ВЛАДИСЛАВ ПИРОГОВ

bhv®

АССЕМБЛЕР для WINDOWS

4-е ИЗДАНИЕ

- СРЕДСТВА ПРОГРАММИРОВАНИЯ В WINDOWS
- ОТЛАДКА, ИССЛЕДОВАНИЕ КОДА ПРОГРАММ, ДРАЙВЕРЫ
- СОЗДАНИЕ ДИНАМИЧЕСКИХ БИБЛИОТЕК
- МНОГОЗАДАЧНОЕ И СЕТЕВОЕ ПРОГРАММИРОВАНИЕ
- ГРАФИЧЕСКОЕ ПРОГРАММИРОВАНИЕ (GDI+, OpenGL, DirectX)
- ПРИМЕРЫ, ПРОВЕРЕННЫЕ НА РАБОТОСПОСОБНОСТЬ В ОС WINDOWS VISTA

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

УДК 681.3.068+800.92Assembler
ББК 32.973.26-018.1
П33

Пирогов В. Ю.

П33 Ассемблер для Windows. Изд. 4-е перераб. и доп. — СПб.: БХВ-Петербург, 2007. — 896 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0084-5

Рассмотрены необходимые сведения для программирования Windows-приложений на ассемблерах MASM и TASM: разработка оконных и консольных приложений; создание динамических библиотек; многозадачное программирование; программирование в локальной сети, в том числе и с использованием сокетов; создание драйверов, работающих в режиме ядра; простые методы исследования программ и др. В 4-м издании материал существенно переработан в соответствии с новыми возможностями ОС. Значительно шире рассмотрены вопросы управления файлами и API-программирования в Windows. Добавлен материал по программированию в ОС семейства Windows NT: Windows 2000/ XP/ Server 2003/Vista. На компакт-диске приведены многочисленные примеры, сопровождающие текст и проверенные на работоспособность в операционной системе Windows Vista.

Для программистов

УДК 681.3.068+800.92Assembler
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.08.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 72,24.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

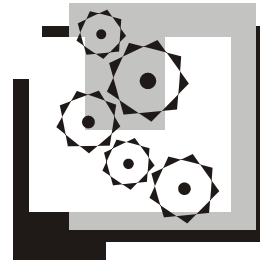
Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0084-5

© Пирогов В. Ю., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление



Глава 1.1

Средства программирования в Windows

Начнем, дорогой читатель. В данной главе я намерен дать некоторую вводную информацию о средствах программирования на языке ассемблера, а также предоставить начальные сведения о трансляции с языка ассемблера. Эта глава предназначена для начинающих программировать на языке ассемблера, поэтому программистам более опытным ее можно пропустить без особого ущерба для себя. Кроме этого, я собираюсь обзорно остановиться на возможностях пакета MASM32. Вместе с тем, начинающим я бы рекомендовал книгу [26], где программирование на языке ассемблера для операционной системы Windows излагается более последовательно и систематически.

Первая программа на языке ассемблера и ее трансляция

Рассмотрим общую схему трансляции программы, написанной на языке ассемблера. В исходном состоянии мы имеем один или более модулей на языке ассемблера (рис. 1.1.1). Двум стадиям трансляции с языка ассемблера соответствуют две программы пакета MASM32: ассемблер ML.EXE и редактор связей LINK.EXE[1]. Эти программы предназначены для того, чтобы перевести модуль (или модули) на языке ассемблера в исполняемый модуль, состоящий из команд процессора и имеющий расширение exe в операционной системе Windows.

Пусть файл с текстом программы на языке ассемблера называется PROG.ASM. Тогда, не вдаваясь в подробный анализ, две стадии трансляции на языке команд будут выглядеть следующим образом:

```
c:\masm32\bin\ml /c /coff PROG.ASM
```

[1] Программу LINK.EXE называют также компоновщиком или просто линковщиком.

Рис. 1.1.1. Схема трансляции ассемблерного модуля

После работы данной команды появляется *объектный модуль* (см. рис. 1.1.1) PROG.OBJ

```
c:\masm32\bin\Link /SUBSYSTEM:WINDOWS PROG.OBJ
```

В результате последней команды появляется исполняемый модуль PROG.EXE. Как вы, я надеюсь, догадались, `/c` и `/coff` являются параметрами программы ML.EXE, а `/SUBSYSTEM:WINDOWS` — параметром для программы LINK.EXE. О других ключах этих программ более подробно я расскажу в *главе 1.4*.

Чем больше я размышляю об этой схеме трансляции, тем более совершенной она мне кажется. Действительно, формат конечного модуля зависит от операционной системы. Установив стандарт на структуру объектного модуля, мы получаем возможность:

- использовать уже готовые объектные модули,
- интегрировать программы, написанные на разных языках *(см. главу 3.7)*.

Но самое прекрасное здесь то, что если стандарт объектного модуля распространить на разные операционные системы, то можно использовать модули, написанные в разных операционных системах[2].

[2] Правда, весьма ограниченно, т. к. согласование системных вызовов в разных операционных системах может вызвать весьма сильные затруднения.

Чтобы процесс трансляции сделать для вас привычным, рассмотрим несколько простых, "ничего не делающих" программ. Первая из них представлена в листинге 1.1.1.

Листинг 1.1.1. "Ничего не делающая" программа

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-----------------------------------------------
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
      RET   ;выход
_TEXT ENDS
END START
```

В листинге 1.1.1 представлена "ничего не делающая" программа. Назовем ее `PROG1`. Сразу отмечу на будущее: команды микропроцессора и директивы макроассемблера будем писать заглавными буквами.

Итак, чтобы получить загружаемый модуль, выполним следующие команды:[3]

```
ml /c /coff PROG1.ASM
link /SUBSYSTEM:WINDOWS PROG1.OBJ
```

Примем пока параметры трансляции программ как некую данность и продолжим наши изыскания.

Часто удобно разбить текст программы на несколько частей и объединить эти части еще на 1-й стадии трансляции. Это достигается посредством директивы `INCLUDE`. Например, один файл будет содержать код программы, а константы, данные (определение переменных) и прототипы внешних процедур помещаются в отдельные файлы. Обычно такие файлы записывают с расширением inc.

Именно такая разбивка демонстрируется в следующей программе (листинг 1.1.2).

[3] Если имя транслируемых модулей содержит пробелы, то название модулей придется заключать в кавычки, например, так: `ML /c /coff "PROG 1.ASM"`.

Листинг 1.1.2. Пример использования директивы INCLUDE

```
;файл CONSTANTS.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
CONS11 EQU 11000
CONS12 EQU 12000
;файл DATA.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

;файл PROG1.ASM
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;подключить файл констант
INCLUDE CONSTANTS.INC
;-------------------------------------------------------
;сегмент данных
_DATA SEGMENT
;подключить файл данных
INCLUDE DATA.INC
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
       MOV EAX,CONS1
       SHL EAX,1  ;умножение на 2
```

```
       MOV DAT1,EAX
;----------------------------------
       MOV EAX,CONS2
       SHL EAX,2  ;умножение на 4
       MOV DAT2,EAX
;----------------------------------
       MOV EAX,CONS3
       ADD EAX,1000 ;прибавим 1000
       MOV DAT3,EAX
;----------------------------------
       MOV EAX,CONS4
       ADD EAX,2000 ;прибавим 2000
       MOV DAT4,EAX
;----------------------------------
       MOV EAX,CONS5
       SUB EAX,3000 ;вычесть 3000
       MOV DAT5,EAX
;----------------------------------
       MOV EAX,CONS6
       SUB EAX,4000 ;вычесть 4000
       MOV DAT6,EAX
;----------------------------------
       MOV EAX,CONS7
       MOV EDX,3
       IMUL EDX ;умножение на 3
       MOV DAT7,EAX
;----------------------------------
       MOV EAX,CONS8
       MOV EDX,7 ;умножение на 7
       IMUL EDX
       MOV DAT8,EAX
;----------------------------------
       MOV EAX,CONS9
       MOV EBX,3 ;деление на 3
       MOV EDX,0
       IDIV EBX
       MOV DAT9,EAX
;----------------------------------
       MOV EAX,CONS10
       MOV EBX,7 ;деление на 7
       MOV EDX,0
       IDIV EBX
       MOV DAT10,EAX
;----------------------------------
       MOV EAX,CONS11
       SHR EAX,1 ;деление на 2
```

```
        MOV DAT11,EAX
;-----------------------------------
        MOV EAX,CONS12
        SHR EAX,2 ;деление на 4
        MOV DAT12,EAX
;-----------------------------------
        RET           ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.2:

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

Программа из листинга 1.1.2 также достаточно бессмысленна (как и все программы данной главы), но зато демонстрирует удобства использования директивы INCLUDE. Напомню, что мы не останавливаемся в книге на очевидных командах микропроцессора *(см. приложение 2)*. Замечу только по поводу команды IDIV. В данном случае команда IDIV осуществляет операцию деления над операндом, находящемся в паре регистров EDX:EAX. Обнуляя EDX, мы указываем, что операнд целиком находится в регистре EAX.

Трансляция программы осуществляется так, как это было указано ранее.

> ***Замечание о типах данных***
>
> В данной книге вы встретитесь в основном с тремя типами данных (простых): байт, слово, двойное слово. При этом используются следующие стандартные обозначения. Байт — BYTE или DB, слово — WORD или DW, двойное слово — DWORD или DD. Выбор, скажем, в одном случае DB, а в другом BYTE, продиктован лишь желанием автора несколько разнообразить изложение. Подробнее *см. в главе 2.7*.

Объектные модули

Перейдем теперь к вопросу об объединении нескольких объектных модулей и подсоединении объектных библиотек на второй стадии трансляции. Прежде всего, замечу, что, сколько бы ни объединялось объектных модулей, один объектный модуль является главным. Смысл этого весьма прост: именно с этого модуля начинается исполнение программы. На этом различие между модулями заканчивается. Условимся далее, что главный модуль всегда в начале сегмента кода будет содержать метку START, ее мы указываем после директивы END — транслятор должен знать точку входа программы, чтобы указать ее в заголовке загружаемого модуля *(см. приложение 4)*.

Обычно во второстепенные модули помещаются процедуры, которые будут вызываться из основного и других модулей. Рассмотрим пример такого модуля. Этот модуль вы можете видеть в листинге 1.1.3.

Листинг 1.1.3. Модуль PROG2.ASM, процедура которого PROC1 будет вызываться из основного модуля

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
_TEXT SEGMENT
PROC1 PROC
      MOV EAX,1000
      RET
PROC1 ENDP
_TEXT ENDS
END
```

Прежде всего, обращаю ваше внимание на то, что после директивы END не указана какая-либо метка. Ясно, что это не главный модуль, процедуры его будут вызываться из других модулей. Другими словами, все его точки входа вторичны и совпадают с адресами процедур, которые там расположены.

Второе, на что я хотел бы обратить ваше внимание, — это то, что процедура, которая будет вызываться из другого объектного модуля, должна быть объявлена как PUBLIC. Тогда это имя будет сохранено в объектном модуле и далее может быть связано с вызовами из других модулей программой LINK.EXE.

Итак, выполняем команду ML /coff /c PROG1.ASM. В результате на диске появляется объектный модуль PROG2.OBJ.

А теперь проведем маленькое исследование. Просмотрим объектный модуль с помощью какой-нибудь простой viewer-программы, например той, что есть у программы Far.exe. И что же мы обнаружим? Вместо имени PROC1 мы увидим имя _PROC1@0. Это особый разговор — будьте сейчас внимательны! Во-первых, подчеркивание в начале имени отражает стандарт ANSI, предписывающий всем внешним именам (доступным нескольким модулям) автоматически добавлять символ подчеркивания.[4] Здесь ассемблер будет действовать автоматически, и у нас по этому поводу не будет никаких забот.

[4] В дальнейшем *(см. главу 3.7)* мы узнаем, что иногда требуется, чтобы символ подчеркивания отсутствовал в объектном модуле.

Сложнее с припиской @0. Что она значит? На самом деле все просто: цифра после знака @ указывает количество байтов, которые необходимо передать в стек в виде параметров при вызове процедуры. В данном случае ассемблер понял так, что наша процедура параметров не требует. Сделано это для удобства использования директивы INVOKE. Но о ней речь пойдет далее, а пока попытаемся сконструировать основной модуль PROG1.ASM (листинг 1.1.4).

Листинг 1.1.4. Модуль PROG1.ASM с вызовом процедуры из модуля PROG2.ASM

```
.586P
; плоская модель памяти
.MODEL FLAT, STDCALL
;--------------------------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET   ;выход
_TEXT ENDS
END START
```

Как вы понимаете, процедура, которая расположена в другом модуле, но будет вызываться из данного модуля, объявляется как EXTERN. Далее, вместо имени PROC1 нам приходится использовать имя PROC1@0. Здесь пока ничего нельзя сделать. Может возникнуть вопрос о типе NEAR. Дело в том, что когда-то в операционной системе MS-DOS тип NEAR означал, что вызов процедуры (или безусловный переход) будет происходить в пределах одного сегмента. Тип FAR означал, что процедура (или переход) будет вызываться из другого сегмента. В операционной системе Windows реализована так называемая *плоская модель памяти*, когда все адресное пространство процесса можно рассматривать как один большой сегмент. И здесь логично использование типа NEAR.

Выполним команду ML /coff /c PROG1.ASM, в результате получим объектный модуль PROG1.OBJ. Теперь можно объединить модули и получить загружаемую программу PROG1.EXE:

```
LINK /SUBSYSTEM:WINDOWS PROG1.OBJ PROG2.OBJ
```

При объединении нескольких модулей первым должен указываться главный, а остальные — в произвольном порядке. Название исполняемого модуля тогда будет совпадать с именем главного модуля.

Директива *INVOKE*

Обратимся теперь к директиве INVOKE. Довольно удобная команда, я вам скажу, правда, по некоторым причинам (которые станут понятными позже) я почти не буду употреблять ее в своих программах.

Удобство ее заключается, во-первых, в том, что мы сможем забыть о добавке @N. Во-вторых, эта команда сама заботится о помещении передаваемых параметров в стек. Последовательность команд

```
PUSH par1
PUSH par2
PUSH par3
PUSH par4
CALL NAME_PROC@N ; N - количество отправляемых в стек байтов
```

заменяется на

```
INVOKE NAME_PROC, par4, par3, par2, par1
```

Причем параметрами могут являться регистр, непосредственно значение или адрес. Кроме того, для адреса может использоваться как оператор OFFSET, так и оператор ADDR *(см. главу 2.6)*.

Видоизменим теперь модуль PROG1.ASM (модуль PROG2.ASM, представленный в листинге 1.1.3, изменять не придется). Измененный модуль расположен в листинге 1.1.5.

Листинг 1.1.5. Пример использования директивы INVOKE

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;--------------------------------------
;прототип внешней процедуры
PROC1 PROTO
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
INVOKE PROC1
```

```
RET   ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.5:

```
ml /c /coff prog1.asm
link /SUBSYSTEM:WINDOWS prog1.OBJ prog2.OBJ
```

Как видите, внешняя процедура объявляется теперь при помощи директивы `PROTO`. Данная директива позволяет при необходимости указывать и наличие параметров. Например, строка

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

будет означать, что процедура требует два параметра длиной в четыре и два байта (всего 6, т. е. `@6`).

Как уже говорилось, я буду редко использовать оператор `INVOKE`. Теперь я назову первую причину такого пренебрежения к данной возможности. Дело в том, что я сторонник чистоты языка ассемблера, и любое использование макросредств вызывает у меня чувство несовершенства. На мой взгляд, и начинающим программистам не стоит увлекаться макросредствами, иначе они не почувствуют всю красоту этого языка. О второй причине вы узнаете позже.

На нашей схеме (см. рис. 1.1.1) указано, что существует возможность подсоединения не только объектных модулей, но и библиотек. Собственно, если объектных модулей несколько, то это по понятным причинам вызовет неудобства. Поэтому объектные модули объединяются в библиотеки. Для подсоединения библиотеки в MASM удобнее всего использовать директиву `INCLUDELIB`, которая сохраняется в объектном коде и используется программой LINK.EXE.

Но как создать библиотеку из объектных модулей? Для этого у программы LINK.EXE имеется специальная опция `/LIB`, используемая для управления статическими библиотеками. Предположим, мы хотим создать библиотеку LIB1.LIB, состоящую из одного модуля — PROG2.OBJ. Выполним для этого следующую команду:

```
LINK /lib /OUT:LIB1.LIB PROG2.OBJ
```

Если необходимо добавить в библиотеку еще один модуль (MODUL.OBJ), то достаточно выполнить команду:[5]

```
LINK /LIB LIB1.LIB MODUL.OBJ
```

Вот еще два полезных примера использования библиотекаря:

---

[5] Вместо косой черты для выделения параметра программы LINK.EXE можно использовать черточку, например, так `LINK -LIB LIB1.LIB MODUL.OBJ`.

- `LINK /LIB /LIST LIB1.LIB` — получить список модулей библиотеки;
- `LINK /LIB /REMOVE:MODUL.OBJ LIB1.LIB` — удалить из библиотеки модуль MODUL.OBJ.

Вернемся теперь к нашему примеру. Вместо объектного модуля мы теперь используем библиотеку LIB1.LIB. Видоизмененный текст программы PROG1.ASM представлен в листинге 1.1.6.

Листинг 1.1.6. Пример использования библиотеки

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;-------------------------------------------------------
INCLUDELIB LIB1.LIB
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
CALL PROC1@0
RET  ;выход
_TEXT ENDS
END START
```

Трансляция программы из листинга 1.1.6:

```
ml /c /coff prog1.asm
link /SUBSYSTEM:WINDOWS prog1.OBJ
```

Данные в объектном модуле

Рассмотрим теперь менее важный (для нас) вопрос об использовании данных (переменных), определенных в другом объектном модуле. Здесь читателю, просмотревшему предыдущий материал, должно быть все понятно, а модули PROG2.ASM и PROG1.ASM, демонстрирующие технику использования внешних[6] переменных, приводятся в листингах 1.1.7 и 1.1.8.

[6] Термин "внешняя переменная" используется нами по аналогии с термином "внешняя процедура".

Листинг 1.1.7. Модуль, содержащий переменную ALT, которая используется в другом модуле (PROG1.ASM)

```
.586P
;модуль PROG2.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
PUBLIC PROC1
PUBLIC ALT
;сегмент данных
_DATA SEGMENT
ALT DWORD 0
_DATA ENDS
_TEXT SEGMENT
PROC1 PROC
MOV EAX,ALT
ADD EAX,10
RET
PROC1 ENDP
_TEXT ENDS
END
```

Листинг 1.1.8. Модуль, использующий переменную ALT, определенную в другом модуле (PROG2.ASM)

```
.586P
;модуль PROG1.ASM
;плоская модель памяти
.MODEL FLAT, STDCALL
;---------------------------------------
;прототип внешней процедуры
EXTERN   PROC1@0:NEAR
;внешняя переменная
EXTERN   ALT:DWORD
;сегмент данных
_DATA SEGMENT
_DATA ENDS
;сегмент кода
_TEXT SEGMENT
START:
MOV  ALT,10
CALL PROC1@0
MOV  EAX,ALT
RET   ;выход
_TEXT ENDS
END START
```

Замечу, что в отличие от внешних процедур, внешняя переменная[7] не требует добавки @N, поскольку размер переменной известен.

Трансляция модулей из листингов 1.1.7 и 1.1.8:

```
ml /c /coff prog2.asm
ml /c /coff prog1.asm
link /subsystem:windows prog1.obj prog2.obj
```

Упрощенный режим сегментации

Ассемблер MASM32 поддерживает так называемую *упрощенную сегментацию*. Я являюсь приверженцем классической структуры ассемблерной программы, но должен признаться, что упрощенная сегментация довольно удобная штука, особенно при программировании под Windows. Суть такой сегментации в следующем: начало сегмента определяется директивой .CODE, а сегмента данных — .DATA[8]. Причем обе директивы могут появляться в тексте программы несколько раз. Транслятор затем собирает код и данные вместе, как положено. Основной целью такого подхода, по-видимому, является возможность приблизить в тексте программы данные к тем строкам, где они используются. Такая возможность, как известно, в свое время была реализована в C++ (в классическом языке C это было невозможно). На мой взгляд, она приводит к определенному неудобству при чтении текста программы и дальнейшей модификации кода. Кроме того, не сочтите меня за эстета, но когда я вижу данные, перемешанные в тексте программы с кодом, у меня возникает чувство дискомфорта.

В листинге 1.1.9 представлена программа, демонстрирующая упрощенный режим сегментации.

Листинг 1.1.9. Пример программы, использующей упрощенную сегментацию

```
.586P
;плоская модель памяти
.MODEL FLAT, STDCALL
;-------------------------------------------------------
;сегмент данных
.DATA
SUM DWORD 0
;сегмент кода
```

[7] То есть (еще раз подчеркну) переменная, определенная в другом модуле.

[8] Разумеется, есть директива и для стека — это .STACK, но мы ее почти не будем использовать.

```
.CODE
START:
;сегмент данных
.DATA
A    DWORD 100
;сегмент кода
.CODE
MOV EAX,A
;сегмент данных
.DATA
B    DWORD 200
;сегмент кода
.CODE
ADD EAX,B
MOV SUM,EAX
RET   ;выход
END START
```

Трансляция программы из листинга 1.1.9:

```
ml /c /coff prog.asm
link /subsystem:windows prog.obj
```

> ***Замечание***
>
> Заметим, что макрокоманды `.DATA` и `.CODE` могут использоваться внутри кодового сегмента, определенного традиционным способом. Это удобно для создания разных полезных макроопределений (о макроопределениях подробнее *см. в главе 2.7*).

О пакете MASM32

В данной книге я делаю упор на использование программ пакета MASM32. Последние версии данного пакета можно свободно скачать с сайта **http://www.movsd.com**. Это сайт Стива Хатчессона, создателя пакета MASM32. Данный пакет специально предназначен для создания исполняемого кода для операционной системы Windows и базируется на созданных и поддерживаемых фирмой Microsoft продуктах, таких как транслятор языка ассемблера ML.EXE или редактор связи LINK.EXE.

Пакет MASM32 является свободно распространяемым продуктом, и вы, уважаемые читатели, можете законно использовать его для создания своих программ. Особенностью пакета MASM32 является то, что он ориентирован на создание программ, состоящих из макроопределений и библиотечных про-