

ВЛАДИСЛАВ ПИРОГОВ



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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.08.07.

Формат 70×100¹/₁₆. 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

- Введение..... 1**
 - Что нового?2
 - Соглашения.....4
 - О Windows Vista.....5
 - Структура изложения.....5
- Введение ко второму изданию книги "Ассемблер для Windows" 11**
- Введение к третьему изданию книги "Ассемблер для Windows" 15**
- ЧАСТЬ I. ОСНОВЫ ПРОГРАММИРОВАНИЯ В WINDOWS 17**
 - Глава 1.1. Средства программирования в Windows 19**
 - Первая программа на языке ассемблера и ее трансляция 19
 - Объектные модули 24
 - Директива *INVOKE* 27
 - Данные в объектном модуле 29
 - Упрощенный режим сегментации 31
 - О пакете MASM32..... 32
 - Обзор пакета MASM32 33
 - Трансляторы..... 35
 - Редактор QEDITOR 35
 - Дизассемблеры..... 37
 - Глава 1.2. Основы программирования в операционной системе Windows..... 42**
 - Вызов функций API..... 44
 - Структура программы..... 46
 - Регистрация класса окон..... 46
 - Создание окна 47

Цикл обработки очереди сообщений.....	47
Процедура главного окна.....	48
Примеры простых программ для Windows.....	49
Еще о цикле обработки сообщений.....	56
Передача параметров через стек.....	58
Глава 1.3. Примеры простых программ на ассемблере	61
Принципы построения оконных приложений	61
Окно с кнопкой.....	63
Окно с полем редактирования	68
Окно со списком.....	76
Дочерние и собственные окна	85
Глава 1.4. Ассемблер MASM.....	95
Командная строка ML.EXE	95
Командная строка LINK.EXE	98
Включение в исполняемый файл отладочной информации	101
Получение консольных и GUI-приложений	107
Автоматическая компоновка.....	107
"Самотранслирующаяся" программа	107
Глава 1.5. О кодировании текстовой информации в операционной системе Windows.....	109
О кодировании текстовой информации	109
OEM и ANSI.....	110
Кодировка Unicode.....	111
ЧАСТЬ II. ПРОСТЫЕ ПРОГРАММЫ, КОНСОЛЬНЫЕ ПРИЛОЖЕНИЯ, ОБРАБОТКА ФАЙЛОВ.....	117
Глава 2.1. Вывод графики и текста в окно. Библиотека GDI	119
Вывод текста в окне	119
Выбор шрифта	135
Графические образы	141
Глава 2.2. Графика: GDI+, DirectX, OpenGL.....	154
Работаем с функциями GDI+	154
Библиотека DirectX	165
Программируем на OpenGL	177

Глава 2.3. Консольные приложения.....	191
Создание консоли.....	194
Обработка событий от мыши и клавиатуры.....	200
Событие <i>KEY_EVENT</i>	201
Событие <i>MOUSE_EVENT</i>	202
Событие <i>WINDOW_BUFFER_SIZE_EVENT</i>	203
Таймер в консольном приложении.....	208
Глава 2.4. Понятие ресурса. Редакторы и трансляторы ресурсов	217
Язык описания ресурсов.....	217
Пиктограммы	218
Курсоры	219
Битовые изображения	220
Строки.....	220
Диалоговые окна.....	220
Меню.....	226
Акселераторы.....	232
Немодальные диалоговые окна.....	235
Глава 2.5. Примеры программ, использующих ресурсы	243
Динамическое меню.....	243
Горячие клавиши.....	254
Управление списками	263
Программирование в стиле Windows XP и Windows Vista.....	270
Глава 2.6. Управление файлами: начало	277
Характеристики файлов.....	277
Атрибут файла	278
Временные характеристики.....	279
Длина файла	280
Имя файла.....	281
Файловая система FAT32	282
Файловая система NTFS.....	285
Каталоги в NTFS.....	290
Сжатие файлов в NTFS	290
Точки повторной обработки	291
Поиск файлов.....	292
Приемы работы с двоичными файлами	310
Пример получения временных характеристик файла.....	324

Глава 2.7. Директивы и макросредства ассемблера	329
Метки.....	329
Строки	332
Структуры	332
Объединения	333
Удобный прием работы со структурами.....	333
Условное ассемблирование	334
Вызов процедур	335
Макроповторения	336
Макроопределения	337
Некоторые другие директивы транслятора ассемблера	339
Конструкции времени исполнения программы.....	340
Пример программы одинаково транслируемой как в MASM, так и в TASM	342
Глава 2.8. Еще об управлении файлами (<i>CreateFile</i> и другие функции).....	344
Полное описание функции <i>CreateFile</i> для работы с файлами.....	344
Другие возможности функции <i>CreateFile</i>	349
Почтовый ящик или mailslot	350
Каналы передачи информации (pipes)	355
Дисковые устройства	356
Обзор некоторых других функций API, используемых для управления файлами.....	360
Асинхронный ввод/вывод	361
Запись в файл дополнительной информации	366
ЧАСТЬ III. СЛОЖНЫЕ ПРИМЕРЫ ПРОГРАММИРОВАНИЯ В WINDOWS	369
Глава 3.1. Таймер в оконных приложениях	371
Общие сведения.....	371
Простейший пример использования таймера.....	373
Взаимодействие таймеров	378
Всплывающие подсказки.....	384
Глава 3.2. Многозадачное программирование.....	397
Процессы и потоки.....	397
Потоки	412

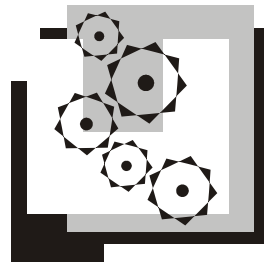
Взаимодействие потоков	418
Семафоры	420
События	422
Критические секции	422
Взаимоисключения	433
Глава 3.3. Создание динамических библиотек.....	434
Общие понятия	434
Создание динамических библиотек	437
Неявное связывание	442
Использование общего адресного пространства	443
Совместное использование памяти разными процессами	452
Глава 3.4. Сетевое программирование	456
Сетевые устройства.....	456
Поиск сетевых устройств и подключение к ним.....	463
О сетевых протоколах TCP/IP.....	478
О модели OSI	478
О семействе TCP/IP	479
Об IP-адресации	481
Маскирование адресов	482
Физические адреса и адреса IP	483
О службе DNS	483
Автоматическое назначение IP-адресов	484
Маршрутизация и ее принципы	484
Управление сокетами.....	485
Пример простейшего клиента и сервера.....	490
Глава 3.5. Разрешение некоторых проблем программирования в Windows.....	504
Глава 3.6. Некоторые вопросы системного программирования в Windows.....	555
Страничная и сегментная адресация	555
Адресное пространство процесса	560
Управление памятью.....	563
Динамическая память.....	563
Виртуальная память.....	571
Фильтры (HOOKS).....	572

Глава 3.7. Совместное использование ассемблера с языками высокого уровня.....	581
Согласование вызовов (исторический экскурс)	581
Согласование имен.....	582
Согласование параметров.....	583
Простой пример использования ассемблера с языками высокого уровня	584
Передача параметров через регистры	589
Вызовы API и ресурсы в ассемблерных модулях	591
Развернутый пример использования языков ассемблера и С	597
Встроенный ассемблер	605
Пример использования динамической библиотеки	607
Использование языка С из программ, написанных на языке ассемблера	610
Глава 3.8. Программирование сервисов.....	615
Основные понятия и функции управления	615
Структура сервисов.....	618
Пример сервиса	623
ЧАСТЬ IV. ОТЛАДКА, АНАЛИЗ КОДА ПРОГРАММ, ДРАЙВЕРЫ	639
Глава 4.1. Обзор инструментов для отладки и дизассемблирования.....	641
Утилиты фирмы Microsoft.....	641
EDITBIN.EXE.....	641
DUMPBIN.EXE	643
Дизассемблер W32Dasm	646
Отладчик OllyDbg.....	646
Другие инструменты	647
DUMPPE.EXE	647
Hiew.exe	647
DEWIN.EXE	650
IDA Pro.....	650
Глава 4.2. Отладчик OllyDbg	656
Начало работы с отладчиком	656
Окна отладчика	656
Отладочное выполнение	659
Точки останова	660
Обычные точки останова	660
Условные точки останова	661
Условные точки останова с записью в журнал.....	661

Точка останова на сообщения Windows	661
Точка останова на функции импорта	663
Точка останова на область памяти	663
Точка останова в окне <i>Memory</i>	663
Аппаратные точки останова	664
Другие возможности	664
Окно наблюдения	664
Поиск информации	665
Исправление исполняемого модуля	665
Глава 4.3. Описание работы с дизассемблером W32Dasm и отладчиком SoftICE	666
Отладчик W32Dasm	666
Начало работы	666
Передвижение по дизассемблированному тексту	668
Отображение данных	669
Вывод импортированных и экспортированных функций	670
Отображение ресурсов	670
Операции с текстом	671
Загрузка программ для отладки	671
Работа с динамическими библиотеками	673
Точки останова	673
Модификация кода, данных и регистров	673
Поиск нужного места в программе	675
Отладчик SoftICE	676
Основы работы с SoftICE	677
Запуск и интерфейс	677
Краткий справочник по SoftICE	688
Глава 4.4. Основы анализа кода программ	712
Переменные и константы	712
Управляющие структуры языка C	717
Условные конструкции	717
Вложенные условные конструкции	717
Оператор <i>switch</i> или оператор выбора	718
Циклы	719
Локальные переменные	720
Функции и процедуры	722
Оптимизация кода	723
Объектное программирование	727

Глава 4.5. Исправление исполняемых модулей	732
Простой пример исправления исполняемого модуля	732
Пример снятия защиты	736
Стадия 1. Попытка зарегистрироваться	736
Стадия 2. Избавляемся от надоедливого окна	738
Стадия 3. Доводим регистрацию до логического конца.....	740
Стадия 4. Неожиданная развязка	741
Глава 4.6. Структура и написание драйверов	743
О ядре и структуре памяти	743
Управление драйверами	745
Пример простейшего драйвера, работающего в режиме ядра.....	747
Драйверы режима ядра и устройства	753
ПРИЛОЖЕНИЯ	767
Приложение 1. Справочник API-функций и сообщений Windows	769
Приложение 2. Справочник по командам и архитектуре микропроцессора Pentium	787
Регистры микропроцессора Pentium	787
Регистры общего назначения	787
Регистр флагов	788
Сегментные регистры.....	789
Управляющие регистры	789
Системные адресные регистры	791
Регистры отладки.....	791
Команды процессора.....	792
Команды арифметического сопроцессора	806
Расширение MMX.....	814
О новых инструкциях MMX.....	817
Приложение 3. Защищенный режим микропроцессора Pentium.....	819
Об уровнях привилегий	819
Селекторы	820
Дескриптор кода и данных	820
Другие дескрипторы.....	821
Сегмент TSS	822
О защите и уровнях привилегий	822

Привилегированные команды	822
Переключение задач	823
Страничное управление памятью	823
Приложение 4. Структура исполняемых модулей	825
Общая структура PE-модуля	826
Заголовок PE-модуля	828
Таблица секций.....	834
Секция экспорта (.edata).....	837
Секция импорта (.idata)	839
Локальная область данных потоков	841
Секция ресурсов (.rdata).....	842
Таблица настроек адресов	843
Отладочная информация (.debug\$S, .debug\$T)	845
Приложение 5. Файл kern.inc, используемый в главе 4.6	846
Приложение 6. Пример консольного приложения с полной обработкой событий.....	855
Приложение 7. Описание компакт-диска.....	865
Список литературы	867
Предметный указатель	869



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.EXE¹. 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 системах².

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 загрузаемую 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 библиотекар:

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 внешних⁶ 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 переменная⁷ 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 считите 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-