7361-797-3

Tytuł oryginału: [The Assembly Programming Master Book](#)

Format: B5, stron: 640



Tajniki tworzenia programów dla systemu Windows w asemblerze

- Poznaj narzędzia programistyczne
- Napisz programy wykorzystujące Windows API
- Wykryj i usuń błędy z programów asemblerowych

Pisanie programów w asemblerze przez długi czas kojarzyło się z systemem MS-DOS. Teraz asembler powoli odzyskuje straconą pozycję języka programowania dla systemu Windows. Wśród wielu zalet asemblera można wymienić: bezpośredni dostęp do procesora, zwarty i niewielki kod wynikowy oraz możliwości trudne do uzyskania za pomocą języków wysokiego poziomu. Asembler jest niezbędny przy tworzeniu sterowników dla urządzeń zewnętrznych, a korzystanie z niego uczy myślenia algorytmicznego, od którego języki obiektowe stopniowo odzwyczajają programistów.

Książka „Asembler. Podręcznik programisty” to kompendium wiedzy dotyczącej pisania programów dla systemu Windows w języku asemblera. Znajdziesz w niej opisy narzędzi programistycznych i sposoby korzystania z procedur Windows API. Nauczysz się tworzyć programy dla Windows – od najprostszych, wykorzystujących konsolę, aż do skomplikowanych aplikacji z interfejsem graficznym. Dowiesz się, jak korzystać z narzędzi do wykrywania i usuwania błędów, a także poznasz zasady stosowania asemblera w połączeniu z językami wysokiego poziomu.

- Narzędzia programistyczne dla systemu Windows
- Wywoływanie funkcji Windows API
- Programy działające w trybie tekstowym
- Tworzenie i wykorzystywanie zasobów
- Praca z systemem plików
- Tworzenie bibliotek DLL
- Programowanie sieciowe
- Wykorzystywanie asemblera w językach wysokiego poziomu
- Turbo Debugger
- Tworzenie sterowników

Odkryj nieznane obszary programowania – poznaj język asemblera

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



Spis treści

Wstęp	9
Część I Podstawy programowania 32-bitowego w systemach Windows	13
Rozdział 1. Narzędzia programistyczne dla systemu Windows	15
Pierwszy program w assemblerze i jego przekształcenia	15
Moduły obiektowe	20
Dyrektywa INVOKE	22
Dane w module obiekowym	24
Asemblacja programem TASM	25
Uproszczony tryb segmentacji	26
Inne narzędzia używane do pracy z assemblerem	27
Edytory	28
Programy uruchomieniowe	28
Deasemblerzy	29
Edytory szesnastkowe	31
Kompilatory zasobów	31
Edytory zasobów	31
Rozdział 2. Podstawy programowania w systemie Windows	33
Wywoływanie funkcji API	34
Struktura programu	37
Rejestracja klasy okna	37
Tworzenie okna	37
Pętla obsługi komunikatów	37
Procedura okna głównego	38
Przykłady prostych programów okienkowych	39
Jak to zrobić w assemblerze TASM32?	45
Przekazywanie parametrów poprzez stos	49
Rozdział 3. Proste programy w języku assemblera	53
Zasady budowania aplikacji w systemie Windows	53
Okno z przyciskiem	55
Okno z polem edycyjnym	59
Okno z listą	65
Okna potomne i okna posiadane	72

Rozdział 4. Przegląd programowania 16-bitowego	81
Zasady programowania 16-bitowego w systemie Windows	81
Przykład aplikacji 16-bitowej	83
Rozdział 5. Asemblery MASM i TASM	89
Opcje wiersza poleceń assemblerów ml.exe i tasm32.exe	89
Opcje wiersza poleceń konsolidatorów link.exe i tlink32.exe	92
Dołączanie do plików wykonywalnych informacji dla programu uruchomieniowego ...	97
Tworzenie aplikacji konsolowych i aplikacji z interfejsem GUI	99
Konsolidacja automatyczna	99
Programy same tłumaczące się na kod wynikowy	99
Część II Programowanie w systemie Windows	101
Rozdział 6. Kodowanie tekstu w systemie Windows	103
Kodowanie informacji tekstowych	103
OEM i ANSI	104
Unikod	105
Rozdział 7. Przykłady prostych programów	109
Wypisywanie tekstu w oknie	109
Wybieranie czcionki	122
Grafika	128
Rozdział 8. Aplikacje konsolowe	139
Tworzenie konsoli	143
Przetwarzanie zdarzeń klawiatury i myszy	147
Czasomierz w aplikacji konsolowej	154
Rozdział 9. Idea zasobów — edytory i kompilatory zasobów	161
Język opisu zasobów	161
Ikony	162
Kursory	164
Bitmapy	165
Ciągi znaków	165
Dialogi	165
Menu	170
Klawisze skrótów	175
Niemodalne okna dialogowe	177
Kompilowanie zasobów i konsolidowanie plików obiektowych w pakiecie TASM32	182
Rozdział 10. Przykłady programów korzystających z zasobów	185
Dynamiczne menu	185
Klawisze szybkiego dostępu	195
Zarządzanie listami	201
Programowanie w stylu Windows XP	207
Rozdział 11. Praca z plikami	213
Charakterystyki plików	213
Atrybuty plików	214
System plików FAT32	217
System plików NTFS	220
Katalogi w systemie plików NTFS	223
Kompresja plików w systemie NTFS	224

Punkty dołączania	224
Wyszukiwanie plików	225
Asemlacja programu za pomocą asemlera TASM	240
Techniki pracy z plikami binarnymi	240
Jak pobrać atrybuty pliku?	251

Część III Bardziej skomplikowane przykłady programów dla systemu Windows255

Rozdział 12. Dyrektywy i makroinstrukcje języka asemlera 257

Etykiety	257
Struktury	259
Unie	260
Wygodna metoda pracy ze strukturami	260
Asemlacja warunkowa	261
Wywołania procedur	262
Makroinstrukcje powtórzeń	263
Makroinstrukcja definiowana	264
Kilka innych dyrektyw i operatorów asemlera	266
Konstrukcje typu HLL	267
Konstrukcje warunkowe	267
Pętla WHILE	268
Tworzenie programów asemlowanych zarówno przez asemler MASM, jak i asemler TASM	269

Rozdział 13. Więcej o zarządzaniu plikami 271

Dokładny opis funkcji CreateFile	271
Inne możliwości funkcji CreateFile	275
Skrytki pocztowe	276
Potoki	281
Napędy dyskowe	282
Przegląd funkcji API zajmujących się zarządzaniem plikami	286
Asynchroniczne wejście i wyjście	287

Rozdział 14. Przykłady programów korzystających z czasomierza 293

Najprostszy przykład użycia czasomierza	295
Interakcja między czasomierzami	299
Okna dymków pomocy	305

Rozdział 15. Wielozadaniowość 317

Tworzenie procesu	317
Wątki	327
Komunikacja między wątkami	333
Semafory	334
Zdarzenia	336
Sekcje krytyczne	336
Wzajemne wykluczenie	346

Rozdział 16. Tworzenie bibliotek dynamicznych (DLL) 347

Koncepcje ogólne	347
Tworzenie biblioteki dynamicznej	349
Konsolidacja niejawna	354
Korzystanie ze wspólnej przestrzeni adresowej	356
Współdzielenie pamięci przez kilka procesów	364

Rozdział 17. Programowanie sieciowe	369
Urządzenia sieciowe	369
Wyszukiwanie i podłączanie dysków sieciowych	374
Słowo o protokołach sieciowych TCP/IP	387
Model OSI	387
Rodzina protokołów TCP/IP	387
Adresowanie w protokole IP	389
Maski adresów	391
Adresy fizyczne i adresy protokołu IP	391
Usługa systemu nazw domen	391
Automatyczne przypisywanie adresów protokołu IP	392
Routing	392
Zarządzanie gniazdami	393
Przykład aplikacji najprostszego serwera i klienta	397
Rozdział 18. Rozwiązywanie problemów związanych z programowaniem w systemie Windows	411
Umieszczanie ikony na tacce systemowej	411
Przetwarzanie plików	417
Kontrolowanie danych w polu edycyjnym	419
Wymiana danych pomiędzy aplikacjami	427
Zabezpieczenie przed wielokrotnym uruchomieniem aplikacji	433
Operacje na grupach plików i katalogów	434
Drukowanie	436
Korzystanie z listy zadań	436
Część IV Debugowanie, analiza kodu i przygotowywanie sterowników	443
Rozdział 19. Programowanie systemowe	445
Adresowanie stron i segmentów	445
Przestrzeń adresowa procesu	450
Zarządzanie pamięcią	452
Haki	458
Rozdział 20. Wykorzystywanie asemblera w językach wysokiego poziomu	467
Koordynacja wywołań	467
Koordynacja nazw	468
Koordynacja parametrów	469
Prosty przykład wykorzystania asemblera w językach wysokiego poziomu	470
Borland C++ 5.0	470
Visual C++ 7.0	471
Delphi 7.0	473
Przekazywanie parametrów przez rejestry	474
Wywołania funkcji API i zasoby w modułach asemblera	475
Połączenie języka C z kodem asemblerowym	480
Assembler wbudowany	486
Przykład wykorzystania biblioteki dynamicznej	488
Rozdział 21. Programowanie usług	493
Podstawowe koncepcje i funkcje kontrolne	493
Struktura usługi	495
Przykładowa usługa	501

Rozdział 22. Programy uruchomieniowe i deasemblerzy	515
Narzędzia firmy Microsoft	515
editbin.exe	515
dumpbin.exe	517
Narzędzia innych producentów	519
dumppe.exe	519
hiew.exe	519
dewin.exe	522
IDA Pro	522
Rozdział 23. Wprowadzenie do programu Turbo Debugger	525
Debugowanie programów napisanych w językach wysokiego poziomu	529
Technika debugowania	531
Rozdział 24. Praca z deasemblerem W32Dasm i programem uruchomieniowym Softlce	533
Program uruchomieniowy W32Dasm	533
Rozpoczęcie pracy	533
Nawigowanie po deasemblowanym kodzie	535
Wyświetlanie danych	536
Wypisywanie importowanych i eksportowanych funkcji	536
Wyświetlanie zasobów	537
Operacje na tekstach	537
Ładowanie programów do debugowania	538
Praca z bibliotekami dynamicznymi	539
Punkty wstrzymania	539
Modyfikowanie kodu, danych i rejestrów	540
Dodatkowe możliwości pracy z funkcjami API	541
Wyszukiwanie w programie potrzebnych lokalizacji	541
Debugger Softlce	542
Instalacja	543
Ładowanie programu do debugowania	543
Przegląd poleceń programu uruchomieniowego	544
Rozdział 25. Podstawy analizy kodu	549
Zmienne i stałe	549
Struktury sterujące języka C	553
Konstrukcje warunkowe	553
Zagnieżdżone konstrukcje warunkowe	554
Operator Switch	555
Pętle	556
Zmienne lokalne	557
Funkcje i procedury	559
Optymalizacja kodu	560
Prędkość albo rozmiar	562
Optymalizowanie skoków warunkowych	563
Optymalizowanie wywołań procedur	563
Programowanie zorientowane obiektowo	564
Rozdział 26. Korygowanie modułów wykonywalnych	569
Praktyczny przykład korygowania modułu wykonywalnego	569
Wyszukiwanie procedury okna	572

Rozdział 27. Struktura sterowników i ich tworzenie	575
Wirtualne sterowniki urządzeń	575
Opis projektu	577
Prosty sterownik	581
Dynamiczne sterowniki wirtualne	583
Podstawowe pojęcia sterowników trybu jądra	589
Jądro systemu operacyjnego i struktura pamięci	590
Kontrolowanie sterowników	591
Przykład prostego sterownika trybu jądra	592
Sterowniki i urządzenia trybu jądra	605
 Dodatki	 615
Bibliografia	617
Skorowidz	619

Rozdział 1.

Narzędzia programistyczne dla systemu Windows

W niniejszym rozdziale przedstawię krótkie wprowadzenie do kwestii związanych z narzędziami wykorzystywanymi w czasie programowania w asemblerze. Rozdział ten przeznaczony jest dla początkujących, dlatego doświadczeni programiści mogą go z czystym sumieniem pominąć.

Na początek trzeba zauważyć, że tytuł tego rozdziału jest nieco zwodniczy, ponieważ techniki asemblacji w systemie MS-DOS i Windows są do siebie bardzo podobne. Mimo to programowanie w systemie MS-DOS jest już właściwie odległą przeszłością.

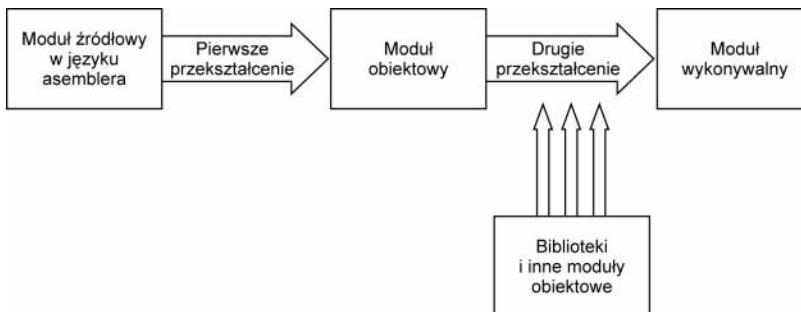
Pierwszy program w asemblerze i jego przekształcenia

Na rysunku 1.1 przedstawiono schemat przekształceń pojedynczego modułu w języku asemblera.

Z etapami przekształceń przedstawionymi na rysunku 1.1 związane są dwa specjalne programy: asembler¹ *ml.exe* i konsolidator *link.exe* (a jeżeli korzystamy z pakietu Turbo Assembler, to odpowiednio programy *tasm32.exe* i *tlink32.exe*). Założmy, że plik źródłowy z programem w języku asemblera nazywa się *prog.asm*. Bez zagłębiania się

¹ Programiści praktycznie od zawsze nazywali programy tłumaczące kod w języku asemblera *assemblerami*, a nie *kompilatorami*.

Rysunek 1.1.
Schemat
przekształceń
modułu
assemblerowego



w szczególności można powiedzieć, że pierwszy etap przekształceń, który nazywać będziemy asemblacją lub po prostu tłumaczeniem, wykonywany jest poniższym poleceniem:

```
c:\masm23\bin\ml /c /coff prog.asm
```

W wyniku działania tego polecenia powstanie plik obiektowy *prog.obj*. W drugim etapie przekształceń nazywanym konsolidacją lub łączeniem, wykorzystamy go w poleceniu:

```
c:\masm32\bin\link /subsystem:windows prog.obj
```

W wyniku jego działania otrzymamy plik wykonywalny *prog.exe*. Myślę, że nietrudno domyślić się znaczenia opcji */c* oraz */coff* programu *ml.exe* i opcji */subsystem:windows* programu *link.exe*.

Pozostałe opcje tych programów ze szczegółami opisane będą w rozdziale 5.

Im więcej zastanawiam się nad tym dwuetapowym schematem przekształceń, tym bardziej wydaje mi się on doskonały. Format wynikowego modułu jest zależny od systemu operacyjnego. Po określeniu wymagań struktury modułu obiektowego uzyskujemy następujące możliwości:

- ♦ zastosowanie gotowych do użycia modułów obiektowych,
- ♦ konsolidacja programów napisanych za pomocą kilku różnych języków programowania.

Jednak główną zaletą jest możliwość stosowania standardu modułów obiektowych dla innych systemów operacyjnych. Oznacza to, że będziemy mogli wykorzystać też moduły przygotowane w innych systemach operacyjnych².

W celu zobrazowania procesu asemblacji i konsolidacji kodu źródłowego zaprezentuję teraz kilka programów, które tak naprawdę nie robią nic użytecznego.

Na listingu 1.1 przedstawiony został program „Nie rób nic”. Zapiszemy go w pliku *prog1.asm*. Zaznaczam teraz, że we wszystkich przykładowych programach instrukcje mikroprocesora i dyrektywy assemblera pisane będą WIELKIMI LITERAMI.

² Tego rodzaju przenośność ma jednak pewne ograniczenia, ponieważ spore trudności może sprawiać koordynacja wywołań systemowych w poszczególnych systemach operacyjnych.

Listing 1.1. Program „Nie rób nic”

```
.586P
; Płaski model pamięci
.MODEL FLAT, stdcall
;-----
; Segment danych
_DATA SEGMENT
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    RET                                ; Wyjście
_TEXT ENDS
END START
```

Teraz, wykonując poniższe polecenia, przekształcimy nasz program w moduł wykonywalny³:

```
m1 /c /coff prog1.asm
link /subsystem:windows prog1.obj
```

Korzystając jednak z pakietu Turbo Assemblera, należy zastosować poniższe polecenia:

```
tasm32 /m1 prog1.asm
tlink32 -aa prog1.obj
```

Na razie proszę nie zagłębiać się w zbytnio w przedstawione przykłady przekształceń.

Bardzo często wygodnym rozwiązaniem jest podzielenie kodu źródłowego na kilka części i łączenie ich w pierwszym etapie przekształceń. Na taki zapis kodu źródłowego programów pozwala dyrektywa `INCLUDE`. Na przykład w jednym z plików można umieścić kod programu, a w drugim — stałe i dane (takie jak definicje zmiennych) wraz z prototypami procedur zewnętrznych. Takie pliki bardzo często opatrywane są rozszerzeniem `.inc`.

Ta metoda podziału programu przedstawiona została na listingu 1.2.

Listing 1.2. Zastosowanie dyrektywy `INCLUDE`

```
; Plik CONS.INC
CONS1 EQU 1000
CONS2 EQU 2000
CONS3 EQU 3000
CONS4 EQU 4000
CONS5 EQU 5000
CONS6 EQU 6000
CONS7 EQU 7000
CONS8 EQU 8000
CONS9 EQU 9000
CONS10 EQU 10000
```

³ Jeżeli nazwy asemblowanych i konsolidowanych modułów zawierają w sobie spację, to muszą być one zamknięte pomiędzy znakami cudzysłowu:

```
m1 /c /coff "mój pierwszy program.asm"
```

```

CONS11 EQU 11000
CONS12 EQU 12000

; Plik DAT.INC
DAT1 DWORD 0
DAT2 DWORD 0
DAT3 DWORD 0
DAT4 DWORD 0
DAT5 DWORD 0
DAT6 DWORD 0
DAT7 DWORD 0
DAT8 DWORD 0
DAT9 DWORD 0
DAT10 DWORD 0
DAT11 DWORD 0
DAT12 DWORD 0
; Plik progl.asm
.586P
; Plaski model pamięci
.MODEL FLAT, stdcall
; Dolączenie pliku ze stałymi
INCLUDE CONS.INC
;-----
; Segment danych
_DATA SEGMENT
; Dolączenie pliku z danymi
INCLUDE DAT.INC
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    MOV EAX, CONS1
    SHL EAX, 1           ; Mnożenie przez 2
    MOV DAT1, EAX
;-----
    MOV EAX, CONS2
    SHL EAX, 2           ; Mnożenie przez 4
    MOV DAT2, EAX
;-----
    MOV EAX, CONS3
    ADD EAX, 1000         ; Dodanie wartości 1000
    MOV DAT3, EAX
;-----
    MOV EAX, CONS4
    ADD EAX, 2000         ; Dodanie wartości 2000
    MOV DAT4, EAX
;-----
    MOV EAX, CONS5
    SUB EAX, 3000         ; Odjęcie wartości 3000
    MOV DAT5, EAX
;-----
    MOV EAX, CONS6
    SUB EAX, 4000         ; Odjęcie wartości 4000
    MOV DAT6, EAX
;-----
    MOV EAX, CONS7
    MOV EDX, 3

```

```
IMUL EDX                ; Mnożenie przez 3
MOV DAT7, EAX
;-----
MOV EAX, CONS8
MOV EDX, 7                ; Mnożenie przez 7
IMUL EDX
MOV DAT8, EAX
;-----
MOV EAX, CONS9
MOV EBX, 3                ; Dzielenie przez 3
MOV EDX, 0
IDIV EBX
MOV DAT9, EAX
;-----
MOV EAX, CONS10
MOV EBX, 7                ; Dzielenie przez 7
MOV EDX, 0
IDIV EBX
MOV DAT10, EAX
;-----
MOV EAX, CONS11
SHR EAX, 1                ; Dzielenie przez 2
MOV DAT11, EAX
;-----
MOV EAX, CONS12
SHR EAX, 2                ; Dzielenie przez 4
MOV DAT12, EAX
;-----
RET                        ; Wyjście
TEXT ENDS
END START
```

Podobnie jak wszystkie przykładowe programy z tego rozdziału program przedstawiony na listingu 1.2 nie ma większego sensu. Doskonale demonstrowuje jednak możliwości, jakie udostępnia nam dyrektywa `INCLUDE`. Ponownie proszę o nieskupianie się na wszystkich instrukcjach mikroprocesora przedstawianych w przykładach. Na razie interesować nas będzie tylko instrukcja `IDIV`.

W naszym programie instrukcja `IDIV` wykonuje operację dzielenia parametru umieszczonego w parze rejestrów `EDX:EAX`. Wpisując do rejestru `EDX` zero, powodujemy, że dzielona będzie tylko liczba zapisana w rejestrze `EAX`.

Aseblacja i konsolidacja programu wykonywane jest programami `MASM` lub `TASM` zgodnie z podanymi wcześniej wskazówkami.



Typy danych

W niniejszej książce najczęściej będziemy korzystać z trzech głównych typów danych: bajtu (*byte*), słowa (*word*) i podwójnego słowa (*double word*). Powszechnie stosowana jest następująca metoda zapisu tych typów: bajt — `BYTE` lub `DB`, słowo — `WORD` lub `DW`, podwójne słowo — `DWORD` lub `DD`. Wybór jednej z metod zapisu (w jednym miejscu piszę `DB`, a w innym `BYTE`) dyktowany był chęcią uwypuklenia pewnych funkcji języka i zróżnicowania zapisu.

Moduły obiektowe

Wyjaśnię teraz, dlaczego w etapie konsolidacji konieczne jest dołączanie innych modułów obiektowych i bibliotek. Po pierwsze, trzeba wspomnieć, że niezależnie od liczby łączonych ze sobą modułów, tylko jeden z nich może być modulem głównym. Wynika to z bardzo prostej zasady mówiącej, że modulem głównym jest ten, od którego rozpoczyna się wykonywanie programu. Jest to jedyna różnica pomiędzy modulem głównym a pozostałymi. Trzeba też pamiętać o tym, że moduł główny w punkcie startowym segmentu musi mieć zdefiniowaną etykietę `START`. Należy wypisać ją również po dyrektywie `END`, ponieważ w ten sposób informujemy assembler, żeby wpisał dane punktu wejścia programu do nagłówka ładowanego modułu.

Z reguły wszystkie procedury wywoływane w danym module umieszczane są w modułach dołączanych dyrektywą `INCLUDE`. Przyjrzyjmy się takiemu modułowi przedstawionemu na listingu 1.3.

Listing 1.3. *Moduł `prog2.asm` przechowujący procedurę `proc1` wywoływaną z modułu głównego*

```
.586P
; Moduł prog2
; Płaski model pamięci
.MODEL FLAT, stdcall
PUBLIC PROC1
_TEXT SEGMENT
PROC1 PROC
    MOV EAX, 1000
    RET
PROC1 ENDP
_TEXT ENDS
END
```

Przed wszystkim proszę zauważyć, że po dyrektywie `END` nie ma żadnej etykiety. Jak widać, z pewnością nie jest to moduł główny, ale zawarte w nim procedury będą wywoływane z innych modułów.

Innym ważnym elementem, na który chciałbym zwrócić uwagę, jest to, że procedura w tym module musi zostać zadeklarowana ze słowem kluczowym `PUBLIC`. Nazwa tej procedury zostanie zapisana w module obiektowym, dzięki czemu będzie można ją łączyć z wywołaniami z innych modułów.

Możemy więc uruchomić następujące polecenie:

```
ml /coff /c prog2.asm
```

W wyniku działania tego polecenia powstanie moduł *prog2.obj*.

Przeprowadzimy teraz małe śledztwo. Proszę przejrzeć zawartość pliku obiektowego za pomocą najprostszej przeglądarki, takiej jak wbudowana w menedżer plików *Far.exe*. Zapewne bardzo szybko zauważymy, że zamiast nazwy `PROC1` w pliku tym zapisana jest nazwa `_PROC1@0`. Znaki, o których teraz będzie mowa, mają bardzo duże znaczenie, dlatego należy dobrze zapamiętać te informacje! Po pierwsze, znajdujący się na

początku znak podkreślenia () oznacza, że stosowany jest standard ANSI, który wymaga, aby wszystkie nazwy publiczne (w tym i nazwy udostępniane innym modułom) były automatycznie uzupełniane o znak podkreślenia. W tym przypadku, zajął się tym za nas program asemblera.

Przyrostek @0 jest już nieco bardziej złożony. Przede wszystkim musimy wiedzieć, co oznacza ta kombinacja znaków. Liczba podana za znakiem @ oznacza liczbę bajtów, jakie należy odłożyć na stos przed wywołaniem procedury. W tym przypadku asembler stwierdził, że procedura nie wymaga podawania żadnych parametrów. Taki zapis wprowadzony został w ramach opisywanej dalej dyrektywy INVOKE stosowanej do wygodnego wywoływania procedur. Teraz spróbujemy przygotować główny moduł programu o nazwie *prog1.asm* — listing 1.4.

Listing 1.4. Moduł *prog1.asm* wywołujący procedurę z modułu *prog2.asm*

```
.586P
; Płaski model pamięci
.MODEL FLAT, stdcall
;-----
; Prototyp procedury zewnętrznej
EXTERN PROC1@0:NEAR
; Segment danych
_DATA SEGMENT
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    CALL PROC1@0
    RET                                ; Wyjście
_TEXT ENDS
END START
```

Jak widać, procedura wywoływana z innego modułu zadeklarowana jest z dyrektywą EXTERN. Co więcej, nazwa tej procedury musi być uzupełniona o przyrostek opisujący wielkość parametrów procedury, czyli nie można podać nazwy PROC1, ale nazwę PROC1@0. Na razie nic nie można w tym zmienić. Mogą pojawić się natomiast pytania o parametr NEAR. W systemie MS-DOS parametr ten oznaczał, że wywołanie procedury (lub skok bezwarunkowy) będzie odbywało się wewnątrz jednego segmentu. Z drugiej strony parametr FAR oznaczał, że wywołanie procedury (lub skok) będzie wykonywane z innego segmentu. W systemie Windows stosowany jest tak zwany płaski (*Flat*) model pamięci, w którym cała pamięć traktowana jest jako jeden wielki segment, dlatego naturalnym wydaje się zastosowanie parametru NEAR.

Możemy teraz wykonać następujące polecenie:

```
ml /coff /c prog1.asm
```

W wyniku otrzymamy moduł obiektowy *prog1.obj*. Połączmy więc dwa przygotowane moduły, tworząc końcowy program wykonywalny o nazwie *prog1.exe*:

```
link /subsystem:windows prog1.obj prog2.obj
```

W czasie łączenia modułów jako pierwsza musi być podana nazwa modułu głównego, a nazwy pozostałych modułów można podawać za nią w dowolnej kolejności.

Dyrektywa INVOKE

Przyjrzyjmy się teraz dyrektywie INVOKE. Jest to bardzo wygodne polecenie, jednak z powodów, o których powiem później, osobiście korzystam z niego niezwykle rzadko.

Główna zaleta dyrektywy INVOKE polega na tym, że pozwala ona pominąć z nazw procedur przyrostek @N. Po drugie, dyrektywa sama zajmuje się załadowaniem odpowiednich parametrów na stos przed wywołaniem procedury. Dzięki temu nie trzeba stosować poniższej sekwencji poleceń:

```
PUSH par1
PUSH par2
PUSH par3
PUSH par4
CALL NAZWA_PROCEDURY@N      ; N — liczba bajtów do zapisania na stos
```

Zamiast nich wystarczy wpisać jedną dyrektywę:

```
INVOKE NAZWA_PROCEDURY, par4, par3, par2, par1
```

Jako parametry można w niej podawać rejestry, wartości bezpośrednie lub adresy. Podając adres, można zastosować zarówno operator OFFSET, jak i ADDR.

Zmodyfikujmy teraz moduł *prog1.asm* (modułu *prog2.asm* nie trzeba modyfikować) tak, jak pokazano na listingu 1.5.

Listing 1.5. Stosowanie dyrektywy INVOKE

```
.586P
; Płaski model pamięci
.MODEL FLAT, stdcall
;-----
; Prototyp procedury zewnętrznej
PROC1 PROTO
; Segment danych
_DATA SEGMENT
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    INVOKE PROC1
    RET      ; Wyjście
_TEXT ENDS
END START
```

Doskonale widać, że tym razem zewnętrzna procedura deklarowana jest za pomocą dyrektywy PROTO. Dyrektywa ta pozwala na łatwe deklarowanie ewentualnych parametrów procedury. Na przykład zapis podobny do poniższego:

```
PROC1 PROTO :DWORD, :WORD
```

oznacza, że procedura wymaga podania dwóch parametrów, z których pierwszy ma długość czterech bajtów, a drugi dwóch (co daje w sumie sześć bajtów i może być oznaczone przyrostkiem @6).

Jak już mówiłem, z dyrektywy `INVOKE` korzystam tylko w wyjątkowych przypadkach. Podam teraz pierwszy z powodów takiego unikania jej stosowania — jestem zdecydowanym zwolennikiem czystości języka assemblerowego, w związku z czym nie najlepiej czuję się, stosując w swoich programach jakiegokolwiek makroinstrukcje. Uważam też, że początkujący programiści nie powinni zbyt często używać makroinstrukcji w swoich programach, ponieważ w ten sposób nigdy nie będą mogli w pełni docenić piękna tego języka. Jest jeszcze jeden powód mojego unikania makroinstrukcji, ale o nim opowiem nieco później.

Według schematu przedstawionego na rysunku 1.1 możliwa jest konsolidacja ze sobą również innych modułów obiektowych i bibliotek. Jeżeli będziemy zmuszeni skorzystać z kilku modułów obiektowych, może to być bardzo niewygodne. Z tego powodu takie moduły łączone są w biblioteki. Najprostszą i najwygodniejszą metodą dołączenia do programu danej biblioteki w assemblerze MASM jest wykorzystanie dyrektywy `INCLUDELIB`.

Dyrektywa ta zapisana zostanie w kodzie obiektowym i będzie później wykorzystywana przez program *link.exe*.

Jak jednak tworzy się biblioteki z modułów? Do tego zadania użyć należy specjalnego programu nazywanego bibliotekarzem (ang. *librarian*). Założmy, że chcielibyśmy przygotować bibliotekę *lib1.lib* składającą się z tylko jednego modułu — *prog2.obj*. W tym celu należy wywołać następujące polecenie:

```
lib /out:lib1.lib prog2.obj
```

Jeżeli w którymś momencie będziemy chcieli dodać do biblioteki dodatkowy moduł (*modul.obj*), to wystarczy wywołać poniższe polecenie:

```
lib lib1.lib modul.obj
```

Podam jeszcze dwa przykłady, w których można zastosować program bibliotekarza:

- ♦ `lib /list lib1.lib` — polecenie to wypisuje moduły zgromadzone w bibliotece.
- ♦ `lib /remove:modul.obj lib1.lib` — usuwa z biblioteki *lib1.lib* moduł *modul.obj*.

Wróćmy teraz do naszego przykładu. Tym razem zamiast modułu obiektowego użyjemy w programie przygotowanej przed chwilą biblioteki. Na listingu 1.6 pokazano zmodyfikowaną treść programu *prog1.asm*.

Listing 1.6. *Wykorzystanie biblioteki*

```
.586P
; Płaski model pamięci
.MODEL FLAT, stdcall
;-----
; Prototyp procedury zewnętrznej
EXTERN    PROC1@0:NEAR
;-----
INCLUDELIB LIB1.LIB
```

```

;-----
; Segment danych
_DATA SEGMENT
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    CALL PROC1@0
    RET                                ; Wyjście
_TEXT ENDS
END START

```

Dane w module obiektowym

Nadszedł czas, żeby przyrzeć się sposobom wykorzystania danych (zmiennych) zdefiniowanych w innym module obiektowym. Po dokładnym przeczytaniu tego podrozdziału nic nie powinno stanowić już w tym zakresie tajemnicy. Na listingach 1.7 i 1.8 przedstawione zostały moduły *prog1.asm* i *prog2.asm* będące demonstracją wykorzystania zmiennych zewnętrznych⁴.

Listing 1.7. Moduł przechowujący zmienną *ALT* wykorzystywaną w module *prog1.asm*

```

.586P
; Moduł prog2.asm
; Płaski model pamięci
.MODEL FLAT, stdcall
PUBLIC PROC1
PUBLIC ALT
; Segment danych
_DATA SEGMENT
    ALT DWORD 0
_DATA ENDS
_TEXT SEGMENT
PROC1 PROC
    MOV EAX, ALT
    ADD EAX, 10
    RET
PROC1 ENDP
_TEXT ENDS
END

```

Listing 1.8. Moduł wykorzystujący zmienną *ALT* zdefiniowaną w module *prog2.asm*

```

.586P
; Moduł prog1.asm
; Płaski model pamięci
.MODEL FLAT, stdcall
;-----

```

⁴ Termin „zmienna zewnętrzna” stosowany będzie tu na zasadzie analogii do terminu „procedura zewnętrzna”.

```

; Prototyp procedury zewnętrznej
EXTERN PROC1@0:NEAR
; Zmienna zewnętrzna
EXTERN ALT:DWORD
; Segment danych
_DATA SEGMENT
_DATA ENDS
; Segment kodu
_TEXT SEGMENT
START:
    MOV ALT, 10
    CALL PROC1@0
    MOV EAX, ALT
    RET
; Wyjście
_TEXT ENDS
END START

```

Proszę zauważyć, że w przeciwieństwie do procedur zewnętrznych zmienne zewnętrzne nie wymagają stosowania przyrostka @N, ponieważ ich wielkość jest znana z góry.

Asemlacja programem TASM

Spróbujemy teraz przetestować działanie wszystkich zaprezentowanych do tej pory programów, asemlując je tym razem programem *tasm*.

Jeżeli chodzi o programy przedstawione na listingach 1.1 i 1.2 to nie powinniśmy mieć żadnych problemów. W celu ich asemlacji i konsolidacji wystarczy wykonać poniższe polecenia:

```

tasm32 /ml prog1.asm
tlink32 -aa prog1.obj

```

Spróbujmy teraz przekształcić moduły *prog2.asm* i *prog1.asm*, które przedstawiane były na listingach 1.3 i 1.4. Utworzenie modułów obiektów nie powinno nastręczać żadnych problemów, jednak przeglądając zawartość modułu *prog2.obj*, zauważymy, że nazwa procedury zewnętrznej nie ma żadnych przyrostków, ale wypisana jest w najprostszej postaci — *proc1*. W związku z tym konieczna jest zmiana nazwy procedury w module *prog1.asm* z *PROC1@0* na *PROC1*. Dalsze łączenie tych modułów również nie będzie już sprawiało żadnych kłopotów:

```

tlink32 -aa prog1.obj prog2.obj

```

Do prac z bibliotekami pakiet TASM udostępnia specjalny program bibliotekarza — *tlb.exe*. Poniższym poleceniem utworzymy z modułu *prog2.obj* nową bibliotekę:

```

tlb lib1.lib + prog2.obj

```

W efekcie wywołania tego polecenia na dysku pojawi się plik biblioteki o nazwie *lib1.lib*. Teraz możemy połączyć moduł *prog1.obj* z utworzoną biblioteką:

```

tlink32 -aa prog1, prog1, prog1, lib1

```

W wyniku otrzymamy moduł wykonywalny *prog1.exe*.

Należy zwracać ścisłą uwagę na opcje wiersza poleceń programu *tlink32*. W najczęściej stosowanej formie wyglądają one mniej więcej tak⁵:

```
tlink32 -aa pliki_obj, plik_exe, plik_map, pliki_lib
```

- ♦ *pliki_obj* — jeden lub kilka plików obiektowych (rozdzielanych spacjami); moduł główny musi być wpisany jako pierwszy.
- ♦ *plik_exe* — plik wykonywalny.
- ♦ *plik_map* — plik *.map* zawierający informacje o strukturze modułu.
- ♦ *pliki_lib* — jedna lub kilka bibliotek (rozdzielanych spacjami).

Pakiet TASM nie pozwala na stosowanie dyrektywy *INVOKE*, dlatego w kolejnych przykładach będę starał się jej unikać⁶.

We wstępie do tej książki zadeklarowałem, że będę próbował jednakowo opisywać oba asemblery. Różnice pomiędzy nimi skupiają się przede wszystkim na dyrektywach i makroinstrukcjach (co zobaczymy w rozdziale 5.), dlatego najprostszym pomysłem na uzyskanie zgodności programów z obydwoma asemblerami jest unikanie stosowania dyrektyw i makroinstrukcji. Podstawę programowania w systemie Windows tworzą wywołania funkcji API (będzie o nich mowa w rozdziale 2.). Jednak wiemy już, że asemblery różnią się sposobem wywoływania zewnętrznych procedur; MASM wymaga stosowania przyrostka *@N*, a TASM obywa się bez niego. W tym zakresie nie uda się nam, niestety, uniknąć stosowania definicji makroinstrukcji. O tym jednak powiemy we właściwym czasie.

Uproszczony tryb segmentacji

Zarówno asembler MASM, jak i TASM obsługują tak zwaną segmentację uproszczoną. Osobiście wolę klasyczną strukturę programów w języku asemblera, ale muszę przyznać, że uproszczona segmentacja jest metodą bardzo wygodną, szczególnie w czasie programowania dla systemu Windows.

Segmentacja uproszczona polega na tym, że punkt początkowy segmentu kodu oznaczany jest dyrektywą *.CODE*, natomiast dyrektywa *.DATA*⁷ oznacza początek segmentu danych. Obie dyrektywy mogą pojawiać się wewnątrz programu wielokrotnie, a program asemblera odpowiednio poskłada ze sobą poszczególne segmenty kodu i danych. Głównym zadaniem tego rozwiązania jest umożliwienie umieszczania w kodzie źródłowym danych możliwie najbliżej miejsca ich wykorzystania w kodzie programu. Po-

⁵ Zaznaczam, że jest to forma nieco uproszczona.

⁶ Poza tym uważam, że najlepszym rozwiązaniem jest ręczne ładowanie parametrów na stos.

⁷ Dostępna jest też specjalna dyrektywa opisująca segment stosu — *.STACK*. Osobiście używam jej jednak wyjątkowo rzadko.

dobne rozwiązanie zostało swego czasu wprowadzone również w języku C++. Według mnie powoduje to jednak znaczące trudności w czasie czytania kodu takiego programu. Poza tym nie chciałbym uchodzić za przesadnego estety, ale nie podoba mi się program, w którym dane i kod są dowolnie przemieszane ze sobą.

Na listingu 1.9 przedstawiono sposób użycia trybu uproszczonej segmentacji.

Listing 1.9. Program korzystający z segmentacji uproszczonej

```
.586P
; Plaski model pamięci
.MODEL FLAT, stdcall
;-----
; Segment danych
.DATA
    SUM DWORD 0
; Segment kodu
.CODE
    START:
; Segment danych
.DATA
    A     DWORD 100
; Segment kodu
.CODE
    MOV EAX, A
; Segment danych
.DATA
    B     DWORD 200
; Segment kodu
.CODE
    ADD EAX, B
    MOV SUM, EAX
    RET                                ; Wyjście
END START
```



Dyrektywy w rodzaju `.DATA` i `.CODE` można umieszczać w tradycyjnie zdefiniowanym segmencie kodu. Jest to wygodna metoda tworzenia przydatnych makroinstrukcji, które będą opisywane w rozdziale 12.

Inne narzędzia używane do pracy z assemblerem

Na zakończenie tego rozdziału przedstawiam krótki opis innych programów często używanych w czasie programowania w języku assemblera. Później część z tych programów będzie opisywana dokładniej, a o pozostałych w ogóle nie będziemy już wspominać.

Edytory

Osobiście w czasie pisania programów assemblerowych nie używam żadnego specjalizowanego edytora, ale chcę, aby opisy w tej książce były pełne, dlatego wspomnę tutaj o dwóch takich narzędziach. Na początek zajmiemy się edytorem *qeditor.exe* dostarczonym w pakiecie assemblera MASM. Sam edytor, a także towarzyszące mu narzędzia, został napisany w assemblerze. Już pobieżna analiza możliwości edytora i jego rozmiaru może budzić uznanie. Jako przykład podam, że sam edytor ma tylko 27 kB, a narzędzie do przeglądania raportów i wyników przekształceń programów — 6 kB. Edytor ten doskonale nadaje się do pracy z niewielkimi aplikacjami mieszczącymi się w pojedynczych modułach. Nie pozwala jednak na wygodną pracę z kilkoma modułami. Działanie edytora opiera się na interakcji kilku narzędzi połączonych ze sobą plikami wsadowymi. Na przykład przekształcenia programów przeprowadzane są przez plik wsadowy *assmbl.bat*, który odpowiednio wywołuje assembler *ml.exe*, a wyniki jego działania zapisuje do pliku *assmbl.txt*. Do przejrzenia zawartości tego pliku konieczne jest użycie programu *thegun.exe*. Konsolidacja modułów programu wykonywana jest w podobny sposób.

Narzędzie *dumppe.exe* stosowane jest do deasemblowania modułów wykonywalnych, a wyniki tej operacji zapisywane są do pliku *disasm.txt*. Pozostałe operacje wykonywane są w podobny sposób. Wprowadzając zmiany do poszczególnych plików wsadowych, można modyfikować zachowania poszczególnych narzędzi, a w razie konieczności można nawet zamienić niektóre z używanych narzędzi (na przykład zamiast programu *ml.exe* zastosować program *tasm32.exe*).

Drugim edytorem, na który chciałbym wskazać, jest program *eas.exe* (*Easy Assembler Shell* — prosta powłoka assemblera). Edytor ten, lub, jak wskazuje jego nazwa, powłoka, pozwala na tworzenie, aseemblowanie i konsolidowanie złożonych projektów składających się z plików *.asm*, *.obj*, *.rc*, *.res* i *.def*. Program ten może współpracować z assemblerem TASM i MASM, a także z innymi narzędziami, takimi jak programy uruchomieniowe, edytory zasobów itd. Współpraca z assemblerami i konsolidatorami może być przygotowana dzięki opcjom tej powłoki pozwalającym na wprowadzenie opcji wiersza poleceń dla stosowanego narzędzia.

Programy uruchomieniowe

Programy uruchomieniowe zwane często także debuggerami (ang. *debuggers*) pozwalają na wykonywanie programów w trybie krok-po-kroku. W czwartej części książki opisywać będę dokładniej programy uruchomieniowe i deasemblerzy. Do najpopularniejszych programów uruchomieniowych⁸ na rynku należą CodeView firmy Microsoft, Turbo Debugger firmy Borland oraz program Ice⁹.

⁸ Razem z systemem Windows dostarczany jest nadal debugger *debug.exe*, jednak program ten nie obsługuje najnowszego formatu plików wykonywalnych.

⁹ Obecnie firma Microsoft udostępnia na swoich stronach internetowych bardzo dobry program uruchomieniowy WinDbg, który przeznaczony jest do pracy pod systemem operacyjnym Windows. Bardzo przyjazny jest z kolei OllyDbg dostępny na stronie autora tego bardzo dobrego programu uruchomieniowego — *przyp. red.*

Deasemblery

Deasemblery konwertują moduły wykonywalne na kod assemblerowy. Przykładem najprostszego deassemblera może być program *dumppe.exe* uruchamiany z poziomu wiersza poleceń. Na listingu 1.10 przedstawiono przykład wydruku przygotowanego przez program *dumppe.exe*. Wydruk ten jest wynikiem deasemblowania programu przedstawionego na listingu 1.4. Raczej trudno byłoby go rozpoznać...

Listing 1.10. *Wynik deasemblowania programu wykonanego przez program dumppe.exe*

	(hex)	(dec)
knla.exe		
.EXE size (bytes)	490	1168
Minimum load size (bytes)	450	1104
Overlay number	0	0
Initial CS:IP	0000:0000	
Initial SS:SP	0000:00B8	184
Minimum allocation (para)	0	0
Maximum allocation (para)	FFFF	65535
Header size (para)	4	4
Relocation table offset	40	64
Relocation entries	0	0
Portable executable starts at	a8	
Signature	00004550 (PE)	
Machine	014C (Intel386)	
Sections	0001	
Time date stamp	3AE6D1B1 Wed Apr 25 19:31:29 2001	
Symbol table	00000000	
Number of symbols	00000000	
Optional header size	00E0	
Characteristics	010F	
Relocation information stripped		
Executable image		
Line numbers stripped		
Local symbols stripped		
32-bit word machine		
Magic	010B	
Linker version	5.12	
Size of Code	00000200	
Size of initialized data	00000000	
Size of uninitialized data	00000000	
Address of entry point	00001000	
Base of code	00001000	
Base of data	00002000	
Image base	00400000	
Section alignment	00001000	
File alignment	00000200	
Operating system version	4.00	
Image version	0.00	
Subsystem version	4.00	
Reserved	00000000	
Image size	00002000	
Header size	00000200	
Checksum	00000000	
Subsystem	0002 (Windows)	

DLL characteristics	0000	
Size of stack reserve	00100000	
Size of stack commit	00001000	
Size of heap reserve	00100000	
Size of heap commit	00001000	
Loader flags	00000000	
Number of directories	00000010	
Directory name	VirtAddr	VirtSize

Export	00000000	00000000
Import	00000000	00000000
Resource	00000000	00000000
Exception	00000000	00000000
Security	00000000	00000000
Base relocation	00000000	00000000
Debug	00000000	00000000
Description/architecture	00000000	00000000
Machine value (MIPS GP)	00000000	00000000
Thread storage	00000000	00000000
Load configuration	00000000	00000000
Bound import	00000000	00000000
Import address table	00000000	00000000
Delay import	00000000	00000000
COM runtime descriptor	00000000	00000000
(reserved)	00000000	00000000
Section table		

Virtual address	0001000	
Virtual size	00000E	
Raw data offset	000200	
Raw data size	0000200	
Relocation offset	000000	
Relocation count	000	
Line number offset	00000000	
Line number count	000	
Characteristics	00000020	
Code		
Executable		
readable		
Disassembly		
00401000 start:		
00401000 E803000000	call	fn_00401008
00401005 C3	ret	
00401006 CC	int	3
00401007 CC	int	3
00401008 fn_00401008:		
00401008 B8E8030000	mov	eax,3E8h
0040100D C3	ret	

Chciałbym też wspomnieć o deassemblerze *W32Dasm*, który opiszę ze szczegółami w ostatniej części książki, a także doskonale znanym deassemblerze *Ida Pro*. W części czwartej przyjrzymy się tym programom dokładniej i omówimy techniki skutecznego ich wykorzystania.

Edytory szesnastkowe

Edytory szesnastkowe pozwalają na przeglądanie i edytowanie modułów wykonywalnych w formacie szesnastkowym. Tego rodzaju edytory wbudowane są w większość popularnych programów uruchomieniowych i deasemblerów. Wspomnę tutaj tylko o programie *hiew.exe*, bardzo popularnym w środowiskach hakerów. Program ten pozwala na przeglądanie zawartości modułów wykonywalnych zarówno w formacie szesnastkowym, jak i w kodzie asemblerowym. Dodatkowo, oprócz możliwości przeglądania plików wykonywalnych, program ten pozwala również na ich edycję.

Kompilatory zasobów

Oba pakiety, MASM i TASM, dostarczane są z kompilatorami zasobów, które będę opisywał w rozdziale 9. Programy te nazywają się odpowiednio *rc.exe* i *brc32.exe*.

Edytory zasobów

Najczęściej korzystam z edytora zasobów dołączanego do pakietu Borland C++ 5.0 albo pochodzącego z pakietu Visual Studio.NET, jednak proste zasoby można tworzyć przy pomocy właściwie dowolnego edytora tekstowego. Język zasobów omawiał będę w rozdziałach 9. i 10.