

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

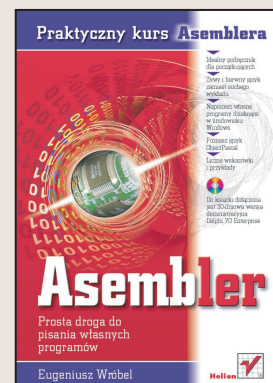
FRAGMENTY KSIĄŻEK ONLINE

Praktyczny kurs asemblera

Autor: Eugeniusz Wróbel

ISBN: 83-7361-433-8

Format: B5, stron: 384



Wejdź w świat programowania w języku asemblera

- Dowiedz się, kiedy użycie asemblera jest niezbędne
- Poznaj zasady programowania w asemblerze
- Napisz szybkie i wydajne programy dla DOS-a i Windows
- Zdobądź wiedzę o zasadach działania procesora i pamięci

Uważasz, że możliwości języków programowania wysokiego poziomu nie pozwalają na napisanie programu, którego potrzebujesz? Chcesz stworzyć sterownik, program rezydentny, demo lub... wirusa? Interesuje Cię, co dzieje się w komputerze podczas wykonywania programu?

Wykorzystaj potencjał asemblera!

Programowanie w języku niskiego poziomu daje niemal nieograniczoną kontrolę nad sprzętem i działaniem aplikacji. Programy napisane w języku asemblera działają szybko, są niewielkie i zajmują mało pamięci. Są bardzo wydajne i otwierają dostęp do takich obszarów komputera, do których dostęp z poziomu C++ czy Visual Basic jest niemożliwy.

Książka „Praktyczny kurs asemblera” wprowadzi Cię w świat programowania w tym języku. Dowiesz się, jak działa procesor, w jaki sposób komunikuje się z pamięcią i pozostałymi elementami komputera. Poznasz typy rozkazów procesora, tryby adresowania i zasady tworzenia programów w asemblerze. Lepiej poznasz swój komputer i dowiesz się, w jaki sposób zapamiętuje i przetwarza dane. Komputer przestanie być dla Ciebie „czarną skrzynką” wykonującą w czarodziejski sposób Twoje polecenia.

- Podstawowe wiadomości o architekturze procesorów Intel
- Organizacja pamięci i tryby adresowania
- Omówienie listy rozkazów procesora
- Narzędzia do tworzenia programów w języku asemblera
- Struktura programu w asemblerze
- Definiowanie zmiennych
- Tworzenie podprogramów i makrorozkazów
- Wykorzystanie funkcji BIOS-a oraz MS-DOS
- Programy w asemblerze uruchamiane w systemie Windows
- Optymalizacja kodu
- Tworzenie modułów dla innych języków programowania

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



Spis treści

Rozdział 1. Wprowadzenie	7
1.1. Co to jest asembler?	7
1.2. Dlaczego programować w języku asemblera?	10
1.3. Dlaczego warto poznać język asemblera?	12
1.4. Wymagane umiejętności	12
1.5. Konwencje stosowane w książce	13
Rozdział 2. Zaczynamy typowo — wiedząc niewiele, uruchamiamy nasz pierwszy program	17
2.1. „Hello, world!” pod systemem operacyjnym MS DOS	18
2.2. „Hello, world!” pod systemem operacyjnym Windows	22
Rozdział 3. Wracamy do podstaw — poznajemy minimum wiedzy na temat architektury procesorów 80x86	29
3.1. Rejestry procesora 8086	30
3.2. Zwiększamy rozmiar rejestrów — od procesora 80386 do Pentium 4	33
3.3. Zwiększamy liczbę rejestrów — od procesora 80486 do Pentium 4	35
3.4. Segmentowa organizacja pamięci	39
3.5. Adresowanie argumentów	43
3.6. Adresowanie argumentów w pamięci operacyjnej	44
Rozdział 4. Poznajemy narzędzia	47
4.1. Asembler MASM	49
4.2. Program konsolidujący — linker	52
4.3. Programy uruchomieniowe	54
4.4. Wszystkie potrzebne narzędzia razem, czyli środowiska zintegrowane	62
Rozdział 5. Nadmiar możliwości, z którym trudno sobie poradzić — czyli lista instrukcji procesora	67
5.1. Instrukcje ogólne — jednostki stałoprzecinkowej	70
5.2. Koprocesor arytmetyczny — instrukcje jednostki zmiennoprzecinkowej	73
5.3. Instrukcje rozszerzenia MMX	75
5.4. Instrukcje rozszerzenia SSE	78
5.5. Instrukcje rozszerzenia SSE2	82
5.6. Instrukcje rozszerzenia SSE3	85
5.7. Instrukcje systemowe	85

Rozdział 6. Wracamy do ogólnej struktury programu asemblerowego	87
6.1. Uproszczone dyrektywy definiujące segmenty	87
6.2. Pełne dyrektywy definiowania segmentów	92
6.3. Spróbujmy drobną część tej wiedzy zastosować w prostym programie, a przy okazji poznamy nowe pomocnicze dyrektywy	96
Rozdział 7. Ważna rzecz w każdym języku programowania — definiowanie i stosowanie zmiennych	105
7.1. Zmienne całkowite	106
7.2. Zmienne zmiennoprzecinkowe	109
7.3. Definiowanie tablic i łańcuchów	110
7.4. Struktury zmiennych	114
7.5. Dyrektywa definiująca pola bitowe	117
Rozdział 8. Podprogramy	119
8.1. Stos	119
8.2. Wywołanie i organizacja prostych podprogramów	122
8.3. Poznajemy dyrektywę PROC-ENDP	123
8.4. Parametry wywołania podprogramu	128
8.5. Zmienne lokalne	137
Rozdział 9. Oddalamy się od asemblera w kierunku języków wyższego poziomu, czyli użycie makroinstrukcji oraz dyrektyw asemblacji warunkowej	139
9.1. Makroinstrukcja definiowana	139
9.2. Dyrektywa LOCAL	144
9.3. Dyrektywy asemblacji warunkowej	144
9.4. Makroinstrukcje niedefiniowane	148
9.5. Makroinstrukcje tekstowe	149
9.6. Makroinstrukcje operujące na łańcuchach (na tekstach)	150
Rozdział 10. Czy obsługę wszystkich urządzeń komputera musimy wykonać sami? Funkcje systemu MS DOS oraz BIOS	153
10.1. Co ma prawo przerwać wykonanie naszego programu?	154
10.2. Obsługa klawiatury oraz funkcje grafiki na poziomie BIOS	156
10.3. Wywoływanie podprogramów systemu operacyjnego MS DOS	163
Rozdział 11. Obalamy mity programując w asemblerze pod systemem operacyjnym Windows	169
11.1. Systemowe programy biblioteczne	170
11.2. Najprawdziwsze pierwsze okno	173
11.3. Struktury programowe HLL — to też jest asembler!	178
11.4. Idziemy jeden krok dalej i wykorzystujemy program generatora okien Prostart	180
Rozdział 12. Czy możemy przyspieszyć działanie naszego programu? Wybrane zagadnienia optymalizacji programu	189
12.1. Kiedy i co w programie powinniśmy optymalizować?	191
12.2. Optymalizujemy program przygotowany dla procesora Pentium 4	193
12.3. Wspieramy proces optymalizacji programem Vtune	200
12.4. Na ile różnych sposobów możemy zakodować kopiowanie tablic?	201
Rozdział 13. Dzielimy program na moduły i łączymy moduły zakodowane w różnych językach programowania	209
13.1. Jak realizować połączenia międzymodułowe?	210
13.2. Mieszmą moduły przygotowane w różnych językach	214

Rozdział 14. Przykładowe programy (MS DOS)	219
14.1. Identyfikujemy procesor	219
14.2. Wchodzimy w świat grafiki — nieco patriotycznie	225
14.3. Program rezydentny, czyli namiastka wielozadaniowości	228
14.4. Pozorujemy głębie	233
14.5. Wyższa graficzna szkoła jazdy ze zmiennymi zespolonymi	236
Rozdział 15. Przykładowe programy (Windows)	243
15.1. Zegarek	243
15.2. Dotknięcie grafiki przez duże „G”	248
15.3. Przekształcamy mapę bitową	250
Załącznik 1. Interesujące strony w internecie	271
Załącznik 2. Lista dyrektyw i pseudoinstrukcji języka MASM	275
Z2.1. Dyrektywy określające listę instrukcji procesora	275
Z2.2. Organizacja segmentów	277
Z2.3. Definiowanie stałych oraz dyrektywy związane z nazwami symbolicznymi	279
Z2.4. Definiowanie zmiennych	280
Z2.4. Dyrektywy asemblacji warunkowej	282
Z2.5. Makroinstrukcje i dyrektywy nimi związane	283
Z2.6. Pseudoinstrukcje typu HLL	285
Z2.7. Dyrektywy związane z podprogramami	286
Z2.8. Dyrektywy wpływające na kształt listingu asemblacji	287
Z2.9. Połączenia międzymodułowe	289
Z2.10. Dyrektywy związane z diagnostyką procesu asemblacji	290
Z2.11. Inne dyrektywy i pseudoinstrukcje	291
Załącznik 3. Operatory stosowane w języku MASM	293
Z3.1. Operatory stosowane w wyrażeniach obliczanych w czasie asemblacji	293
Z3.2. Operatory stosowane w wyrażeniach obliczanych w czasie wykonywania programu	297
Załącznik 4. Symbole predefiniowane	299
Załącznik 5. Przegląd instrukcji procesora Pentium 4	303
Z5.1. Instrukcje ogólne (jednostki stałoprzecinkowej)	303
Z5.2. Instrukcje jednostki zmiennoprzecinkowej (koprocessora arytmetycznego)	309
Z5.3. Instrukcje rozszerzenia MMX	313
Z5.4. Instrukcje rozszerzenia SSE	315
Z5.5. Instrukcje rozszerzenia SSE2	319
Z5.6. Instrukcje rozszerzenia SSE3	323
Z5.7. Instrukcje systemowe	325
Załącznik 6. Opis wybranych przerwań systemu BIOS	327
Z6.1. Funkcje obsługi klawiatury wywoływane przerwaniem programowym INT 16h	327
Z6.2. Funkcje obsługi karty graficznej wywoływane przerwaniem programowym INT 10h	329
Załącznik 7. Wywołania funkcji systemu operacyjnego MS DOS	335
Z7.1. Funkcje realizujące odczyt lub zapis znaku z układu wejściowego lub wyjściowego	335
Z7.2. Funkcje operujące na katalogach	337
Z7.3. Operacje na dysku	337
Z7.4. Operacje na plikach (zbiorach) dyskowych	339
Z7.5. Operacje na rekordach w pliku	341

Z7.6. Zarządzanie pamięcią operacyjną.....	342
Z7.7. Funkcje systemowe.....	342
Z7.8. Sterowanie programem	344
Z7.9. Funkcje związane z czasem i datą.....	345
Z7.10. Inne funkcje	345
Załącznik 8. Opis wybranych funkcji API	347
Z8.1. CheckDlgButton	347
Z8.2. CloseHandle.....	348
Z8.3. CopyFile.....	349
Z8.4. CreateFile.....	350
Z8.5. CreateWindowEx.....	352
Z8.6. DeleteFile.....	355
Z8.7. ExitProcess.....	355
Z8.8. GetFileSize.....	356
Z8.9. MessageBox.....	357
Z8.10. ShowWindow.....	359
Załącznik 9. Tablica kodów ASCII oraz kody klawiszy	361
Z9.1. Kody ASCII	361
Z9.2. Kody klawiszy	361
Załącznik 10. Program Segment Prefix (PSP)	367
Załącznik 11. Płyta CD załączona do książki	369
Skorowidz	371

Rozdział 2.

Zaczynamy typowo — wiedząc niewiele, uruchamiamy nasz pierwszy program

Każdy szanujący się podręcznik programowania niezależnie od tego, jakiego języka dotyczy, musi rozpocząć się zgodnie z powszechnie szanowaną tradycją: programem wyświetlającym na ekranie monitora komunikat „Hello, world!”. Wprawdzie nie znamy jeszcze ani architektury procesora Pentium, ani też samego języka asemblera i jego składni, to jednak spróbujmy pokazać, jak wyglądałby taki miniprogram, i to zarówno w wersji przeznaczonej do uruchomienia pod systemem operacyjnym MS DOS, jak i pod systemem Windows. Tak jak to zostało zasygnalizowane w pierwszym rozdziale, nie będziemy chcieli w naszym pierwszym programie zrobić wszystkiego sami! Skorzystamy w obu przypadkach z podprogramów (biblioteki) systemowych wyświetlania tekstu na ekranie monitora. Tak więc nasz program sprowadzi się do:

- ◆ zadeklarowania, jaki tekst chcemy wyświetlić,
- ◆ przygotowania parametrów wywołania procedury systemowej,
- ◆ wywołania tejże procedury,
- ◆ zakończenia programu, czyli oddania sterowania systemowi operacyjnemu.

2.1. „Hello, world!” pod systemem operacyjnym MS DOS

Najprostszy program asemblerowy, który w okienku wirtualnego systemu MS DOS bądź na całym ekranie wyświetli żądany napis, może mieć następującą postać:

```
; Mój pierwszy program w asemblerze
; System operacyjny MS DOS
.model small          ; model pamięci: 1 segment programu i 1 segment danych
.386                  ; dostępny zbiór rozkazów procesora 80386

.data                 ; początek segmentu danych
    tekst byte "Hello, world!",0ah,0dh,"$"
.stack 100h           ; segment stosu o zadeklarowanym rozmiarze
.code                 ; początek segmentu z kodem programu
    .startup          ; makroinstrukcja generująca sekwencję
                        ; inicjującą rejestry segmentowe DS i SS
    mov dx, offset tekst ; wprowadzenie parametrów wywołania
    mov ah, 09h        ; procedury wyświetlenia komunikatu
    int 21h            ; wywołanie procedury systemowej
    .exit              ; makroinstrukcja generująca sekwencję końcową programu
end                    ; koniec programu źródłowego
```

Spróbujmy teraz kolejno objaśnić poszczególne linie programu. Od razu można zauważyć szczególną rolę znaku średnika (;). Za tym znakiem umieszczamy będziemy komentarz objaśniający działanie programu, który nie jest analizowany przez asembler. Im więcej będzie tego komentarza i im bardziej będzie wyczerpujący, tym nasz program będzie bardziej zrozumiały dla innych, jak i (po jakimś czasie) dla nas samych. A teraz wyjaśnijmy, co oznaczają poszczególne linie programu:

`.model small`

Procesor Pentium narzuca pewną specyficzną, segmentową organizację pamięci operacyjnej. Dyrektywa `.model` pozwala w prosty sposób zdefiniować tzw. model pamięci, czyli sposób zorganizowania segmentów w naszym programie. Parametr `small` informuje, że w naszym programie będziemy mieli do dyspozycji jeden segment z kodem programu oraz jeden segment z danymi.

`.386`

Ta dyrektywa powoduje, że asembler (program tłumaczący) zezwoli na używanie w programie zbioru instrukcji procesora 80386 (a więc także procesora Pentium). W naszym prostym przykładzie moglibyśmy pominąć tę dyrektywę, wtedy jednak asembler akceptowałby jedynie instrukcje procesora 8086. Wpłynęłoby to jedynie na nieco inne przetłumaczenie objaśnionej poniżej makroinstrukcji `.startup`, jednak bez zmiany jej znaczenia w programie.

`.data`

Dyrektywa definiująca początek segmentu z danymi.

```
tekst byte "Hello, world!", 0ah, 0dh, "$"
```

Linia programu deklarująca naszą jedyną zmienną, czyli tekst, jaki chcemy wyświetlić na ekranie. Słowo `tekst` jest zdefiniowaną przez nas nazwą zmiennej tekstowej, z kolei słowo `byte` jest dyrektywą asemblera informującą, że zmienna tekstowa jest w rzeczywistości zmienną bajtową. Każda z liter wyświetlanego tekstu umieszczonego w cudzysłowie zamieniona zostanie w procesie asemblacji na odpowiadający jej bajt w kodzie ASCII. `0ah`, `0dh` to kody ASCII nowej linii oraz powrotu karetki (przesunięcia kursora do początku linii) wyrażone liczbą szesnastkową. Ponieważ do wyświetlenia komunikatu na ekranie wykorzystamy procedurę systemową, łańcuch znaków zakończyć musimy kodem ASCII znaku dolara, co zapisaliśmy: `"$"`. Brak znaku dolara spowodowałby, że wyświetlane byłyby kolejne bajty z pamięci operacyjnej zawierające przypadkowe wartości.

```
.stack 100h
```

Dyrektywa kończy segment danych i deklaruje segment stosu o wielkości 256 bajtów (czyli 100 szesnastkowo).

```
.code
```

Dyrektywa definiująca kolejny segment, tym razem z kodem programu.

```
.startup
```

Pierwsze instrukcje programu powinny zainicjować odpowiednimi wartościami rejestry, które odpowiadają za rozmieszczenie segmentów w pamięci operacyjnej. Program musi mieć informację, jaki jest adres początku segmentu danych i segmentu stosu (adres początku segmentu programu zostanie ustawiony przez system operacyjny w czasie ładowania programu do pamięci). Makroinstrukcja predefiniowana `.startup` zostanie w czasie asemblacji zamieniona na ciąg instrukcji procesora wpisujących do rejestrów segmentowych odpowiednie wartości. Będzie ona równocześnie informacją, gdzie zaczyna się program.

Teraz nastąpi główna część programu. Instrukcje `mov`, będące jednymi z najczęściej używanych w asemblerze, powodują skopiowanie prawego argumentu w miejsce lewego. Będzie to przygotowanie parametrów wywołania podprogramu systemowego, który wywołamy rozkazem `int`.

```
mov dx, offset tekst
```

Rozkaz załaduje do rejestru procesora **DX** adres początku zmiennej tekstowej o nazwie `tekst` w pamięci operacyjnej. Będzie to adres 16-bitowy (taką wielkość ma rejestr **DX**) liczony względem początku segmentu danych, nazywany przez niektórych odległością w bajtach od początku segmentu. Adres ten będziemy nazywać krótko angielskim terminem *offset*. Offset to także nazwa operatora, który użyty w omawianej instrukcji powoduje wydzielenie z nazwy `tekst` tej części adresu.

```
mov ah, 09h
```

Wszystkie udostępnione programom użytkowym funkcje systemu operacyjnego MS DOS wywołuje się w identyczny sposób, dlatego ich rozróżnienie następuje przez podanie jej numeru w rejestrze **AH** procesora. Procedura wyprowadzająca na ekran monitora tekst, którego offset umieszczony jest w rejestrze **DX**, ma numer 9.

```
int 21h
```

Instrukcja procesora `int` powoduje wywołanie specyficznego podprogramu, tzw. podprogramu obsługi przerwania (ang. *interrupt*). System operacyjny MS DOS standardowo wykorzystuje rozkaz `int 21h` do wywoływania udostępnionych programom użytkowym funkcji (procedur) systemowych, rozróżnianych zawartością rejestru **AH**.

```
.exit
```

Makroinstrukcja predefiniowana `.exit` zostanie w procesie asemblacji zamieniona na następujący ciąg instrukcji procesora, który powoduje zakończenie programu i oddanie sterowania systemowi operacyjnemu:

```
mov ah, 4ch  
int 21h
```

Jak widać, jest to znane nam już wywołanie procedury systemowej, tym razem o numerze 4c szesnastkowo. W taki sposób kończyć będziemy każdy program asemblerowy uruchamiany pod systemem MS DOS.

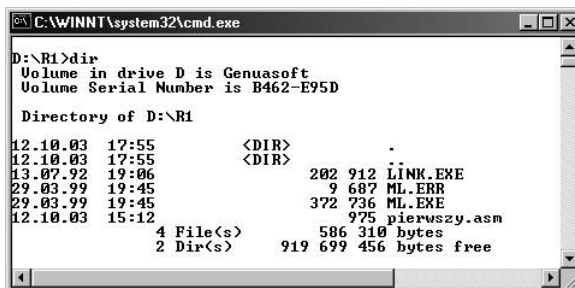
```
end
```

Dyrektywa kończąca program źródłowy, wstrzymująca proces asemblacji. Jeżeli w zbiorze z programem źródłowym będą jeszcze jakieś linie programu, to nie będą one analizowane.

Skoro wiemy już, jak wyglądać powinien nasz pierwszy program w asemblerze i co oznaczają poszczególne linie programu źródłowego, to należałoby teraz pokazać, co trzeba zrobić, aby program ten można było uruchomić. W celu maksymalnego uproszczenia przyjmijmy, że w wydzielonej kartotece mamy wszystkie niezbędne w tym przypadku programy (zbiory) tak, jak pokazuje to rysunek 2.1.

Rysunek 2.1.

Zawartość kartoteki
przed asemblacją



Zbiór o nazwie *pierwszy.asm* zawiera program źródłowy „Hello, world!”. Wyjaśnijmy rolę pozostałych zbiorów:

ML.EXE — program asemblera, pozwalający równocześnie wywołać program konsolidujący *LINK.EXE*,

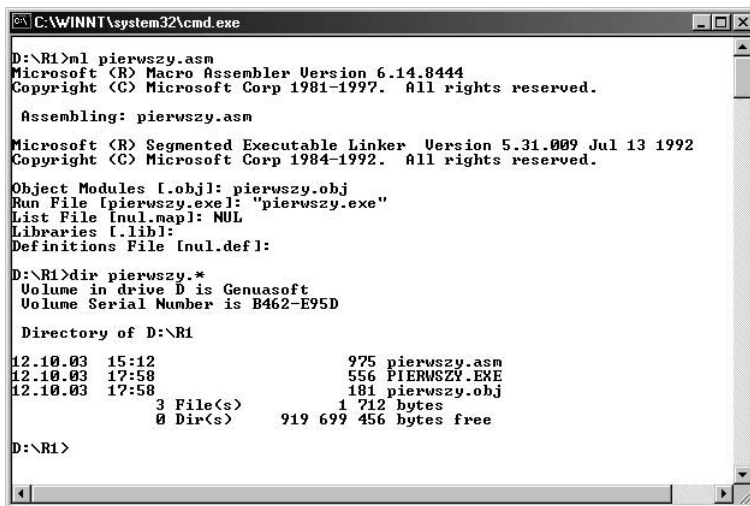
ML.ERR — zbiór tekstowy z komunikatami asemblera,

LINK.EXE — program konsolidujący (linker), może być wywoływany automatycznie przez *ML.EXE*.

Przykładowa sesja wywołania z linii komendy MS DOS programu asemblera i linkera zilustrowana jest na rysunku 2.2. Wyświetlenie zawartości kartoteki pokazuje, że w wyniku asemblacji oraz konsolidacji utworzone zostały zbiory *pierwszy.obj* oraz *pierwszy.exe*.

Rysunek 2.2.

Wywołanie
assemblera *ML.EXE*



```
C:\WINNT\system32\cmd.exe

D:\R1>ml pierwszy.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: pierwszy.asm

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: pierwszy.obj
Run File [pierwszy.exe]: "pierwszy.exe"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:

D:\R1>dir pierwszy.*
Volume in drive D is Genuasoft
Volume Serial Number is B462-E95D

Directory of D:\R1

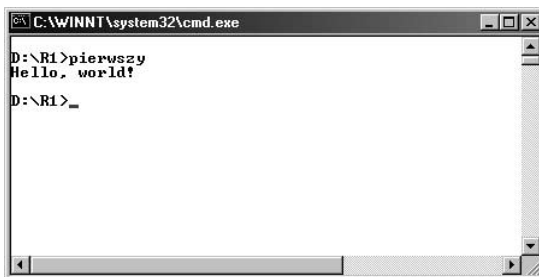
12.10.03 15:12                975 pierwszy.asm
12.10.03 17:58                556 PIERWSZY.EXE
12.10.03 17:58                181 pierwszy.obj
               3 File(s)          1 712 bytes
               0 Dir(s)          919 699 456 bytes free

D:\R1>
```

Na razie nie będziemy zajmować się rozwikłaniem wszystkich komunikatów, które wyświetliły się na ekranie. Ważne jest, że nie pojawił się żaden komunikat o błędzie asemblacji, możemy więc zaryzykować uruchomienie naszego pierwszego programu. Uzyskany efekt ilustruje rysunek 2.3.

Rysunek 2.3.

Efekt działania
programu
pierwszy.exe



```
C:\WINNT\system32\cmd.exe

D:\R1>pierwszy
Hello, world!

D:\R1>_
```



W linii programu źródłowego może wystąpić:

- ◆ instrukcja procesora zwana także rozkazem procesora (np. MOV, INT),
- ◆ dyrektywa sterująca pracą asemblera w procesie asemblacji (np. .MODEL, .DATA, .STACK, .CODE, END, .386), lub
- ◆ makroinstrukcja predefiniowana (np. .STARTUP, .EXIT) lub indywidualnie zdefiniowana przez nas (o czym będzie w następnych rozdziałach) i zamieniona przez asembler na ciąg instrukcji procesora i dyrektyw asemblera.

Funkcje systemu MS DOS udostępnione programom użytkowym wywołuje się rozkazem INT 21h po wcześniejszym załadowaniu do rejestru **AH** numeru funkcji.

2.2. „Hello, world!” pod systemem operacyjnym Windows

Okienkowa wersja naszego pierwszego (nazwiemy go dla rozróżnienia „drugim”) programu będzie niestety zupełnie inna. Wynika to z oczywistego faktu, że tak naprawdę to cały program sprowadza się do wywołania podprogramu bibliotecznego i nie ma w nim praktycznie żadnych obliczeń. Skoro system operacyjny jest zupełnie inny, inna jest biblioteka dostępnych funkcji, inna wreszcie jest także organizacja pamięci operacyjnej w przypadku pracy procesora w trybie adresacji wirtualnej z ochroną, ponadto system jest wielozadaniowy — zatem nasz program będzie musiał dostosować się do wszelkich wynikających z tego wymogów. Hasło „Hello, world!” wyświetlimy w odrębnym oknie (a ściślej w małym okienku z komunikatem) wykorzystując do tego funkcję *API* (ang. *The Microsoft® Win32® Application Programming Interface*). *API* stanowi zbiór dynamicznych bibliotek (DLL), z którego korzystają wszystkie procesy uruchamiane w systemie Windows. Więcej na ten temat — w jednym z dalszych rozdziałów książki, teraz wystarczy wiedzieć, że aby skorzystać ze wspomnianych funkcji *API*, należy do programu źródłowego dołączyć odpowiednie biblioteki oraz zbiory z deklaracjami tzw. *prototypów*. Program mógłby¹ wyglądać następująco:

```
; Mój drugi program w asemblerze, a pierwszy pod systemem operacyjnym Windows
.386
.model flat, stdcall                ; płaski model pamięci
option casemap :none                ; małe i duże litery rozróżniane

; dołączenie zbiorów z deklaracjami prototypów i symboli zewnętrznych
include windows.inc
include user32.inc
include kernel32.inc

; dołączenie zbiorów bibliotecznych
include lib user32.lib
include lib kernel32.lib
```

¹ Podkreślam słowo „mógłby wyglądać”, bowiem podobnie jak w innych językach programowania każdy problem można rozwiązać na wiele sposobów.

```
.data
    Tytuł_okna      byte "Mój drugi program",0
    Tekst_w_oknie   byte "  --- Hello, world! ---  ",0

.code
start:
    invoke MessageBox, NULL, ADDR Tekst_w_oknie, ADDR Tytuł_okna, MB_OK
    invoke ExitProcess, NULL
end start
```

Podobnie jak poprzednio, wyjaśnimy teraz znaczenie poszczególnych linii programu źródłowego, który — co widać na pierwszy rzut oka — kojarzy się raczej z programem w języku wyższego poziomu niż językiem asemblera, bowiem nie występują w nim wprost żadne instrukcje procesora. Wspominałem już, że wynika to z braku w programie jakichkolwiek obliczeń. Program sprowadza się do wywołania dwóch podprogramów bibliotecznych:

- ♦ *MessageBox* — wyświetlenia komunikatu, oraz
- ♦ *ExitProcess* — zakończenia programu.

.386

Podobnie jak w programie DOS-owym, dyrektywa ta powoduje możliwość używania w programie zbioru instrukcji procesora 80386. Nie jest przypadkiem, że tym razem dyrektywa ta jest pierwszą w programie. Ze względu na inny model pamięci, zadeklarowany w następnej linii programu źródłowego, taką kolejność dyrektyw będziemy zawsze stosować na początku programu, który ma być uruchomiony pod systemem Windows.

.model flat, stdcall

Procesor Pentium z systemem operacyjnym Windows pracuje z tzw. płaskim modelem pamięci (ang. *flat*). Jest to szczególny przypadek modelu segmentowego, dostępny w trybie adresacji wirtualnej z ochroną. Z kolei drugi parametr *stdcall* określa, w jakiej kolejności parametry wywołania procedur bibliotecznych w naszym programie przekazywane będą na stos i w jaki sposób stos będzie opróżniany z tych parametrów. Nie wnikając na tym etapie rozważań w większe szczegóły, zapamiętamy, że taka linia programu będzie się musiała zawsze znaleźć na drugim miejscu po dyrektywie wyboru procesora.

option casemap:none

Dyrektywa powoduje, że w nazwach symbolicznych rozróżniane będą małe i duże litery. Dyrektywa jest niezbędna, aby mogło się odbyć prawidłowe dołączenie procedur bibliotecznych *API*.

include

Następujące po sobie trzy linie programu z dyrektywą *include* powodują dołączenie do naszego programu źródłowego zbiorów o nazwach odpowiednio: *windows.inc*, *user32.inc* oraz *kernel32.inc*. W zbiorach tych znajdują się (w postaci kodów źródłowych) definicje tzw. prototypów procedur

bibliotecznych i nazw symbolicznych², które możemy dołączać do programu użytkowego. Brak tych prototypów i definicji spowodowałby, że użyte w programie nazwy wywoływanych procedur i niektórych stałych uznane byłyby jako niezdefiniowane i w konsekwencji program wynikowy nie mógłby zostać wygenerowany.

```
include lib
```

Dwie linie programu z dyrektywą `include lib` informują program konsolidujący *LINK.EXE*, jakie moduły biblioteczne powinien dołączyć w procesie konsolidacji, czyli generowaniu zbioru *drugi.exe*.

```
.data
```

Dyrektywa definiująca początek segmentu danych, w związku z płaskim modelem pamięci nazywanym w tym przypadku przez niektórych *sekcją* z danymi. Podobnie jak w przypadku programu uruchamianego pod systemem operacyjnym MS DOS, następne linie programu to deklaracje zmiennych tekstowych.

```
Tytuł_okna      byte "Mój drugi program",0
Tekst_w_oknie   byte "  --- Hello, world!  --- ",0
```

Dwie dyrektywy `byte` definiują dwie zmienne tekstowe zainicjowanymi napisami, z których jeden pokaże się na ramce okienka, drugi zaś — jako właściwy komunikat — wewnątrz. Podprogram biblioteczny, który wykorzystamy do wyświetlenia napisu, wymaga, aby tekst kończył się bajtem o wartości zero (podobnie jak w przypadku funkcji systemu MS DOS znakiem dolara).

```
.code
```

Początek segmentu (sekcji) z kodem programu.

```
start:
```

Etykieta (czyli adres symboliczny) informująca, że w tym miejscu powinno rozpocząć się wykonywanie programu. Etykieta ta będzie musiała dodatkowo wystąpić jako argument dyrektywy `end` kończącej program źródłowy.

```
invoke MessageBox, NULL, ADDR Tekst_w_oknie, ADDR Tytuł_okna, MB_OK
```

Dyrektywa `invoke` pozwala wywołać procedurę `MessageBox` — przesyła ona na stos parametry wywołania i wyświetla komunikat. Parametrami wywołania są w tym przypadku kolejno:

`NULL` — „uchwyt” do okna³, w naszym przypadku niepotrzebny i równy zero,
`ADDR Tekst_w_oknie` — adres początku komunikatu, jaki ma być wyświetlony,

² Użyliśmy dwóch takich niezdefiniowanych bezpośrednio w naszym programie nazw: `NULL` i `MB_OK`.

³ Pojęcie „uchwyt” związane jest z programowaniem pod systemem operacyjnym Windows i jest niezależne od języka programowania.

ADDR Tytuł_okna — adres początku napisu, jaki ma być na ramce okienka,

MB_OK — parametr określający, jakie standardowe przyciski mają być wyświetlone (w naszym przypadku jedynie przycisk „OK”).

Dyrektywa `invoke` należy do grupy dyrektyw asemblera nawiązujących do konstrukcji typowych dla języków wysokiego poziomu⁴. Operator `ADDR` w polu parametrów wywołania procedury ma w tym przypadku identyczne znaczenie, jak poznany wcześniej operator `offset` w polu argumentu instrukcji procesora. W czasie procesu asemblacji dyrektywa zostanie zastąpiona następującą prostą sekwencją instrukcji procesora:

```
push MB_OK
push offset Tytuł_okna
push offset Tekst_w_oknie
push NULL
call MessageBox
```

Jeżeli w programie źródłowym zamiast dyrektywy `invoke` napiszemy powyższe instrukcje, końcowy efekt działania programu będzie oczywiście dokładnie taki sam. Instrukcje procesora `push` przesyłają argumenty instrukcji na szczyt stosu, skąd pobierze go procedura `MessageBox`. Procedura wywołana zostanie typową instrukcją procesora wywołania podprogramu `call`.

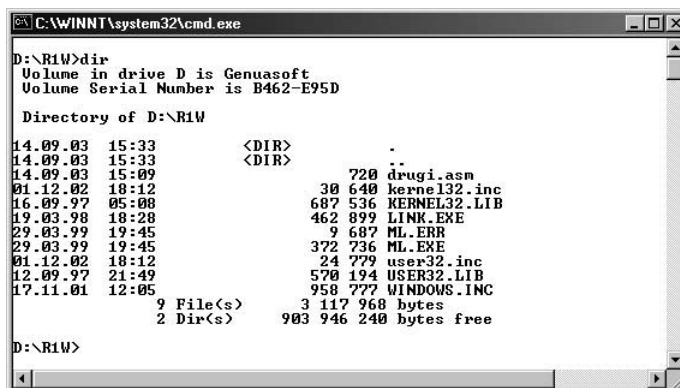
`invoke ExitProcess, NULL`

Wywołanie procedury kończącej program. W czasie asemblacji dyrektywa zostanie zastąpiona następującą sekwencją instrukcji:

```
push NULL
call ExitProcess
```

Skoro wiemy już, jak powinien wyglądać najprostszy program pracujący pod kontrolą systemu Windows, to spróbujmy go uruchomić. Podobnie jak w przypadku poprzedniego programu uruchomionego pod systemem MS DOS, umieścimy dla prostoty wszystkie niezbędne zbiory w jednej kartotece, tak jak to pokazuje rysunek 2.4.

Rysunek 2.4.
Zawartość kartoteki
przed asemblacją



⁴ Czasem nieco tylko złośliwie mówię, że tego typu dyrektywy oddalają język asemblera od prawdziwego asemblera...

Oprócz zbioru z programem źródłowym *drugi.asm*, programami asemblera *ML.EXE*, *ML.ERR* oraz programu konsolidatora *LINK.EXE*, w kartotece znajdujemy niezbędne dla prawidłowej asemblacji zbiory biblioteczne typu *.LIB* oraz dołączane do programu źródłowego zbiory typu *.INC* zawierające wspomniane definicje prototypów i symboli. Aby prawidłowo przeprowadzić proces asemblacji i konsolidacji, musimy wywołać program asemblera *ML.EXE* z przełącznikami, które pozwolą wygenerować zbiory *drugi.obj* oraz *drugi.exe* w postaci akceptowanej przez program *LINK.EXE* oraz system operacyjny Windows. Ilustruje to rysunek 2.5.

Rysunek 2.5.

Wywołanie
assemblera
z przełącznikami
pozwalającymi
wygenerować
program pod system
operacyjny Windows

```

C:\WINNT\system32\cmd.exe
D:\R1W>ml /coff drugi.asm /link /subsystem:windows
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: drugi.asm
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/subsystem:windows
"drugi.obj"
"/OUT:drugi.exe"

D:\R1W>_

```

Uważny Czytelnik zauważył zapewne, że użyliśmy tutaj dokładnie tego samego programu asemblera, co w poprzednim przykładzie dla systemu MS DOS. Musieliśmy jednak zastosować inny program konsolidatora *LINK.EXE*, który uwzględni fakt innej niż w poprzednim przypadku organizacji pamięci operacyjnej. Poszczególne przełączniki w naszym przykładzie mają następujące znaczenie:

/coff — Zbiór *drugi.obj* wygenerowany zostanie w formacie *Common Object File Format*

/link — Kolejne przełączniki dotyczyć będą programu konsolidatora

/subsystem:windows — Program wykonalny będzie uruchamiany pod systemem operacyjnym Windows

Po sprawdzeniu, że zbiory *drugi.obj* i *drugi.exe* zostały rzeczywiście utworzone, możemy uruchomić nasz program. Rysunek 2.6 pokazuje uzyskany rezultat. Program możemy uruchomić zarówno z linii komendy systemu MS DOS, jak i klikając myszą odpowiednią ikonę w okienku menedżera plików. Warto (tak przy okazji) zwrócić uwagę na wielkość zbioru *drugi.exe* i porównać go z uruchomionym wcześniej *pierwszy.exe*.

Rysunek 2.6.

Efekt wykonania
programu *drugi.exe*

```

C:\WINNT\system32\cmd.exe
D:\R1W>dir drugi.*
Volume in drive D is Genuasoft
Volume Serial Number is B462-E95D

Directory of D:\R1W

14.09.03  15:09                720 drugi.asm
14.09.03  15:45                2 560 drugi.exe
14.09.03  15:45                640 drugi.obj
               3 File(s)              3 920 bytes
               0 Dir(s)          903 942 144 bytes free

D:\R1W>drugi
D:\R1W>

```

Mój drugi program

--- Hello, world! ---

OK



Program asemblerowy, który ma być uruchomiony pod systemem operacyjnym Windows, może korzystać z programów bibliotecznych *API*, podobnie jak programy pisane w innych językach programowania. Program asemblera generujący zbiór typu *.obj* nie różni się w tym przypadku od asemblera używanego w przypadku programu uruchamianego pod systemem MS DOS, choć inne nieco jest jego wywołanie. Inny natomiast musi być zastosowany program konsolidatora, który uwzględnia specyfikę stosowanego po systemem operacyjnym Windows modelu pamięci.

Skoro uruchomiliśmy już dwa pierwsze proste programy, to przed próbą zaprojektowania nieco bardziej złożonych programów musimy wrócić do podstaw i w minimalnej dawce — niezbędnej programiście — omówić architekturę procesora, jego listę instrukcji oraz tryby adresowania argumentów.